University of Crete
Computer Science Department

Defending against known and unknown attacks using a Network of Affined Honeypots

Spiros Antonatos

Doctoral Dissertation

Heraklion, October 2009

University of Crete
Computer Science Department

Defending against known and unknown attacks using a Network of Affined Honeypots

A dissertation submitted by Spiros Antonatos
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

THESIS APPROVAL

Committee:

Evangelos P. Markatos
Professor – Thesis Advisor

Angelos Bilas
Associate Professor

Vasilios A. Siris
Assistant Professor

Angelos Keromytis
Associate Professor

Maria Papadopouli
Assistant Professor

Dionisios Pnvmatikatos
Associate Professor

Sotiris Ioannidis
Associate Researcher

Department:

Dimitris Plexousakis
Professor – Chairman of the Department

Heraklion, October 2009

Abstract

Security is increasingly regarded as an essential function for maintaining a reliable, available, and trustworthy network infrastructure. Viruses, worms, trojans, spyware and other types of malicious programs are discouraging the effective use of the Internet and crippling the network infrastructure. The exponential increase of attack volume and the evolution of attacking methods urge the need for efficient and accurate defense mechanisms. While research and development has produced a multitude of security products, including firewalls, antivirus systems and intrusion detection systems, demand for better security is growing far beyond what current systems can offer.

Besides traditional network attacks, like worms and DoS attacks, new attack methods have appeared over the last few years. As the attack surface of remote exploits has been reduced due to advances in defenses and operating systems, new propagation methods are invented. Attackers, in their effort to adapt to security response, proactively explore and mitigate new threats. The World Wide Web is an excellent platform for launching attacks due to the vast number of users and plethora of technologies and architectures available for exploitation. This thesis discusses a new class of attacks that misuses the World Wide Web to create botnet-like infrastructures, which we call *puppetnets*. Puppetnets rely on websites that *coerce* web browsers to (unknowingly) participate in malicious activities. Such activities include distributed denial-of-service, worm propagation and reconnaissance probing, and can be engineered to be carried out in stealth, without any observable impact on an otherwise innocent-looking website. In this thesis we experimentally assess the threat from puppetnets. We discuss the building blocks for engineering puppetnet attacks and attempt to quantify how puppetnets would perform.

In an effort to defends against known and unknown attacks, this thesis presents the *NoAH infrastructure*, a Network Of Affined Honeypots geographically distributed around the world. Honeypots act like traps; their purpose is to lure attackers by monitoring unused portions of the IP address space or imitating the behavior of real users. Honeypots have been shown to be very useful for accurately detecting attacks, including zero-day threats, at a reasonable cost and without false positives, unlike intrusion and anomaly detection systems. The NoAH architecture is extensible enough to allow the detection of both known and unseen attacks. The NoAH infrastructure is not a centralized farm of honeypots. On the contrary, it is a distributed set of honeyfarms that collaborate. Deployed honeyfarms either forward traffic to a centralized set of honeypots, called the NoAH core, or provide attack statistics for analysis and correlation purposes. Services like automated signature generation for zero-day attacks and display of statistics also run inside the core. An approach to validate the signatures generated by the NoAH components is also proposed. Our approach, called *Sigval*, is able to test signatures against large volumes of benign traffic in the order of few seconds. Finally, the pilot deployment and operation of the NoAH infrastructure are also presented in this thesis.

The deployment of honeypots requires both administrative expertise and dedicated resources that many organizations cannot afford. Furthermore, the effectiveness of honeypots heavily depends on the unused IP address space they cover. Unused IP address space can be found in almost every organization, institution and public body due to underutilized or even totally empty subnets. In response to these problems, this thesis proposes *Honey@home*, a

new architecture that enables large-scale deployment at low-cost. The Honey@home architecture relies on communities of regular users installing a virtual honeypot that monitors unused addresses. The Honey@home tool is a lightweight daemon that automatically claims one or more unused IP addresses or ports and forwards the traffic directed to them to the NoAH core for further processing. The answers from the NoAH core are forwarded back to the attacker, thus providing the illusion to her that she communicates with a real service. The Honey@home approach is an excellent way to both extend the reachability and coverage of the NoAH infrastructure as well as to enable users unfamiliar with security technologies help in combating the cyber-crime.

Supervisor: Evangelos P. Markatos, Professor
Department of Computer Science
University of Crete

Περίληψη

Ανίχνευση Γνωστών και Άγνωστων Επιθέσεων με τη Χρήση ενός Δικτύου από Συνεργαζόμενα Honeypots

Σπύρος Αντωνάτος

Διδακτορική Διατριβή

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Η ασφάλεια υπολογιστών θεωρείται ολοένα και περισσότερο μια κρίσιμη λειτουργία για την διατήρηση μιας αξιόπιστης, διαθέσιμης και έμπιστης δικτυακής υποδομής. Ιοί, 'σκουλήκια' (worms), 'δούρειοι ίπποι' (trojans), καταγραφείς πληκτρολογίου (keyloggers) και άλλοι τύποι κακόβουλου λογισμικού απειλούν την αποτελεσματική χρήση του Διαδικτύου και ακρωτηριάζουν την δικτυακή υποδομή. Η εκθετική αύξηση του αριθμού των επιθέσεων και η εξέλιξη του τρόπου που εκτελούνται κάνουν επιτακτική την ανάγκη για πιο αποδοτικούς και ακριβείς μηχανισμούς ασφαλείας. Ενώ ο τομέας της έρευνας και ανάπτυξης έχει παράγει δεκάδες προϊόντα ασφαλείας, όπως αντιπυρικά τείχη (firewalls), αντιϊκά συστήματα (antivirus) και συστήματα ανίχνευσης επιθέσεων, η αναγκαιότητα για καλύτερη ασφάλεια μεγαλώνει και επεκτείνεται πέρα από αυτά που μπορούν να προσφέρουν τα τρέχοντα συστήματα.

Πέρα από τις παραδοσιακές δικτυακές επιθέσεις, όπως worms και επιθέσεις άρνησης υπηρεσίας (DoS), νέοι μέθοδοι επιθέσεων έχουνε εμφανιστεί τα τελευταία χρόνια. Καθώς η επιφάνεια επίθεσης των απομακρυσμένων exploits έχει μειωθεί χάρη στην αναβάθμιση των αμυντικών συστημάτων και των λειτουργικών συστημάτων, νέοι μέθοδοι διάδοσης έχουν εφευρεθεί. Οι επιτιθέμενοι, στην προσπάθεια τους να προσαρμοστούν στα αμυντικά συστήματα, εξερευνούν νέες απειλές. Ο Παγκόσμιος Ιστός είναι μια πρώτης τάξεως πλατφόρμα για την εξαπόλυση επιθέσεων εξαιτίας του τεράστιου αριθμού χρηστών και την πληθώρα τεχνολογιών και αρχιτεκτονικών που μπορούν να παραβιαστούν. Αυτή η διατριβή διαπραγματεύεται έναν νέο τύπο επιθέσεων που εκμεταλλεύονται τον Παγκόσμιο Ιστό για να δημιουργήσουν μια υποδομή από υπολογιστές-bots, τους οποίους καλούμε 'μαριονέτες' (puppetnets). Τα puppetnets βασίζονται σε ιστοσελίδες που αναγκάζουν τους browsers να συμμετάσχουν σε κακόβουλες δραστηριότητες χωρίς να το γνωρίζουν. Τέτοιες δραστηριότητες περιλαμβάνουν τις κατανεμημένες επιθέσεις DoS, εξάπλωση worms και ανίχνευση ανοιχτών πορτών, και μπορούν να δράσουν μυστικά, χωρίς καμία επίδραση σε μία, κατα τα άλλα, φαινομενικά αθώα ιστοσελίδα. Σε αυτή την εργασία μελετάμε πειραματικά το μέγεθος της απειλής. Συζητάμε τα βασικά δομικά στοιχεία για την κατασκευή puppetnets και προσπαθούμε να ποσοτικοποιήσουμε την απόδοση τους.

Στην προσπάθεια μας να αμυνθούμε από γνωστές και άγνωστες επιθέσεις, η διατριβή αυτή παρουσιάζει την υποδομή NoAH, ένα γεωγραφικά κατανεμημένο δίκτυο απο συνεργαζόμενους υπολογιστές-δολώματα (honeypots). Τα honeypots λειτουργούνε σαν παγίδες. Ο σκοπός τους είναι είναι να ανιχνεύσουν επιτιθέμενους παρακολουθώντας αχρησιμοποίητα κομμάτια του χώρου IP διευθύνσεων ή μιμούμενα την συμπεριφορά πραγματικών χρηστών. Τα honeypots έχουν αποδειχτεί χρήσιμα για τον ακριβή εντοπισμό επιθέσεων, συμπεριλαμβανομένων και άγνωστων α-

πειλών, με προσιτό κόστος και χωρίς λανθασμένους συναγερμούς, σε αντίθεση με τα παραδοσιακά σύστηματα ανίχνευσης επιθέσεων. Η αρχιτεκτονική του NoAH είναι αρκετά παραμετροποιήσιμη ώστε να επιτρέπει την ανίχνευση γνωστών και άγνωστων επιθέσεων. Η υποδομή του NoAH δεν είναι μια κεντρική φάρμα από honeypots. Αντιθέτως, είναι ένα κατανεμημένο σύνολο από φάρμες που συνεργάζονται. Οι φάρμες αυτές είτε προωθούν κίνηση προς ένα κεντρικό σύνολο από honeypots, τον πυρήνα του NoAH, ή παρέχουνε στατιστικά επιθέσεων για περαιτέρω ανάλυση και συσχέτιση με άλλα στατιστικά δεδομένα. Επιπρόσθετα, υπηρεσίες όπως αυτόματη παραγωγή υπογραφών για καινούργιες επιθέσεις και απεικόνιση στατιστικών τρέχουν στον πυρήνα του NoAH. Προτείνουμε μια προσέγγιση για τον έλεγχο ποιότητας των παραχθέντων υπογραφών. Η προσέγγιση μας, με το όνομα Sigval, μπορεί να δοκιμάσει υπογραφές πάνω από τεράστιους όγκους αθώας κίνησης μέσα σε λίγα δευτερόλεπτα. Παρουσιάζουμε επίσης την πιλοτική εγκατάσταση και λειτουργία της υποδομής του NoAH.

Η εγκατάσταση και συντήρηση honeypots απαιτεί τόσο διαχειριστικές ικανότητες όσο και αφοσιωμένους πόρους που αρκετοί οργανισμοί δεν μπορούν να διαθέσουν. Επιπρόσθετα, η αποτελεσματικότητα των υπολογιστών-δολωμάτων εξαρτάται από το μέγεθος του χώρου IP διευθύνσεων που παρακολουθούν. Αχρησιμοποίητος χώρος IP διευθύνσεων μπορεί να βρεθεί σε κάθε οργανισμό και ίδρυμα εξαιτίας της ύπαρξης άδειων subnets ή subnets που χρησιμοποιούνται σε μικρό ποσοστό. Για την αντιμετώπιση αυτών των προβλημάτων, η διατριβή αυτή προτείνει το Honey@home, μια νέα αρχιτεκτονική που επιτρέπει την εγκατάσταση honeypots σε ευρεία κλίμακα με χαμηλό κόστος. Η αρχιτεκτονική του Honey@home βασίζεται σε κοινότητες απλών χρηστών που εγκαθιστούν ένα εικονικό honeypot το οποίο παρακολουθεί αχρησιμοποίητες IP διευθύνσεις. Το εργαλείο Honey@home είναι μια ελαφρία διεργασία που αυτόματα καταλαμβάνει μία ή περισσότερες IP διευθύνσεις και προωθεί όλη τη κίνηση προς αυτές στον πυρήνα του NoAH για περαιτέρω επεξεργασία. Οι απαντήσεις από τον πυρήνα του NoAH προωθούνται πίσω στους επιτιθέμενους, δημιουργώντας τους την ψευδαίσθηση ότι επικοινωνούνε με μια πραγματική υπηρεσία. Η προσέγγιση του Honey@home είναι ένας εξαιρετικός τρόπος για να επεκταθεί η κάλυψη που προσφέρει η υποδομή του NoAH καθώς και να επιτρέψει σε χρήστες που δεν είναι γνώστες του τομέα ασφάλειας δικτύων να βοηθήσουν στην καταπολέμηση του κυβερνο-εγκλήματος.

Επόπτης: Καθηγητής Ευάγγελος Μαρκάτος

Acknowledgments

I would like to deeply thank my supervisor, Evangelos P. Markatos, for his great support and feedback. His simple way of thinking and broad perspective will be a guide to my academic route. I would also like to express my deep gratitude to Kostas Anagnostakis, a valuable partner whose contribution was a key for writing this thesis. It is truly an unprecedented experience to work with these people.

I would also like to deeply thank my friends and colleagues Michalis Polychronakis, Dimitris Antoniadis, Iasonas Polakis, Christos Papachristos, Elias Athanasopoulos, Manolis Stamatogiannakis, Antonis Papadogiannakis and George Kondaxis for their support and hilarious moments at the DCS lab.

The ex-DCS members and friends Dimitris Koukis, Manos Athanatos, Michalis Foukarakis, Vassilis Pappas and Giorgos Vasiliadis for their help, support and creative input to my work.

The rest of the DCS members Alexandros Kapravelos, Irini Xaritaki, Thanasis Petsas, Eleni Gessiou, John Velegrakis, Harris Papadakis and Apostolis Zarras as well as the non-DCS members Dimitra Zografistou and Mary Koutraki for their help and support.

The friends I made in Computer Science Department and Heraklion, George Pikrakis, Kostas Terzakis, Manos Agapoulakis, Stelios Agapoulakis, Dimitris Kampas and Efi Kalaitzi.

My old friends Makis Anastasakis, Kostas Brentas, Maria Papathanasopoulou and Katerina Lagoudaki who all stood by me since I was a kid.

The ISPs that provided me with useful data for this work and the admins that have tolerated me for so many years.

Last but most important of all, my mother. Her love, help and support is truly unimaginable.

To my mother

Contents

List of Figures

Chapter 1

Introduction

In the last few years, we have been witnessing an increasing number of cyberattacks on the Internet. Viruses, worms, trojans, spyware and other types of malicious programs are discouraging the effective use of the Internet and crippling the network infrastructure. We can provide endless examples of worms and viruses that are fast and can cause severe damage. The SQL "Slammer" worm was able to infect more than 70,000 victim computers in less than 15 minutes. During the summer of 2003, the Blaster worm managed to infect more than 400,000 computers. In 2001, more than 4,000 Denial-of-Service (DoS) attacks were launched on the Internet every week[130], often attracting the interest of popular media. Although the problem of Denial-of-Service attacks was already widely known, the magnitude of the problem had been largely underestimated. It is difficult to measure the damage caused by viruses and worms, however some estimates put the cost in the order of billions of dollars[166]. However, this damage may actually be small compared to what these attacks can potentially do, as illustrated in a study on so-called "Warhol" worms[179]. Such worms can cause massive damage of unprecedented effect causing severe disruption to the Internet infrastructure and services.

Besides worms and DoS attacks, new attack methods have appeared. As the attack surface of remote exploits has been reduced due to advances in defenses and operating systems, new attack vectors are invented. Attackers, in their effort to adapt to security response, proactively explore and mitigate new threats. The World Wide Web is an excellent platform for launching attacks due to the vast number of users and plethora of technologies and architectures available for exploitation. The misuse of the World Wide Web for performing attacks like worm propagation, Denial-of-Service attacks and reconnaissance probes is demonstrated in this dissertation. We call this threat *puppetnets*. Puppetnets rely on websites that coerce web browsers to unknowingly participate in malicious activities. We demonstrate that puppetnets can form powerful botnet-like infrastructures.

Although there exist tools and systems that can help us protect our infrastructure from cyber-attacks, these tools are usually limited to combating only known forms of attacks. Antivirus systems can protect users against known viruses, but are usually helpless when confronted with a new computer virus. Similarly, Intrusion Detection Systems can generate alerts for known forms of worms but are of little help when confronted with a previously unknown attacks. Thus, we need a security infrastructure that is able to detect new forms of attacks, and allows scientists and engineers to study, analyze and rapidly develop defenses, and has the critical mass to detect new types of attacks as early as possible. In this dis-

sertation, we propose the *Network of Affined Honeypots (NoAH)*, a scalable and extensible architecture based on the use of a distributed system of affined honeypots. Honeypots are non-production systems that monitor unused resources, such as unused IP address space. By default, honeypots should not receive any kind of activity thus any communication between a host and them is flagged as suspicious and is examined. The value of honeypots relies on the fact that they can detect novel attacks without any false positives, unlike traditional defense mechanisms such as network intrusion and prevention systems. There are cases where there is need for fine-grained deployment of honeypots. Shadow honeypots[60] have shown how honeypots can be used to protect servers, for example servers that are victims of Puppetnet attacks, by examining a mirror of the traffic they receive and is considered suspicious. To facilitate the deployment of such scenarios we propose NoAH as an affordable architecture to run honeypot services.

The architecture of NoAH presents a flexible design for deployment and collaboration of honeypots. NoAH is not restricted to a single type of honeypot but tries to combine the good characteristics of all honeypot types. Inside the NoAH core, low-interaction honeypots act as filtering mechanisms to dispose of any uninteresting traffic. Medium and high-interaction honeypots perform the attack detection and provide all necessary information to the prevention and signature generation modules. NoAH's modular architecture permits the construction of a network of honeypots with minimal overhead and affordable administrative overhead and allows the participation of cooperating organizations and home users. The goal of NoAH is to automatically generate signatures for novel attacks. However, as signatures may produce false positives, there is a need to validate them against benign traffic. This dissertation proposes *Sigval*, a novel approach to validate signatures against large amount of data in the order of seconds. Sigval provides all the necessary mechanisms to protect network administrators that want to deploy new signatures from the danger of hoaxes. As the source of signatures cannot be identified in most of the cases, Sigval can help identify whether a signature may block legitimate traffic or not.

The high administrative overhead and maintenance cost makes the installation of honeypots prohibitive for many organizations and institutions. To alleviate this problem, the Honey@home approach is proposed. The Honey@home tool is a lightweight tool that automatically claims one or more unused IP addresses and forwards all traffic destined to them to the honeypots of the core (and the responses from the honeypots back to the attacker). The advantages of Honey@home are twofold. First, the minimal installation and maintenance overhead of Honey@home enables the deployment of a honeypot in any environment, from home computers to large institutions and organizations. This allows us to monitor diverse environments and assess threats for multiple network domains. Second, Honey@home avoids the problem of honeypot identification. Unlike other honeypot installations that are susceptible to be identified and thus blacklisted, the Honey@home approach is dynamic. As Honey@home users come and go and as the monitored addresses are dynamically allocated, it is extremely difficult for the attacker to create a blacklist of potential Honey@home addresses. We evaluate the identification risk and effectiveness of our proposed approach.

1.1 Technical challenges

The cyber-attacks landscape is constantly changing and evolving. The arms race between attackers and defenders is an open war that has become more complex than ever. While in the

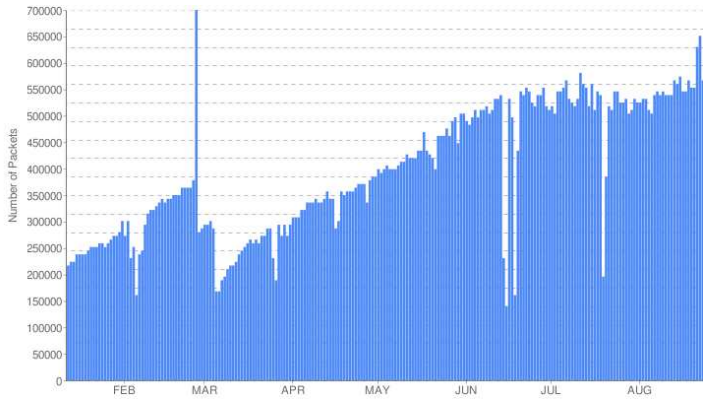


Figure 1.1: Number of packets per day for six months in the life of NoAH

past simple defense mechanisms like signature-based Intrusion Detection Systems (IDS)[145, 126, 61, 64] were able to detect and prevent most of the attack instances, they are proved to be inefficient for current attack vectors. We can pin down three main challenges that are the driving forces behind our contributions.

First, the volume of attacks becomes larger and larger. Most users have broadband connectivity with high bandwidth. While a botnet in the late 90s had limited firepower, nowadays botnets has significant power, delivering Denial-of-Service in the order of Gigabits/sec. Our observations from our honeypots have verified this trend. Figure 1.1 displays the volume of traffic we have received over a period six months. *The scalability of defense mechanisms is of critical importance in order to deal with the ever-increasing traffic and attack volume. Scalability allows faster detection and reaction as well as more detection modules to be plugged in.*

Second, the complexity of attacks has increased significantly. Techniques like polymorphism and metamorphism have rendered traditional defense mechanisms useless or unable to detect and react. Early versions of attacks were static; that allowed defense mechanisms to generate signatures that could accurately detect and prevent these attacks. Latest attacks have shown that polymorphism is heavily used and no signature can be generated to detect attacks at the network level. *Deployed defense mechanisms must be configurable enough to cope with the complexity of attacks without building their architecture around specific attack characteristics.*

Third, the attack surface has expanded to multiple domains. In the early years of Internet, remote exploitation vectors targeted only a few services specific to a limited number of operating systems. As an example, worms like Blaster[68] and Sasser[98] all exploited the RPC vulnerabilities of Windows operating system. However, the number of targeted services has increased dramatically. The World Wide Web and applications that surround it are popular targets and latest statistics have verified this trend. *Deployed defense mechanisms must be extensible enough to allow detection plug-ins that will deal with new forms of attacks.*

1.2 Thesis statement

The attack surface of remote exploits is gradually diminishing. The adoption of defense mechanisms from operating systems vendors, such as address space randomization and non-executable stacks, has raised the bar and forces the attackers to change their methodologies. *Even in the case where remote exploits are eliminated either through the adoption of defense mechanisms or improved programming techniques, attacks as we know them today can still exist but in other forms. This work documents, demonstrates and evaluates such a new form of attacks using existing and very popular platforms.* Puppetnets have shown that attacks like worm propagation, Denial-of-Service attacks, scanning, malicious computations and spamming can be achieved *without the need of compromised machines* but based on the bad design of the most widely-used platform, the World Wide Web. Our evaluation has shown that Puppetnet attacks are very powerful and we propose a set of mechanisms to defend against them. The use of honeypots is a defense measure for botnet and puppetnet attacks and is presented in this thesis through the Network of Affined Honeypots.

1.3 Summary of contributions

The contribution of the research presented in this dissertation are as follows.

- A new class of attacks that misuses the World Wide Web architecture called *puppetnets* is introduced. Puppetnets are able to propagate worms, perform Denial-of-Service attacks and do reconnaissance probes by using the resources of Web users.

- An evaluation of the effect of puppetnets is presented as well as defenses against them. We show that attackers are able to create powerful botnet-like infrastructures that can cause significant damage. We explore the effectiveness of countermeasures including anomaly detection and more fine-grained browser security policies.

- The design and implementation of the Network of Affined Honeypots (NoAH), a honeypot infrastructure for detecting known and unknown cyber-attacks. NoAH combines multiple honeypot technologies to form a scalable and efficient architecture. Furthermore, it is extensible enough to permit cooperating organizations to participate.

- The pilot-demonstration and operation of the NoAH infrastructure. We have deployed geographically distributed sensors that daily record attack conversations and capture malware instances. We have built a web interface to view gathered statistics and monitor the health of the infrastructure.

- The design and implementation of Honey@home. Honey@home enables users that are unfamiliar with honeypot technologies to install a honeypot to their personal computer with minimal overhead and zero maintenance cost.

- *Sigval*, a validation algorithm to facilitate the qualitative evaluation of the signatures generated by the defense mechanisms of the NoAH infrastructure. Sigval is able to test signatures against large amount of data in the order of few seconds.

1.4 Organization of dissertation

This dissertation is organized as follows. In Chapter 2, an overview of cyber-attacks as well as honeypot technologies and systems is presented. Chapter 3 introduces *puppetnets*, a new class of attacks that misuses the World Wide Web to propagate worms, perform Denial-of-Service attacks and do reconnaissance probes. In Chapter 4 we provide a detailed description and evaluation of NoAH, a scalable honeypot infrastructure for detecting cyber-attacks. The Honey@home approach to extend the NoAH reachability and enable normal users to participate is presented in Chapter 5. Chapter 6 is a description and evaluation of an approach to validate signatures created by the mechanisms of NoAH. The related work to the attacks, defenses and evaluation methodologies proposed in this dissertation is presented in Chapter 7. We discuss and provide some concluding remarks in Chapter 8.

Chapter 2

Background

The aim of this Chapter is to provide some basic background knowledge about the security attacks we deal in this thesis as well as the most well-known honeypot systems and technologies that help us defend against such threats.

2.1 Worms

Worms are malicious self-replicating programs that need no human intervention. They exploit vulnerabilities that can be remotely triggered so as to compromise other hosts, copy themselves and continue their propagation style. As worms are completely automated, they can propagate faster than any human can react. Their severity and attack speed has attracted the interest and focus of many researchers.

Computer viruses have been studied extensively over the last couple of decades. Cohen was the first to define and describe computer viruses in their present form. In [85], he gave a theoretical basis for the spread of computer viruses. The strong analogy between biological and computer viruses led Kephart *et al.* [111] to investigate the propagation of computer viruses based on epidemiological models. They extend the standard epidemiological model by placing it on a directed graph, and use a combination of analysis and simulation to study its behavior. They conclude that if the rate at which defense mechanisms detect and remove viruses is sufficiently high relative to the rate at which viruses spread, it is possible to prevent widespread virus propagation.

The Code Red worm [80] was analyzed extensively in [185]. The authors conclude that even though epidemic models can be used to study the behavior of Internet worms, they are not accurate enough because they cannot capture some specific properties of the environment these operate in: the effect of human countermeasures against worm spreading (*i.e.,* patching, filtering, disconnecting, *etc.*) and the slowing down of the worm infection rate due to the impact of worm on Internet traffic and infrastructure. They derive a new general Internet worm model called *two-factor worm* model, which they then validate in simulations that match the observed Code Red data available to them. Their analysis seems also to be independently supported by the data on Code Red propagation in [129].

A similar analysis on the SQL "Slammer" (or Sapphire) worm [51] can be found in [53]. Sapphire, the fastest worm today, was able to infect more than 70,000 victim computers in less than 15 minutes.

The Blaster/Welchia epidemic is an interesting example of a "vigilante" worm (Welchia)

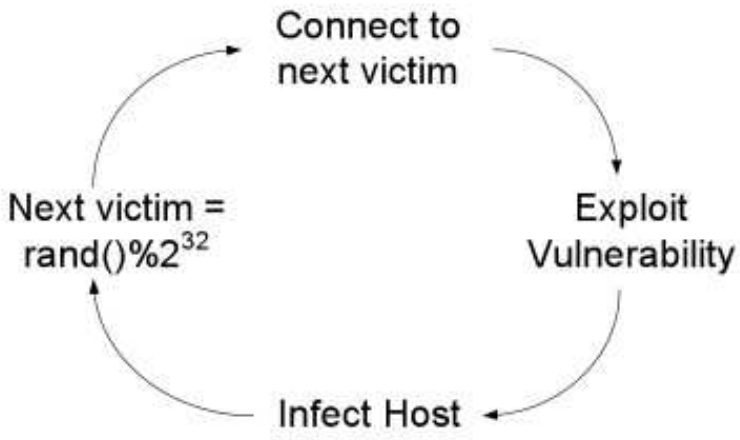


Figure 2.1: A simple propagation scheme for a worm

causing more trouble than the original outbreak (Blaster). A "vigilante" worm attempts to clean-up another worm by using the same vulnerability. However, the very notion of "vigilante" worms is rendered useless if worms immediately disable the vulnerability after compromising a machine.

The Witty worm [151] is of interest for several reasons. First, it was the first widely propagated Internet worm to carry a destructive payload. Second, Witty was started in an organized manner with an order of magnitude more ground-zero hosts than any previous worm and also began to spread as early as only one day after the vulnerability was publicized, which is an indication that the worm authors had already prepared all the worm infrastructure, including the ground-zero hosts and the replication mechanisms, and were only waiting for an exploit to become available in order to launch the worm. Finally, Witty spread through a population almost an order of magnitude smaller than that of previous worms, showing that a hitlist is not required even for targeting small populations.

All these worms use (random) scanning to determine their victims, by using a random number generator to select addresses from the entire IP address space. A simple propagation scheme for a worm that chooses its next target uniformly is shown in Figure 2.1. Although some worms chose their next target uniformly among all the available IP addresses, other worms seemed to prefer local addresses over distant ones, so as to spread the worm to as many local computers as possible. Once inside an organization, these worms make sure that they will infect several of its computers before trying to infect any outside hosts.

2.2 Web-based attacks and worms

The vast number of web users and the decrease in the attack surface of remote exploits has made the World Wide Web a very attractive platform for attack propagation. Attackers have successfully taken over browsers in the past to gain control of the user's machine. The WMF and JPEG vulnerabilities ([24],[23]) have shown how the Internet Explorer browser can be compromised and execute arbitrary code on the victim's side. Other attacks, like the carpet bombing attack against the Safari browser[36], can also lead to arbitrary code execution. All

these attacks are called drive-by downloads as they download malicious code to the user's machine without knowledge of the user.

Apart from the exploitation techniques, attackers also take advantage of the new technologies used in the World Wide Web and browser features. Many web sites make extensive use of client-side scripts (mostly written in JavaScript) to enhance user experience. Unfortunately, this trend has also increased the popularity and frequency of cross-site scripting (XSS) attacks. XSS vulnerabilities constitute one of the most serious threats to the security of modern web applications. According to [25], XSS vulnerabilities were the most commonly reported vulnerabilities for the years 2005 and 2006. They allow an attacker to inject malicious content into web pages served by trusted web servers. Since the malicious content runs with the same privileges as trusted content, the malicious content can steal a victim user's private data or take unauthorized actions on the user's behalf.

The XSS vulnerabilities can be used to do several things. One of the most common cases is to steal user's credentials. Such an attack targeted the Hotmail service[95], one of the most popular web e-mail services. Another form of attack is the Cross Site Request Forgery (CSRF). CSRF works by exploiting the trust that a site has for the user. Site tasks are usually linked to specific URLs, for example `http://site/stocks?buy=100\&stock=ebay` allowing specific actions to be performed when requested. If a user is logged into the site and an attacker tricks their browser into making a request to one of these task URLs, then the task is performed and logged as the logged in user.

XSS attacks can also be used to launch web worms, an attack closer to the focus of this work. The Samy worm[40], also known as MySpace worm, one of the fastest spreading worms to date, exemplified this. The worm carried a payload that would display the string "but most of all, Samy is my hero" on a victim's profile. When a user viewed that profile, they would have the payload planted on their page. Within just 20 hours of its October 4, 2005 release, over one million users had run the payload, making Samy one of the fastest spreading viruses of all time[8]. Other than the Samy worm, several XSS worms have been executed or proofs of their concepts have been released. It seems that social networking sites are popular targets, mainly due to their large user base. An XSS worm also hit the Orkut social network[31], infecting approximately 400,000 users. Another popular network with millions of users, Twitter.com, was also affected by XSS worms[42]. In the case of Twitter four variants of the web worm were discovered. Another example is the Justin.tv worm. Justin.tv is a video casting website, which shows an active user base of approximately 20 thousand users. The cross-site scripting vulnerability that was exploited was caused by the lack of output sanitization within a "Location" profile field. Since the social website that was targeted was not particularly active (compared to other popular XSS worm targets), the worm infected a total of 2525 profiles within roughly 24 hours. Finally, even web-based games, like GaiaOnline, were hit by XSS worms[50].

2.3 Honeypots

A formal definition of a honeypot is a "trap set to detect, deflect or in some manner counteract attempts at unauthorized use of information systems"[1]. Practically, honeypots are computer systems set up to lure attackers. They are non-production systems, which means machines that do not belong to any user or run publicly available services. Instead, in most cases, they

[1] http://en.wikipedia.org/wiki/Honeypot_%28computing%29

passively wait for attackers to contact them. By default, all traffic destined to honeypots is malicious or unauthorized as it should not exist in the first place. Honeypots can also assume other forms, like files, database records, or e-mails.

The original goal of honeypots was to detect worms. As worms cannot afford to acquire the knowledge whether their next victim is a honeypot or not, they blindly try to attack the whole IP address space. Based on that behavior, honeypots are able to catch instances of worms by running decoy services, analyze them and pass all the gathered information to signature generation systems and behavior analysis modules. However, as the attack landscape has changed and the existence of automated worms has become more rare than ever, the use of honeypots has also evolved. Honeypots can also be used to detect malicious web traffic. The "Google Hack" honeypot[11] is a perfect example of how a honeypot can provide reconnaissance against attackers that use search engines as a hacking tool. Furthermore, honeypots are flexible enough to provide any kind of emulation and interactivity and become a valuable tool for discovering new flaws in all kind of applications, including web applications. The Project Honeypot[33], the Web Application Security Consortium (WASC) Open Proxy Honeypot[48], the DShield Web Honeypot project[9], the Honeypot-PHP software[17] for detecting PHP flaws and the HiHAT analysis toolkit[12] for SQL injection, XSS and remote file inclusion detection are few of the steps done towards this direction. Production systems and large antivirus companies also use honeypots to acquire cyber-attack information.

Several honeypot designs have been proposed. The two main axes on which a honeypot is designed is the level of interactivity with the attackers and the side we want to protect. Concerning the level of interactivity, honeypots can either do simple service emulation (low-interaction), more advanced emulation (medium-interaction) or run real services (high-interaction). As far as the second axis is concerned, we can categorize honeypots as server-side and client-side. Most honeypots protect the server side but client-side honeypots follow a different approach. Instead of waiting to be attacked, they search for attackers. Client-side can either crawl the World Wide Web, IM networks or IRC networks to spot malicious pages and communication channels. In the following Sections, we present an overview of the most well-known honeypot systems and technologies for each of the honeypot categories.

2.3.1 Low-interaction honeypots

A low-interaction honeypot tries to emulate real services and features of operating systems, usually through specialized components. The emulation components are designed to mimic real software but their scope is limited. The main advantage of low-interaction honeypots is that they are very lightweight processes and their emulated services are unlike to be infected as they are usually scripts or responders that imitate real responses. Their main disadvantage is that emulation is inaccurate and can only detect previously known attacks.

Honeyd

The most popular low-interaction honeypot is honeyd [142] . Honeyd is in fact a framework for building honeypots. Honeyd is able to create arbitrarily large virtual networks. In order to achieve this, honeyd comes with a number of modules that respond to ARP requests to claim addresses, request addresses from DHCP servers and finally components that emulate latency and loss characteristics of the network. A virtual topology created with honeyd is configurable. The user can define the range of IP address space that will be handled by

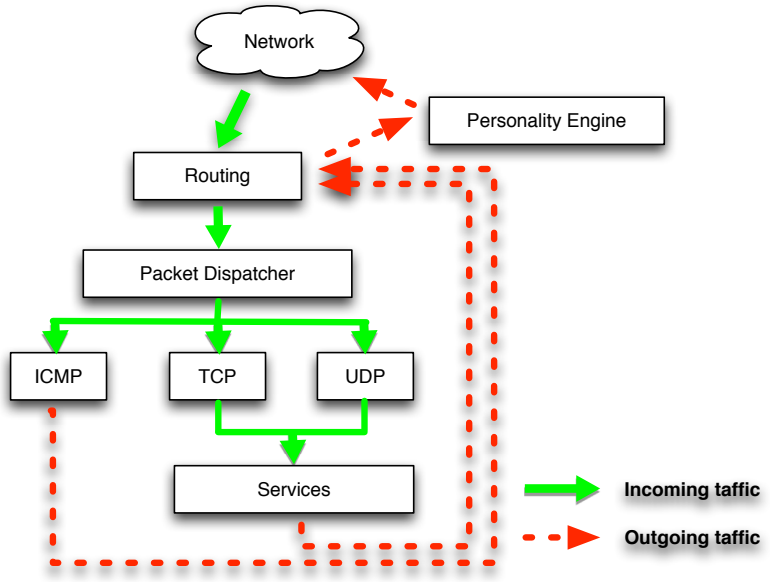


Figure 2.2: Architecture of honeyd. Green (solid) lines indicate how incoming traffic is handled. The routing process sends the packets to the packet dispatcher which decides on which handler to deliver the packet to based on the protocol. Handlers send the traffic to specific services based on the destination port. The responses from the services (red dotted lines) are sent back to the routing process, which consults the personality engine before injecting them to the network.

honeyd, the number and type of virtual hosts and their communication characteristics. An example of honeyd configuration is as follows

```
set windows personality "Windows XP Home Edition Service Pack 2"
add windows TCP port 80 "scripts/web.sh"
bind 10.1.0.2 windows
```

In the example above, honeyd will claim that in the IP address 10.1.0.2, a machine running Windows XP Service Pack 2 exists. Additionally, this virtual host will have port 80 open and the port will be handled by the "web.sh" script. All emulation is done through scripts, usually written in the Perl language. A very interesting property of honeyd is its ability to imitate network stacks of various operating systems. Honeyd does not bind to any sockets; instead it performs network stack emulation, thus it is highly scalable and can listen to arbitrarily large address spaces. Network stack emulation is more advanced than simply maintaining state for each connection. As the implementation of network protocols do not confront to standards, each operating system's network stack fills certain fields in their own way, for example the IP identification number or TCP timestamps. Honeyd maintains a database

of network stack characteristics, originally created by p0f[32] tool. These profiles are called *personalities*. When it needs to respond on behalf of a virtual host that "runs" Windows XP SP2, the database is advised and packets are formed in such a way to resemble those sent by a normal SP2 operating system. This property makes the virtual honeypot resilient to remote OS detection scans, like the ones performed by nmap[29] tool. The architecture of Honeyd is shown in Figure 2.2. Once a packet is received, it passes through the packet dispatcher. The dispatcher sends the packet to the appropriate services based on the configuration. Before the response from the service is sent to the network, the personality and configuration engines are invoked to determine the appropriate transfer protocol characteristics.

Honeytrap

Honeytrap[19], developed by TillmanWerner, is a low-interaction honeypot with several interaction capabilities. Differently from other approaches, Honeytrap is not bound a priori to a set of ports. It takes advantage of sniffers or user-space hooks in the netfilter library to detect incoming connections and bind consequently to the required socket. Each inbound connection can be handled according to 4 different operation modes:

- Service emulation. It is possible to take advantage of responder plugins similarly to what happens with honeyd.
- Mirror mode. When enabling mirror mode for a given port, every packet sent by an attacker to that port is simply mirrored back to the attacker. This mode is very functional, and is based on the assumption that, in case of a self-propagating worm, the attacker must be exposed to the same vulnerability that he is trying to exploit.
- Proxy mode. Honeytrap allows to proxy all the packets directed to a port or set of ports to another host, such as a high interaction honeypot.
- Ignore mode. Used to disable TCP ports that should not be handled by honeytrap.

Honeytrap takes advantage of a set of plugins to exploit the information collected by the network interaction. The ability of these plugins to handle network attacks can be assimilated to a best effort service. For instance, if an URL appears in the network stream, honeytrap will try to download the file located at that URL. If the URL is not directly present in the network stream, but is embedded within an obfuscated shellcode, honeytrap will not be able to detect it or download it. Thus, Honeytrap view on the network attacks is thus not uniform, but heavily depends on their structure and their complexity.

LaBrea

LaBrea[20] is a program that listens to an unused IP address space and answers connection attempts in such way that attackers at the other end get stuck. The original purpose of this tool was to slow down scanning from machines infected by the CodeRed worm. LaBrea claims unused IP address space by responding to ARP requests that remain unanswered (similar to farpd). When a SYN packet is destined for the claimed IP addresses, a SYN-ACK that tarpits the connection attempt is sent back. LaBrea also tries to mimic normal machines but in a limited fashion, e.g. it can respond to ping and send RST to SYN-ACK.

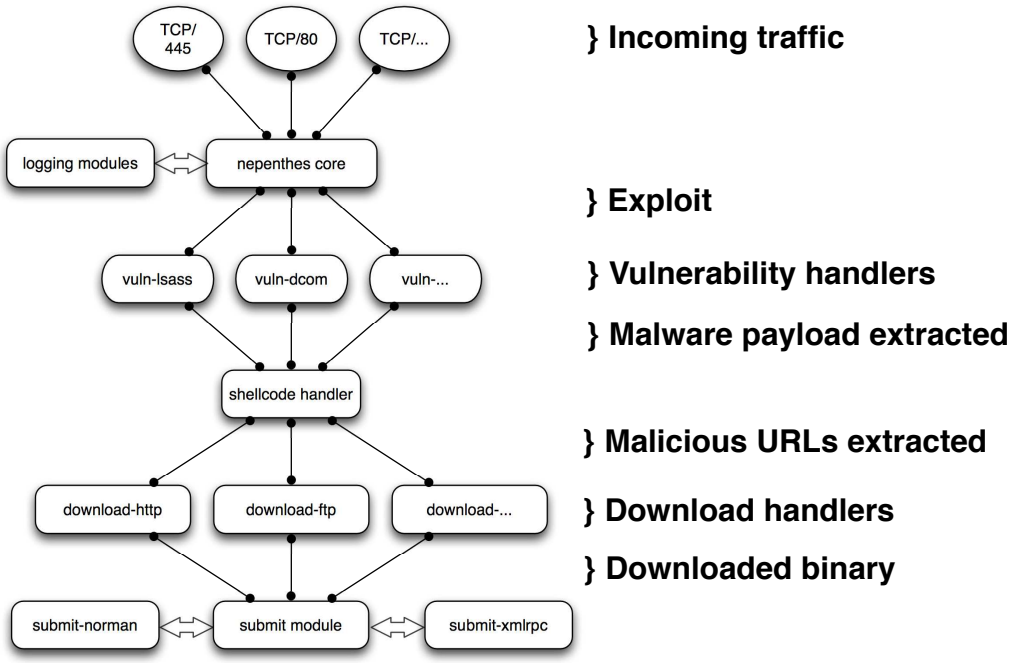


Figure 2.3: The architecture of Nepenthes. Incoming traffic is handed over to the appropriate vulnerability handler based on the destination port. Vulnerability handlers extract the expoit and invoke the shellcode manager. The shellcode manager emulates the shellcode and extracts the URLs it contains. The URLs are given to download handlers who actually download the malicious binary. Downloaded binaries are either stored locally or/and submitted to a central repository.

2.3.2 Medium-interaction honeypots

Medium-interaction honeypots also emulate services but, unlike low-interaction honeypots, they do not manage network stacks and transport protocols themselves. Instead, they bind to sockets and let the operating system do the connection management. In contrast to systems like honeyd, which implement network stacks and transport protocols, they focus more on the application-level emulation part. The most-well known medium-interaction honeypot is Nepenthes.

Nepenthes

The Nepenthes platform[28] is a system that was designed to automatically collect malware[2]. Its functionality is based on five types of modules: vulnerability, shellcode parsing, fetching, logging and submission modules. Vulnerability modules emulate the vulnerable services, like a DCOM service or a WINS server. Shellcode parsing modules analyze the payload received by vulnerability modules and try to extract information about the propagating malware. If such information is found, fetch modules download the malware from the designated destination and finally the malware is submitted to a central service (disk, database, anti-virus company)

[2]Historically, Nepenthes is the evolution of mwcollect platform

through the submission modules. The whole process is logged by the logging modules. For the time being, only sixteen vulnerability modules have been implemented for well-known exploits, like buffer overflow in Microsoft RPC services, buffer overruns in SQL server 2000 and exploits in the LSASS service. Nepenthes was originally designed to capture malware that spreads automatically, like Blaster or Slammer worms who were targeting hosts blindly.

The host running Nepenthes listens to several ports on one or more black IP addresses. The assignment of these addresses to this host and creation of virtual interfaces in order to have multiple IP addresses to a single interface must be done by the administrator manually. As the host running Nepenthes listens to many open ports, it is vulnerable to detection. The workflow of Nepenthes is shown in Figure 2.3. After a connection is established to one of the open ports, the payload of the packets of this connection is handled by the appropriate module. The main restriction here is that for each open port we can only have one vulnerability module. This means that for example we cannot emulate vulnerabilities for both Apache and IIS simultaneously. Vulnerability modules do not provide full service emulation but only emulate the necessary parts of the vulnerable service. When the exploitation attempt has arrived, the shellcode parsing modules analyze the received payload. In most cases, this parsing involves an XOR decoding of the shellcode and then some pattern matching is applied, like searching for URLs or strings like "CreateProcess". If a URL is detected, fetch modules download the malware from the remote location. These modules implement HTTP, FTP, TFTP and IRC-based downloads. However, a shellcode parser can be more complicated. Some malware can, for example, open command shells and wait for commands or bind to sockets. Shell emulation modules of Nepenthes provide command emulation for the virtual shells. Most shell commands are trivial, like echo or START directives.

The Nepenthes platform has evolved to a distributed network of sensors. Institutions and organizations participate in the mwcollect alliance, where all binaries captured are submitted to a central repository, accessible to all members of the alliance.

Multipot

Multipot[26] is a medium-interaction honeypot for the Windows platforms. Multipot follows the same design as Nepenthes. It emulates six vulnerabilities (among them the well-known MyDoom[27] and Beagle[4]). When multipot receives a shellcode, five shellcode handlers try to emulate it. Most common handlers are: `recv_cmd` for shellcodes that bind cmd to a port and receive commands, `recv_file` for shellcodes that open a port and receive a file and finally `generic_url` that performs XOR decryption and extracts HTTP, FTP, TFTP and echo strings. If a location is detected, the malware is downloaded. Multipot sets a maximum download size limit of 3 Megabytes to avoid malware that try to exhaust disk space.

Billy Goat

Billy Goat[107] is a honeypot developed by IBM Zurich Research Labs that focuses on the detection of worm outbreaks in enterprise environments. It is thus called by the authors Worm Detection System (WDS) in opposition to classical Intrusion Detection Systems. Billy Goat automatically binds itself to any unused IP address in a company network, and aims at quickly detecting and identifying the infected machines and retrieve information on the type of activity being performed.

In order to gather as much information as possible on the kind of activity observed by the

sensors, Billy Goat employes responders that emulate the application level protocol conversation. While the general responder architecture is very similar to that employed by Honeyd, Billy Goat takes advantage of a solution for the emulation of SMB protocols, that consists in taking advantage of a hardened version of the open-source implementation of the protocol[37]. Such an implementation is derived from the work done by Overton[135], that takes advantage of a similar technique to collect worms trying to propagate through open SMB shares. This choice puts Billy Goat in a hybrid position between low and high interaction techniques, since it offers to attacking clients the real protocol implementation, even if hardened to reduce security risks. The increased level of interaction of this technique has allowed interesting analyses such as the work done by Zurutuza in [186].

2.3.3 High-interaction Honeypots

High-interaction honeypots, unlike the two previous categories, do not emulate services. On the contrary, they run services in their native environment. This allows the maximum interactivity with attackers as vulnerabilities are not emulated but actually exploited. Recent advances in virtualization allows the creation of scalable and secure high-interaction honeypots. Operating systems of honeypots are running inside a virtual machine, like VMware[46], Qemu[72] and Xen[70]. This choice was made for two reasons. First, we can run multiple virtual machines in a single physical machine. Thus, we can run hundreds of high-interaction honeypots with a limited number of physical machines. Second, as high-interaction honeypots are vulnerable to being compromised, virtual machines can act as a containment environment. Once the vulnerability is exploited, the honeypot can be used as an attack platform to propagate worms or launch DoS attacks. Instrumented versions of virtual machines can be used as a containment mechanism to prevent vulnerabilities from being exploited and in parallel maintain the highest level of interaction.

In this Section, we will describe three high-interaction honeypots, all based on virtual machine technology.

Minos

Minos[89] is a microarchitecture that implements Biba's low water-mark integrity policy[76] on individual words of data. Minos stops attacks that corrupt control data to hijack program control low. Control data is any data that is loaded into the program counter on control-low transfer, or any data used to calculate such data. The basic idea of Minos is to track the integrity of all data and protect control by checking this integrity when a program uses the data for control transfer

Minos requires only a modicum of changes to the architecture, very few changes to the operating system, no binary rewriting, and no need to specify or mine policies for individual programs. In Minos, every 32-bit word of memory is augmented with a single integrity bit at the physical memory level and the same for the general-purpose registers. This integrity bit is set by the kernel when the kernel writes data into a user process memory space. The integrity is set to either "low" or "high" based upon the trust the kernel has for the data being used as control data. Biba's low water-mark integrity policy is applied by the hardware as the process moves data and uses it for operations. If data with "low" integrity is going to be executed, then an attack is taking place.

Minos was emulated on the Bochs Pentium emulator. The software Minos emulator is able

to execute 10 million instructions per second on a Pentium 4 running at 2.8GHz. The emulated Minos architecture runs for nearly two years without any false positives. Furthermore, various exploits have been launched against Minos to check his detection capabilities. These attacks tried to exploit various types of vulnerabilities such as format string, heap globbing, buffer overflow, integer overflow, heap corruption and double free()'s. All attacks were caught by Minos.

Argos

Argos[141] is a containment environment for worms and manual system compromises. It is actually an extended version of the Qemu emulator that tracks whether data coming from the network is used as jump targets, function addresses or instructions. To identify such activities, Argos performs dynamic taint analysis[134] (memory tainting). Memory tainting is the process where unsafe data that resides in the main memory or the registers are tagged. All data coming from the network is marked as tainted because by default they are considered as unsafe. Tainted data is tracked during execution. For example, if we have an `add` operation between a tainted register and an untainted one, the result of the addition will be tainted. Before data enters the Argos emulator, it is recorded in a network trace. As Argos has control of all operations that happen in the guest OS[3], it can detect whenever tainted data is tried to be executed or are used as jump targets, e.g. override function pointers. When tainted data are tried to be executed, an alarm is raised and the attack is logged. This log contains information about the attack and specifically registers, physical memory blocks and the network trace. This information is given as input to the signature generation component, which basically correlates information between the memory dump and the network trace using two approaches that will be described in more detail at Section 4.4. The signature generation time is linear to the size of the network trace and generated signatures did not produce false positives for DEFCON[6] and home-grown traces (traces collected at the site of authors).

The basic advantage of Argos is that it is able to detect without false positives that an automated attack is taking place, regardless of the application under attack or the attack's level of polymorphism. The major drawback of the Argos approach is the performance overhead. An application running in the Argos environment is 20 to 30 times slower than running in its native environment. A large part of this overhead is due to the underlying Qemu[72] emulation. The rest of the overhead is due to memory tainting and tracking of tainted data. However, as honeypots receive significantly less traffic than production systems, their overhead may be acceptable for some cases.

Practical Taint-Based Protection using Demand Emulation

The concept of building a honeypot that is robust against being compromised using memory tainting is also presented in [103]. Unlike other tainting-based approaches, like Argos, that run exclusively inside an emulated environment,the technique of this work switches between virtualized and emulated execution. The reason for this switching is primarily performance. The proposed approach is based on the Xen[70] virtual machine monitor that runs a protected operating system within a virtual machine. Xen provides a virtualized environment on top of which operating systems can run with very high performance. The containment of the

[3]In virtual machine terminology, guest OS is the operating system running inside the virtual machine while host OS is the operating system that runs the virtual machine software

approach is done as follows. When the processor accesses tainted data, Xen switches the virtual machine from the virtual CPU to an emulated processor. The emulated processor runs as user-space application in a separate control virtual machine, referred as ControlVM. The emulator tracks the propagation of the tainted data throughout the system and when no tainted data is accessed, Xen switches the virtual machine back to virtualized execution. The tracking of tainted data is extended to handle disk operations and not only memory and registers. As the proposed approach runs in an emulated environment only when needed, the performance of the approach makes it very attractive. While a fully emulated system that performs tainting runs 88 to 170 times slower, the proposed approach runs 1.2 to 3.7 times slower. The performance benchmark was performed using the LMbench suite[22], which measures time for well-known system calls.

HoneySpot

HoneySpot[18] is an architecture designed by the Spanish Honeynet Projet (SHP) that aims at monitoring the attacker's activities in wireless networks. The paper describes several approaches when designing a wireless honeypot. Basically, the design depends on the goals of these kind of experiments. Is it for monitoring dedicated wireless attacks, or, is it for monitoring attacks where wireless technologies are used as a transport protocol for these attacks? This has tremendous impacts on the architecture design. The proposed architecture is composed of several modules. The Wireless Access Point module that will provide the attacker with wireless connectivity. The Wireless Client module that will simulate client activity on the wireless honeypot fooling the wireless hacker and giving enough information in order to launch wireless attacks (such as guessing a WEP key). The Wireless Monitor module that will collect wireless traffic listening for wireless attacks (like a wireless intrusion detection system). The Wireless Data Analysis module that will take care of all reported data by the Wireless Monitor module. Finally, the Wired Infrastructure module that will simulate a real-world wired network with Internet access or emulated network and services. This wireless honeypot architecture is by far the most advanced in terms of features.

2.3.4 Client-side honeypots

Recently, we have observed exploits that target client applications and especially web browsers. The WMF and JPEG vulnerabilities ([24],[23]) have shown how the Internet Explorer browser can be compromised and execute arbitrary code on the victim's side. Instead of waiting passively for the attackers to contact them, as we have seen so far, client-side honeypots try to spot locations where malicious content is hosted. Although they are not passive systems, they are still characterized as honeypots as they are non-production systems. Client-side honeypots try to cover the gap of classic detection techniques. According to [45], only 1.5% of IDS signatures are based on client-side attacks, although the number of client-based vulnerabilities increases over time. In this Section strider honeymonkeys and honeyclient, the most popular client-side honeypots, are presented.

Strider HoneyMonkeys

Strider Honeymonkeys system[178] uses *monkey* programs that attempt to mimic human web browsing. A monkey program runs within virtual machines with OS's of various patch levels. The exploit detection system of honeymonkeys has three stages. In the first stage, each

honeymonkey visits N URLs simultaneously within an unpatched virtual machine. If an exploit is detected then the one-URL-per-VM mode is enabled. In that mode, each one of the N suspects runs in a separate virtual machine in order to determine which ones are exploit URLs. In Stage 2, HoneyMonkeys scan detected exploit-URLs and perform recursive redirection analysis to identify all web pages involved in exploit activities and to determine their relationships. In Stage 3, HoneyMonkeys continuously scan Stage-2 detected exploit-URLs using (nearly) fully patched VMs in order to detect attacks exploiting the latest vulnerabilities. The exploit detection is based on a non-signature approach. Each URL is visited in a separate browser instance. Then, honeymonkey waits for a few minutes to allow downloading of any code. Since the honeymonkey is not instructed to click on any dialog box to permit software installation, any executable files or registry entries created outside the browser sandbox indicate an exploit. This approach allows the detection of 0-day exploits. During the first month of the honeymonkey deployment, 752 unique URLs were hosting malicious pages, while a malicious web-site was performing zero-day exploits.

HoneyClient

HoneyClient[14] is a honeypot system designed to proactively detect client-side exploits. It runs on Windows platforms and drives Internet Explorer through two Perl scripts. The first script acts as a proxy, while the seconds performs integrity checking. Internet Explorer is set to have as proxy the script. After each crawl is finished, files and registry are checked. In case a change is detected, file and registry key value changes as well registry key additions or/and deletions are logged. The idea of HoneyClient is not restricted to Web crawling. It can be extended to other protocols, like P2P, IRC and instant messaging clients.

Capture

Capture[5] is a high-interaction client honeypot that tries to find malicious servers on the network following a similar approach with HoneyMonkeys. It consists of two main components, Capture Server and Capture Client. The primary purpose of the Capture Server is to control numerous Capture clients to interact with web servers. It allows to start and stop clients, instruct clients to interact with a web server retrieving a specified URI, and aggregating the classifications of the Capture clients regards the web server they have interacted with. The server provides this functionality in a scripting fashion. The URIs are distributed to the client in a round robin fashion.

Capture clients accept the commands of the server so as to know when to start and stop themselves and which URI to visit. A Capture client runs inside a virtual machine and interacts with a web server, monitoring its state for changes on the file system, registry, and processes that are running. Clients Since some events occur during normal operation (e.g. writing files to the web browser cache), exclusion lists allow to ignore certain type of events. If changes are detected that are not part of the exclusion list, the client makes a malicious classification of the web server and sends this information to the Capture server. Since the state of the Capture client has been changed, the Capture client resets its state to a clean state before it retrieves new instructions from the Capture server. In case no state changes are detected, the Capture client retrieves new instructions from the Capture server without resetting its state. Capture allows to automatically collect network traces and downloaded files malwarwhen a malicious server is encountered.

Shelia

Shelia[39] is a high-interaction client-side honeypot that tries to locate malware sources by inspecting the mail of the user and more specifically her spam messages. Shelia is implemented for Windows platforms and currently works with the Outlook Express mailer. Spam messages can contain all kind of information. They can include various types of attachments, URLs for malicious websites or even HTML code that can lead to infection.

The architecture of Shelia includes two main components. The first one is the client emulator which is responsible for identifying all the attachments and URLs received by email and to invoke the proper application to handle these attachments/URLs. The second one is the attack detector. The attack detector has a process monitor engine that monitors the client's actions and applies the containment strategy if required. The detector also determines if an action taken by a client is illegal. All attack information is logged by the attack log engine. Shelia monitors the processes and generates alerts when the process attempts to execute an invalid operation (i.e., execute a call to change the registry, create files, or attempt specific network operations) from a memory area that is not supposed to be executable code. The process monitoring is performed by hooking Win32 API calls, such as CreateProcessA, CreateThread, LoadLibraryA etc. The containment strategy followed by Shelia may allow the attack to run until it downloads the malware, which is then captured and stored in a specific directory. However, the downloaded malware is never executed.

2.3.5 Research papers

This Section is an overview of research papers that are based on honeypot technologies and infrastructures.

ScriptGen

By default, service emulation in honeyd is done by shell scripts, written usually in Perl, that ship with default distribution. The original purpose of honeyd is not to provide accurate scripts (in fact its scrips are just a proof of concept) but rather the framework to hook advanced scripts. Towards this direction, ScriptGen[121] is an effort to build an automated script generation tool. The input for this tool is a tcpdump trace, from which sequence of messages is extracted. A message is defined as the application-level content of a packet. Authors focus on TCP-based protocols only. These messages are used to build a state machine. As the size of the state machine can become very large, as a next step this state machine is simplified using the PI algorithm, an algorithm introduced in the Protocol Informatics Project[41], and an algorithm proposed by authors, the Region Analysis algorithm. Finally, from the simplified state machine a honeyd script is generated. PI is an algorithm that performs multiple alignments on a set of protocol samples and is able to identify the major classes of messages. The result of the PI alignment is given as input to the Region Analysis algorithm. The aligned sequences produced by PI are examined in a byte per byte basis and for each byte its type (ASCII or binary), value, variability of its values and the presence of gaps in that byte are computed. A region is then defined as a sequence of bytes that have the same type, have similar variability and may, or may not, have gaps. The quality of the results of ScriptGen is limited by the number of samples we use in the state machine generation. For example, if we use two samples, "GET /file1.html" and "GET /file2.html", ScriptGen would generate a state machine that is triggered by the "GET /file?.html" regular expression.

This is wrong as any other request will not be handled by the generated state machine. Another effort for semi-automatically creating emulators of network server applications is Honeybee[13]. The application consists of two parts: A scanner and a generic emulators per protocol. The Honeybee scanner talks to a real server and extracts its personality. These personalities are stored in database files and are used to control the generic emulator. The generic emulators use Honeyd's interface for Python plug-ins.

A Pointillist Approach for Comparing Honeypots

Pouget and Holz in [78] used the honeypot architecture of Leurre.com platforms to compare low- and high-interaction systems. The low-interaction honeypot setup is the same as this of a Leurre.com platform. The high-interaction honeypot setup was the same as the low-interaction one but instead of using modified version of honeyd, a VMware client was used. All virtual guests had adjacent IP addresses and they were monitored for 10 continuous weeks (August to October 2004). Both setups observed around seven thousands attacks. Attacks were classified into three types: first type (Type I) contains attacks that target a single host, Type II includes attacks that target 2 hosts and finally Type III includes the attacks that targeted all three hosts of the setup. Most of the attacks (around 67%) are Type I. However, for these attacks, high-interaction setup received 40 times more packets and this is due to the fact that they target talkative services. As low-interaction honeypots do not emulate service in depth, they receive considerably less packets. Around 4% of the attacks, targeted only 2 out of 3 systems of each setup. However, the majority of these attacks are incomplete Type III attacks. Concerning Type III attacks, an interesting fact is that all IP sources were observed on both environments. Authors reached three main conclusions. First, it is not necessary to deploy honeypots using hundreds of public IP addresses in order to identify scan activities against large block IPs. Second, low-interaction honeypots bring as much information as high interaction ones when it comes down to global statistics on the attacks. Finally, both interaction levels are required to build an efficient network of distributed honeypots, using high-interaction as a means to configure low-interaction ones.

Configuration of Dark Addresses

Honeypot networks are usually setup based on manual and ad-hoc configurations. However, such configurations do not lead to good visibility and are vulnerable to discovery. A typical example is when the production network consists of Linux workstations whose SSH service is attacked, while honeypots of the same domain emulate services like Microsoft SQL server. Taking into account the huge diversity in network and host configurations and the fact that vulnerabilities increase year by year, the task of configuring honeypots becomes even harder. Sinha et al. in [155] address this problem and propose an automated technique to generate honeynet configurations. Authors examined four configurations. First one was the one used by the Internet Motion Sensor project, where incoming connections are accepted but no application-level response is provided (All TCP Responder). The second one was the default configuration of Honeyd (Generic Honeyd) and the third one was random combinations of chosen OSes and services created by RandomNet[156] (Random Honeyd). The last configuration was the default one of Nepenthes system. The configurations were compared against the top 10 open TCP ports of an academic /16 network. All configurations failed to capture attacks on several ports that were attacked on the production environment and at the same

time made certain ports, like TCP port 139 (Microsoft RPC service), appear more important than others. Such behavior does not only affect the visibility of honeypots but also allows attacker to identify them.

Authors used a simple methodology to show that the configurations mentioned before expose honeypots to detection. They constructed a set of tests to make profiles for each host. These tests included checking ambiguities in implementation of TCP software (*tcp* test), identification of a web server and its configuration (*http* test) and finally a scan to TCP ports 1 to 1024 to discover open ones (*ports* test). Hosts are grouped by test values and then subnet is compared against the network as a whole for the values of the test. Then values of the test are aggregated and values of next test are compared. This process iterates over the determined test order until the last test. Tests are ordered by increasing entropy. To identify anomalies in a test, authors use z-statistics. The subnets with the highest anomaly score ideally should be the honeynets. Their methodology was evaluated by embedding different honeynet configurations into six networks, two of which were academic network and the rest four were web-server farms. The honeynet configurations were created using all four approaches (generic honeyd, random honeyd, all TCP responder and Nepenthes). Results yielded that it is easy to discover honeynets created by all four configurations, although someone would expect that configurations like Generic Honeyd and Nepenthes would be more resilient to identification.

The automated creation of configurations is a challenging task due to two main reasons. The first one is that there is a variation in the threat landscape. Ports that are frequently attacked in one month, may be rarely contacted the next month and vice versa. Authors confirm this variation by observing data collected over a 5-month period. The second one is the heterogeneity in operating systems and services. By examining 5512 hosts, 352 unique TCP stack implementation were found, 1237 hosts were running 241 different version of web servers. The proposed technique is based on profiling the existing network and generating configuration according to the extracted profiles. The profiles proportionally represent the network and are individually consistent. Individually consistency guarantees that a honeypot will not have an unrealistic profile and that it will match a live host. To achieve proportionality, real hosts are sampled using stratified sampling. Stratified sampling involves separating the population and allocating the size to be sampled proportionally at each point in the hierarchy. Additionally, weights in the values of some tests are supported. That means that if a port is preferred open than closed, a host with that port open will replace a host with that port closed during the profile extraction. Honeynets with configurations created by that process were embedded in the same five networks mentioned before. Results show that honeynets provide better visibility into vulnerable population. The distribution of services and operating systems is close to the vulnerable population. Furthermore, honeynet configurations are more resistant to fingerprinting when configured to use the authors' approach.

Data Reduction for the Scalable Automated Analysis of Distributed Darknet Traffic

M. Bailey et al. in [67] examine the properties of darknets in order to determine the effectiveness of building hybrid honeypot systems, e.g. systems that combine monitoring of unused address space with honeypot farms. Their goal is to analyze darknet traffic and filter out redundant traffic that will impose unnecessary overhead to honeypot farms. They try to identify threat characteristics that will enable them to limit the number of connections that

reach honeypots. Their measurements and analysis was done over 60 darknets in 30 organizations, a monitored space of 17 million addresses. The dark address space was monitored using the Internet Motion Sensor architecture[66]. The 14 IMS sensors monitored networks with variable size, ranging from /25 up to /17, over a period of 10 days (mid-August 2004). Initially, the source IP addresses were evaluated. The observed distribution showed that 90% of the packets were sent from less than 10% of source IP addresses. Secondly, the destination port distribution was examined. 90% of the packets target less than 1% of the destination ports. However, as the cross product of unique source IP addresses and total destination port is large, the top 10% source IP addresses were evaluated separately in terms of the destination port they contact and unique payloads they send. Over 55% of these IP addresses contacted a single destination port, 90% contacted less than six while 99% contacted less than ten. In terms of the paylod they send, 30% of addresses send only one payload, 70% send two or less and 92% of addresses send less than ten payloads.

The authors evaluated three methods of source-based filtering previously reported in [136]: source-connection, source-port and source-payload. In source-connection filtering, N connections from a single source are recorded, and all subsequent traffic from that source is ignored. In source-port filtering, N connections are maintained for every source and destination port pair. In source-payload filtering, the first N payloads sent by a source are used to define all future behavior by that source. The average reduction due to these three methods for N=1 was ranging from 86% up to 97%, but there were huge variations over time. In some time periods the reduction was dropped down to 47% and the main reason behind that behavior is that there is little overlap across subnets. Any additional darknet block added for monitoring will bring its own sensor-specific events. Based on that two observations, authors constructed a new filtering mechanism. The author's approach examines the distribution of unique IP addresses to a specific port.

Every hour, each darknet is queried for the number of unique source IP addresses that have contacted it with destination port x. For each hour, the number of unique source IP addresses contacting destination port x at each darknet is added up. This data is scanned over, one hour at a time, comparing the average (per hour) over the event window (last event window hours) to the average over the history window (last event window product the history factor) hours. If the ratio of event window average to history average is greater than the event threshold, an event is generated. These events are then filtered based on whether they are global or local, via the coverage threshold. The coverage threshold defines the number of darknets that would have generated an event individually for a threat. Their alert mechanism was able to identify such as a MYSQL worm, WINS and Veritas Backup Agent vulnerabilities, along with most major events that took place during the 4-month deployment phase of the technique.

Detectability of honeypots

Detectability of honeypots is an interesting research area. Nevertheless, we can only find a limited number of works in this area. In the article published in [2], three techniques were introduced to detect three subsystems of honeypots. The first one detects Sebek, the kernel module used in honeynets for logging host activities. Sebek hides itself from the kernel module list and this is done by a module called *cleaner*. After cleaner runs, the lsmod command does not provide any visual output that a module is running. However, information still remains in /dev/kmem and a simple grep of "insmod_sebek_S.data" reveals the existence of Sebek. The second technique introduces a way to detect honeyd. The idea is based on the fact that

by default honeyd resets the connections it cannot handle. Even when the SYN packet is malformed, the honeyd will respond with reset. A random scanning tool was used to identify honeyd systems by looking at the resets at random ports. The last technique identifies if an operating system runs inside VMware. As VMware is commonly used in the setup of honeypots, malware may use the technique to detect it runs on a honeypot or to prevent itself from being executed inside a virtual guest. The idea is very simple; the device names of virtual guest OS reveal the existence of VMware : "Model Number: VMware Virtual IDE CDROM Drive". Although this can be fixed, scanning of the devices reveals even more information as VMware does not return all necessary information.

Detection techniques against Honeynets, and especially Sebek, are also described in [93]. The first technique checks if network is congested due to Sebek. As Sebek client sends a packet to Sebek server for each read() operation, if one can call read() thousands times per second, this leads to network congestion. The detection is as follows. A local host is ping()'ed. Local host can be reached within a millisecond and this time is recorded. After the first ping, a second one is sent but now a *dd* operation runs in the background. According to measurements, the response time changed from 0.7 milliseconds to nearly 5 seconds. The second proposed approach is to check the system call table and observe the difference between the addresses of sys_read and sys_write. As sys_write is successor of sys_read in the system call table, these two functions originally have a few kilobytes distance. When Sebek overwrites the sys_read pointer, this distance increases to tens of megabytes. Authors also propose a technique to detect the presence of honeywall. As honeywall by default limits the number of connections to 15 per day, it is trivial to detect if the 16th connection is dropped or not, provided that connections target a known destination host. Additionally, an attacker can send packets that match snort inline signature (snort inline runs in the honeywall) to a machine it controls and then check if this packet was rewritten according to the matching signature.

Chapter 3

Misusing web browsers as an attack platform

In the last few years researchers have observed two significant changes in malicious activity on the Internet [150, 181, 165]. The first is the shift from amateur proof-of-concept attacks to professional profit-driven criminal activity. The second is the increasing sophistication of the attacks. Although significant efforts are made towards addressing the underlying vulnerabilities, it is very likely that attackers will try to adapt to *any* security response, by discovering new ways of exploiting systems to their advantage [132]. In this arms race, it is important for security researchers to proactively explore and mitigate new threats before they materialize.

This Chapter discusses one such threat, for which we have coined the term *puppetnets*. Puppetnets rely on websites that *coerce* web browsers to (unknowingly) participate in malicious activities. Such activities include distributed denial-of-service, worm propagation and reconnaissance probing, and can be engineered to be carried out in stealth, without any observable impact on an otherwise innocent-looking website. Puppetnets exploit the high degree of flexibility granted to the mechanisms comprising the web architecture, such as HTML and JavaScript. In particular, these mechanisms impose few restrictions on how remote hosts are accessed. A website under the control of an attacker can thereby transform a collection of web browsers into an *impromptu* distributed system that is effectively controlled by the attacker. Puppetnets expose a deeper problem in the design of the web. The problem is that the security model is focused almost exclusively on protecting browsers and their host environment from malicious web servers, as well as servers from malicious browsers. As a result, the model ignores the potential of attacks against third parties.

Websites controlling puppetnets could be either legitimate sites that have been subverted by attackers, malicious "underground" websites that can lure unsuspected users by providing interesting services (such as free web storage, illegal downloads, etc.), or websites that openly invite users to participate in vigilante campaigns. We must note however that puppetnet attacks are different from previous vigilante campaigns against spam and phishing sites that we are aware of. For instance, the Lycos "Make Love Not Spam" campaign [167] required users to install a screen-saver in order to attack known spam sites. Although similar campaigns can be orchestrated using puppetnets, in puppetnets users may not be aware of their participation, or may be coerced to do so; the attack can be launched stealthily from an innocent-looking web page, without requiring any extra software to be installed, or any other kind of user action.

Puppetnets differ from botnets in three fundamental ways. First, puppetnets are not heavily dependent on the exploitation of specific implementation flaws, or on social engineering tactics that trick users into installing malicious software on their computer. They exploit *architectural* features that serve purposes such as enabling dynamic content, load distribution and cooperation between content providers. At the same time, they rely on the *amplification* of vulnerabilities that seem insignificant from the perspective of a single browser, but can cause significant damage when abused by a popular website. Thus, it seems harder to eliminate such a threat in similar terms to common implementation flaws, especially if this would require sacrificing functionality that is of great value to web designers. Additionally, even if we optimistically assume that major security problems such as code injection and traditional botnets are successfully countered, some puppetnet attacks will still be possible. Furthermore, the nature of the problem implies that the attack vector is pervasive: puppetnets can instruct *any* web browser to engage in malicious activities.

Second, the attacker does not have complete control over the actions of the participating nodes. Instead, actions have to be composed using the primitives offered from within the browser sandbox – hence the analogy to puppets. Although the flexibility of puppetnets seems limited when compared to botnets, we will show that they are surprisingly powerful.

Finally, participation in puppetnets is dynamic, making them a moving target, since users join and participate unknowingly while surfing the net. Thus, it seems easy for the attackers to maintain a reasonable population, without the burden of having to look for new victims. At the same time, it is harder for the defenders to track and filter out attacks, as puppets are likely to be relatively short-lived.

A fundamental property of puppetnet attacks, in contrast to most web attacks that directly harm the browser's host machine, is that they only *indirectly* misuse browsers to attack third parties. As such, users are less likely to be vigilant, less likely to notice the attacks, and have lesser incentive to address the problem. Similar problems arise at the server side: if puppetnet code is installed on a website, the site may continue to operate without any adverse consequences or signs of compromise (in contrast to defacement and other similar attacks), making it less likely that administrators will react in a timely fashion.

In this Chapter we experimentally assess the threat from puppetnets. We discuss the building blocks for engineering denial-of-service attacks, worm propagation and other puppetnet attacks, and attempt to quantify how puppetnets would perform. Finally, we examine various options for guarding against such attacks.

3.1 Puppetnets: design and analysis

We attempt to map out the attackers' opportunity space for misusing Web browsers. In lieu of the necessary formal tools for analyzing potential vulnerabilities, neither the types of attacks nor their specific realizations are exhaustive enough to provide us with a solid worst-case scenario. Nevertheless, we have tried to enhance the attacks as much as possible, in an attempt to approximately determine in what ways and to what effect the attacker could capitalize on the underlying vulnerabilities. In the rest of this section, we explore in more detail a number of ways of using puppetnets, and attempt to quantify their effectiveness.

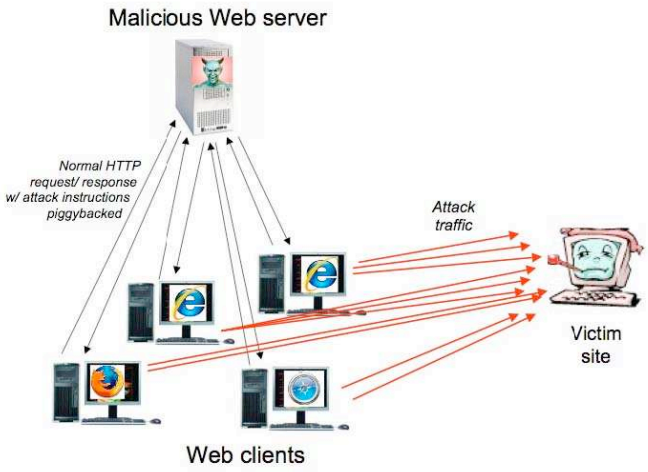


Figure 3.1: **DDoS using puppetnets**

3.1.1 Distributed Denial of Service

The flexibility of Web architecture provides many ways for launching DoS attacks using puppetnets. The common component of the attack in all of its forms is an instruction that asks the remote browser to access some object from the victim. There are several ways of embedding such instructions in an otherwise legitimate webpage. The simplest way is to add an image reference, as commonly used in the vast majority of web pages. Other ways include opening up pop-up windows, creating new frames that load a remote object, and loading image objects through JavaScript. We are not aware of any browser that imposes restrictions on the location or type of the target referenced through these mechanisms.

We assume that the intention of the attacker is to maximize the effectiveness of the DDoS attack, at the lowest possible cost, and as stealthily as possible. An attack may have different objectives: maximize the amount of ingress traffic to the victim, the egress traffic from the victim, connection state, *etc.* Here we focus on raw bandwidth attacks in both directions, but emphasize on ingress traffic as it seems harder to defend against: the host has full control over egress traffic, but usually limited control over ingress traffic.

To create a large number of requests to the target site, the attacker can embed a sequence of image references in the malicious web page. This can be done using either a sequence of `IMG SRC` instructions, or a JavaScript loop that instructs the browser to load objects from the target server. In the latter case, the attack seems to be much more efficient in terms of *attack gain*, e.g., the effort (in terms of bandwidth) that the attacker has to spend for generating a given amount of attack traffic. This assumes that the attacker either targets the same URL in all requests, or is able to construct valid target URLs through JavaScript without wasting space for every URL. To prevent client-side caching of requests, the attacker can also attach an invariant modifier string to the attack URL that is ignored by the server [1] but considered by the client in the context of deciding whether the object is already cached.

Another constraint is that most browsers impose a limit on the number of simultaneous

[1]The URL specification [74] states that URLs have the form `http://host:port/path?searchpart`. The `searchpart` is ignored by web servers such as Apache if included in a normal file URL.

connections to the same server. For MSIE and Firefox the limit is two connections. However, we can circumvent this limit using aliases of the same server, such as using the DNS name instead of the IP address, stripping the "www" part from or adding a trailing dot to the host name, etc. Most browsers generally treat these as different servers. Servers that host more than one website (i.e., "virtual hosting") are especially vulnerable to this form of amplification.

To make the attack stealthy in terms of not getting noticed by the user, the attacker can employ hidden (*e.g.,* zero-size) frames to launch the attack-bearing page in the background. To maximize effectiveness, the requests should not be rendered within a frame and should not interfere with normal page loading. To achieve this, the attacker can employ the same technique used by Web designers for pre-loading images for future display on a web page. The process of requesting target URLs can then be repeated through a loop or in an event-driven fashion. The loop is likely to be more expensive and may interfere with normal browser activity as it may not relinquish control frequently enough for the browser to be used for other purposes. The event-driven approach, using JavaScript timeouts, appears more attractive

Analysis of DDoS attacks

We explore the effectiveness of puppetnets as a DDoS infrastructure. The "firepower" of a DDoS attack will be equal to the number of users concurrently viewing the malicious page on their web browser (henceforth referred to as *site viewers*) multiplied by the amount of bandwidth each of these users can generate towards the target server. Considering that some web servers are visited by millions of users every day, the scale of the potential threat becomes evident. We consider the *size* of a puppetnet to be equal to the site viewers for the set of web servers controlled by the same attacker. Although it is tempting to use puppetnet size for a direct comparison to botnets, a threat analysis based on such a comparison alone may be misleading. Firstly, a bot is generally more powerful than a puppet, as it has full control over the host, in contrast to a puppet that is somewhat constrained within the browser sandbox. Secondly, a recent study [86] observes a trend towards smaller botnets, suggesting that such botnets may be more attractive, as they are easier to manage and keep undetected, yet powerful enough for the attacker to pursue his objectives. Finally, an attacker may construct a hybrid, two-level system, with a small botnet consisting of a number of web servers, each controlling a number of puppets.

To estimate the firepower of puppetnets we could rely on direct measurements of site viewers for a large fraction of the websites in the Internet. Although this would be ideal in terms of accuracy, to the best of our knowledge there is no published study that provides such data. Furthermore, carrying out such a large-scale study seems like a daunting task. We therefore obtain a rough estimate using "second-hand" information from website reports and Web statistics organizations. There are several sources providing data on the number of daily or monthly visitors:

- Many sites use tools such as Webalizer [71] and WebTrends [104] to generate usage statistics in a standard format. This makes them easy to locate through search engines and to automatically post-process. We have obtained WebTrends reports for 249 sites and Webalizer reports for 738 sites, covering a one-month period in December 2005. Although these sites may not be entirely random, as the sampling may be influenced by the search engine, we found that most of them are non-commercial sites with little content and very few visits.

- Some Web audit companies such as ABC Electronic provide public databases for a fairly large number of their customers [56]. These sites include well-known and relatively popular sites. We obtained 138 samples from this source.

- Alexa [59] tracks the access patterns of several million users through a browser-side toolbar and provides, among other things, statistics on the top-500 most popular sites. Although Alexa statistics have been criticized as inaccurate because of the relatively small sample size [63], this problem applies mostly to less popular sites and not the top-500. Very few of these sites are tracked by ABC Electronic.[2]

We have combined these numbers to get an estimate on the number of visitors per day for different websites. Relating the number of visitors per day to the number of site viewers is relatively straightforward. Recall that Little's law [124] states that if λ is the arrival rate of clients in a queuing system and T the time spent in the system, then the number of customers active in the system N is $N = \lambda T$.

To obtain the number of site viewers we also need, for each visit to a website, the time spent by users viewing pages on that site. None of the sources of website popularity statistics mentioned above provide such statistics. We therefore have to obtain a separate estimate of web browsing times, making the assumption that site popularity and browsing times are not correlated in a way that could significantly distort our rough estimates.

Note that although we base our analysis on estimates of "typical" website viewing patterns, the attacker may also employ methods for increasing viewing times, such as incentivizing users (*e.g.,* asking users to keep a pop-up window open for the download to proceed), slowing down the download of legitimate pages, and providing more interesting content in the case of a malicious website.

Web session time measurements based entirely on server log files may not accurately reflect the time spent by users viewing pages on a particular website. These measurements compute the session time as the difference between the last and first request to the website by a particular user, and often include a timeout threshold between requests to distinguish between different users. The remote server usually cannot tell whether a user is *actively* viewing a page or whether he has closed the browser window or moved to a different site. As we have informally observed, many users leave several browser instances open for long periods of time, we were concerned that web session measurements may not be reliable enough by themselves for the purposes of this study. We thus considered the following three data sources for our estimates:

- We obtain *real* browsing times through a small-scale experiment: we developed browser extensions for both MSIE and Firefox that keep track of web page viewing times and regularly post anonymized usage reports back to our server. The experiment involved roughly 20 users and resulted in a dataset of roughly 9,000 page viewing reports.

- We instrumented all pages on the server of our institution to include JavaScript code that makes a small request back to the server every 30 seconds. This allows us to infer

[2]Alexa only provides relative measures of daily visitors as a fraction of users that have the Alexa toolbar installed, and not absolute numbers of daily visitors. To obtain the absolute number of daily visitors we compare the numbers from Alexa to those from ABC Electronic, for those sites that appear on both datasets. This gives us a (crude) estimate of the Internet population, which we then use to translate visit counts from relative to absolute.

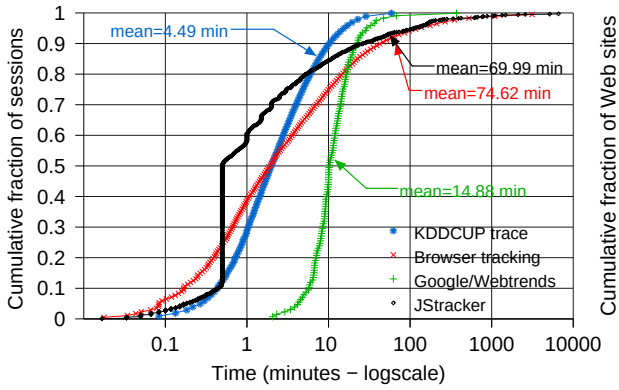


Figure 3.2: **Website viewing times**

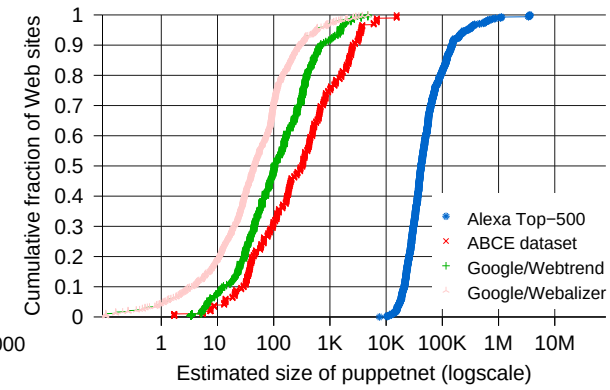


Figure 3.3: **Estimated size of puppetnets**

how long the browser points to one of the instrumented pages. We obtained data on more than 3,000 sessions over a period of two months starting January 2006. These results are likely to be optimistic, as the instrumented website is not particularly deep or content-heavy.

- We analyzed the KDD Cup 2000 dataset [114] which contains clickstream and purchase data from a defunct commercial website. The use of cookies, the size of the dataset, and the commercial nature of the measured website suggest that the data are reasonably representative for many websites.

- We obtained, through a search engine, WebTrends reports on web session times from 249 sites, similar to the popularity measurements, which provide us with mean session time estimates per site.

The distributions of estimated session times, as well as the means of the distributions, are shown in Figure 3.2[3]. As suspected, the high-end tail of the distribution for the more reliable browser-tracking measurements is substantially larger than that for other measurement methods. This confirms our informal observation that users tend to leave browser windows open for long periods of time, and our concern that logfile-based session time measurements may underestimate viewing times. The JavaScript tracker numbers also appear to confirm this observation. As in the case of DDoS, we are interested in the mean number of active viewers. Our results show that because of the high-end tails, the mean time that users keep pages on their browser is around 74 minutes, 6-13 times more than the session time as predicted using logfiles[4].

From the statistics on daily visits and typical page viewing times we estimate the size of a puppetnet. The results for the four groups of website popularity measurements are shown in Figure 3.3. The main observation here is that puppetnets appear to be comparable in size to botnets. Most top-500 sites appear highly attractive as targets for setting up puppetnets, with the top-100 sites able to form puppetnets controlling more than 100,000 browsers at any time. The sizes of the largest potential puppetnets (for the top-5 sites) seem comparable to

[3]We highlight the means as more informative in this case, as medians hide the high-end contributors to the aggregate page viewing time in a puppetnet.

[4]Note that the WebTrends distribution seems to have much lower variance and a much higher median than the other two sources. This is an artifact, as for WebTrends we only have access to a distribution of *means* for different sites, rather than the distribution of session times.

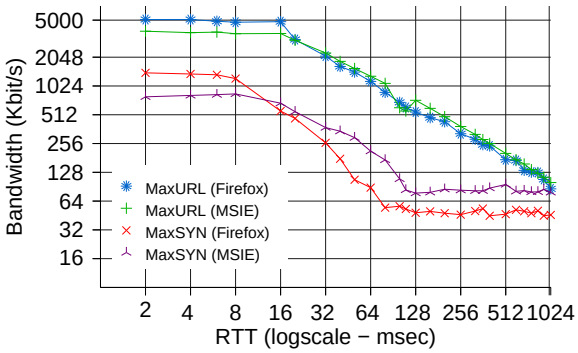


Figure 3.4: **Ingress bandwidth consumed by one puppet vs. RTT between browser and server with the attack code based on JavaScript**

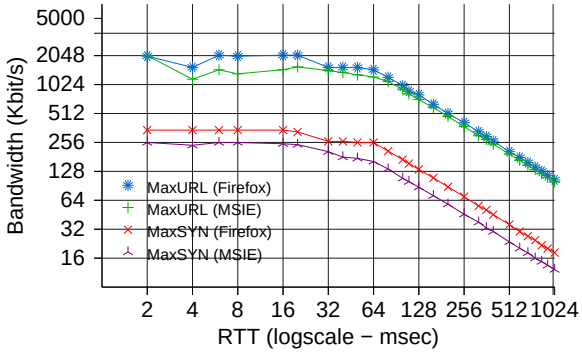


Figure 3.5: **Ingress bandwidth consumed by one puppet vs. RTT between browser and server with the attack code based on static HTML code**

	Firefox	**MSIE**
maxSYN 2 aliases	83.97	106.30
maxSYN 3 aliases	137.26	173.28
maxURL 2 aliases	664.74	502.06
maxURL 3 aliases	1053.79	648.33

Table 3.1: **Estimated bandwidth (Mbit/s) of ingress DDoS from 1000 puppets**

the largest botnets seen [110], at 1-2M puppets. Although one could argue that top sites are more likely to be secure, the figures for sites other than the top-500 are also worrying: More than 20% of typical commercial sites can be used for puppetnets of 10,000 nodes, while 4-10% of randomly selected sites can be popular enough for hosting puppetnets of more than 1,000 nodes.

As discussed previously, however, the key question is not how big a puppetnet is but whether the firepower is sufficient enough for typical DDoS scenarios. To estimate the DDoS firepower of puppetnets we first need to determine how much traffic a browser can typically generate under the attacker's command.

We experimentally measure the bandwidth generated by puppetized browsers, focusing initially only on ingress bandwidth, since it is harder to control. Early experiments with servers and browsers in different locations (not presented here in the interest of space) show that the main factor affecting DoS strength is the RTT between client and server. We therefore focus on precisely quantifying DoS strength in a controlled lab setting, with different line speeds and network delays emulated using *dummynet* [144], and an Apache webserver running on the victim host. We consider two types of attacks: a simple attack aiming to maximize SYN packets (maxSYN), and one aiming to maximize the ingress bandwidth consumed (maxURL). For the maxSYN attack, the sources of ten JavaScript image objects are set to be non-existent URLs repeatedly every 50 milliseconds. Every time the script writes into the image source URL variable, the browser stalls old connections are stalled and establishes new connections to fetch the newly-set image URL. For the maxURL attack we load a page with several thousand requests for non-existent URLs of 2048 bytes each (as MSIE can handle URLs of up to 2048 characters). The link between puppet and server was set to 10 Mbit/s in all experiments.

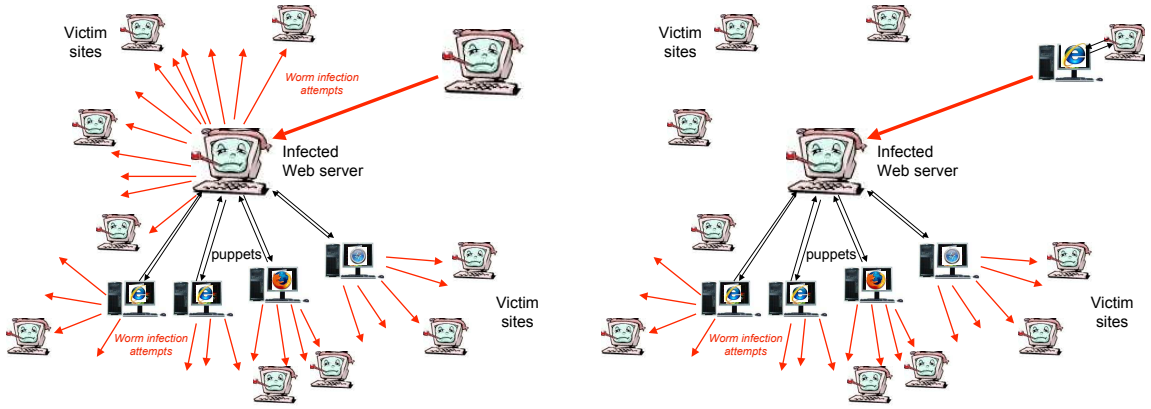
In Figure 3.4, the ingress bandwidth of the server is plotted against the RTT between the puppet and the server, for the case of 3 aliases. The effectiveness of the attack decreases for high RTTs, as requests spend more time "in-flight" and the connection limit to the same server is capped by the browser. For the maxSYN experiment, a puppet can generate up to 300 Kbit/s to 2 Mbit/s when close to the server, while for high RTTs around 250 msec the puppet can generate only around 60 Kbit/s. For the maxURL attack, these numbers become 3-5 Mbit/s and 200-500 Kbit/s respectively. The results seem to differ for both browsers: MSIE generates more attack traffic than Firefox for maxSYN, while Firefox generates more traffic for maxURL. We have not been able to determine the exact cause of the difference, mostly due to the lack of source code for MSIE. The same figures apply for slower connections, with RTTs remaining the dominant factor determining puppet DoS performance.

We also considered the case where a puppet has JavaScript disabled. We redesigned the attack page so that it only includes HTML code without any dynamic object or JavaScript code. The results are summarized in Figure 3.5. We notice a 30% reduction in the effectiveness of DDoS attack in the absence of JavaScript for maxURL, while for maxSYN the attack is up to 50% less potent.

Using the measurements of Figure 3.4, the distribution of RTTs measured in [157] and the capacity distribution from [149], we estimate the firepower of a 1000-node puppetnet, for different aliasing factors, as shown in Table 3.1. From these estimates we also see that around 1000 puppets are sufficient for consuming a full 155 Mbit/s link using SYN packets alone, and only around 150 puppets are needed for a maxURL attack on the same link. These estimates suggest that puppetnets can launch powerful DDoS attacks and should therefore be considered as a serious threat.

Considering the analysis above, we expect the following puppetnet scenarios to be more likely. An attacker controlling a popular web page can readily launch puppetnet attacks; many of the top-500 sites are highly suspect offering "warez" and other illegal downloads. Furthermore, we have found that some well-known underground sites, not listed in the top-500, can create puppetnets of 10,000-70,000 puppets (see [120]). Finally, the authors of reference [178] report that by scanning the most popular one million web pages according to a popular search engine, they found 470 malicious sites, many of which serve popular content related to celebrities, song lyrics, wallpapers, video game cheats, and wrestling. These malicious sites were found to be luring unsuspected users with the purpose of installing malware on their machines by exploiting client-side vulnerabilities. The compromised machines are often used to form a botnet, but visits to these popular sites could be used for staging a puppetnet attack instead.

Another way to stage a puppetnet attack is by compromising and injecting puppetnet code to a popular website. Although one might argue that popular sites are more likely to be secure, checking the top-500 sites from Alexa against the defacement statistics from zone-h[184] reveals that in the first four months of 2006 alone, 7 pages having the same domain as popular sites were defaced. For the entire year 2005 this number reaches 18. We must note, however, that the defaced pages were usually not front pages, and therefore their hits are likely to be less than those of the front pages. We also found many of them running old versions of Apache and IIS, although we did not go as far as running penetration tests on them to determine whether they were patched or not.



(a) client+server **(b) client-only**

Figure 3.6: **Two different ways that puppetnets could be used for worm propagation: (a) illustrates an infected server that uses puppets to propagate the worm, and (b) a server that propagates only through the puppet browsers.**

3.1.2 Worm propagation

Puppetnets can be used to spread worms that target vulnerable websites through URL-encoded exploits. Vulnerabilities in Web applications are an attractive target for puppetnets as these attacks can usually be encoded in a URL and embedded in a web page. Web applications such as blogs, wikis, and bulletin boards are now among the most common targets of malicious activity captured by honeynets. The most commonly targeted applications according to recent statistics [139] are Awstats, XMLRPC, PHPBB, Mambo, WebCalendar, and PostNuke.

A Web-based worm can enhance its propagation with puppetnets as follows. When a web server becomes infected, the worm adds puppetnet code to some or all of the web pages on the server. The appearance of the pages could remain intact, just like in our DDoS attack, and each unsuspected visitor accessing the infected site would automatically run the worm propagation code. In an analogy to real-world diseases, web servers are *hosts* of the worm while browsers are *carriers* which participate in the propagation process although they are not vulnerable themselves. Besides using browsers to increase the aggregate scanning rate, a worm could spread entirely through browsers. This could be particularly useful if behavioral blockers prevent servers from initiating outbound connections. Furthermore, puppetnets could help worms penetrate NAT and firewall boundaries, thereby extending the reach of the infection to networks that would otherwise be immune to the attack. For example, the infected web server could instruct puppets to try propagating on private addresses such as 192.168.x.y. The scenarios for worm propagation are shown in Figure 3.6.

Analysis of worm propagation

To understand the factors affecting puppetnet worm propagation we first utilize an analytical model, and then proceed to measure key parameters of puppetnet worms and use simulation to validate the model and explore the resulting parameter space.

Analytical model: We have developed an epidemiological model for worm propagating

using puppetnets. Briefly, we have extended classical homogeneous models to account for how clients and servers contribute to worm propagation in a puppetnet scenario. The key parameters of the model are the browser scanning rate, the puppetnet size, and the time spent by puppets on the worm-spreading web page.

Modeling puppetnet worm propagation

To investigate the propagation dynamics of a puppetnet worm, we adopt the classical Random Constant Spread (RCS) model, as used in other studies [163]. We use a notation system similar to [163]:

- N: vulnerable population
- a: fraction of vulnerable machines which have been compromised
- a_0: initial fraction of compromised machines
- t: time
- t_h: mean holding time/latency of users (e.g., the amount of time the browser is actively displaying the infected Web page)
- C: number of concurrent clients per server
- K_s: server's initial compromise rate
- K_c: client's initial compromise rate
- K_p: puppetnet's initial compromise rate
- K: overall initial compromise rate
- $\psi(t)$: distribution process of holding time on a web page

At time t, the number of new machines being compromised within dt is affected by two parameters:

- Due to server: $Na(t)K_s(1-a(t))dt$.
- Due to puppetnet: each server has C clients and every client compromises at a rate of $K_c(1-a)$. However, due to the time $\psi(t)$ between web page infection and the user browsing to another infected page on the server, only puppetnets controlled by servers that have become active at time t are capable of propagating worm.

$$Nda(t) = Na(t)K_s(1-a(t))dt + N\,[\int_0^t a(\tau)\psi(t-\tau)d\tau]\;CK_c(1-a(t))dt$$

where $a(t)$ and $\psi(t)$ express full notation of a and ψ as functions of time t. Note that $a(t) = 0$ and $\psi(t) = 0$ for $t < 0$.

$$\frac{da}{dt} = K_s a(1-a) + [\int_0^t a(\tau)\psi(t-\tau)d\tau]CK_c(1-a) \tag{3.1}$$

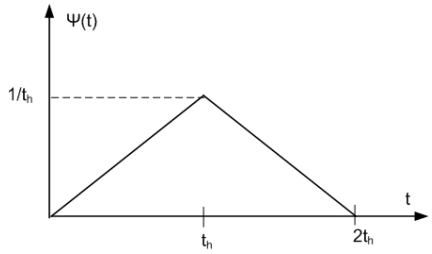


Figure 3.7: Triangular latency process of clients

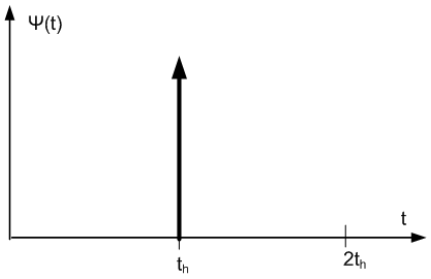


Figure 3.8: Impulse arrival of clients : universal holding time t_h

Equation 3.1 is generally non-linear without closed form solution and therefore could only be solved numerically.

We consider two cases of $\psi(t)$ as shown in Figure 3.7 and Figure 3.8. Figure 3.7 shows a model of a user's latency process at a web server. Clients slowly arrive at the web server at linear rate before and after a mean holding time t_h, until all users are either new users that joined after the server was infected, or old users that refreshed the page (and hence got the infected one) or browsed to another page on the same server. There is no closed-form solution for Equation 3.1 in this case.

Note that the area bounded by $\psi(t)$ and x-axis is unity: $\int_0^\infty \psi(\tau)d\tau = 0$

Figure 3.8 is a further simplification, with the holding time becoming a universal constant t_h. This means that all puppetnets controlled by servers that are infected at time $(t - t_h)$ become active at time t. With this assumption, $\psi(t)$ is an impulse function at $t = t_h$. Despite its simplicity, a symbolic approximate solution could be found, which gives us some intuition on the role of different parameters on the worm outbreak.

$[\int_0^t a(\tau)\psi(t-\tau)d\tau] = a(t - t_h) \approx a(t) - t_h\frac{da(t)}{dt}$ (first order approximation when t is sufficiently large).

Substituting in Equation 3.1, we get:

$$\frac{a^{1+K_p t_h}}{1-a} = e^{K(t-T)}$$

where $K_p = CK_c$ and $K = K_s + K_p$

- If t_h is negligible, the final solution form becomes:

$$a = \frac{e^{K(t-T)}}{1+e^{K(t-T)}}$$

  where T is a constant that specifies the time of the initial worm outbreak. The propagation pattern is exactly the same as a normal worm without puppetnets [163], except that the worm propagation rate is increased by the contribution of puppets: $K = K_s + K_p$. More importantly, this contribution is significant if $K_p >> K_s$.

- If $t_h > 0$, only a numerical solution can be obtained. Compared to the above case of zero holding time, the worm outbreak is delayed by:

$$\Delta t = \frac{K_p}{K} t_h ln \frac{a}{a_0}$$

In all cases, worm propagation with puppetnets obeys the logistic form: its graph has the same sigmoid shape as a typical random scanning worm. The contribution to the propagation process by clients and servers mainly depends on the constants K_s (sever-side) and K_c (client-side).

Finally, we examine puppetnet-like viral propagation characteristics for general computer viruses and worms. We incorporate puppetnet factor into two general epidemiological models: Susceptible-Infected-Susceptible (SIS) and Susceptible-Infected-Removed (SIR) [177].

In this case, δ is defined as the node cure rate (or virus death rate), i.e. the rate at which a node will be cured if it is infected. Note that our modified-RCS model is a special case of the SIS model with $\delta = 0$.

The modified SIS equation for puppetnet worms is:

$$\frac{da}{dt} = K_s a(1-a) + [\int_0^t a(\tau)e^{-\delta(t-\tau)}\psi(t-\tau)d\tau]CK_c(1-a) - \delta a \tag{3.2}$$

while the steady state solution is:

$$a(\infty)_{SIS} = 1 - \frac{\delta}{K_s + CK_c\int_0^\infty \frac{\psi(\tau)}{e^{\delta\tau}}d\tau}$$

Additionally, the modified SIR equation for puppetnet worms is:

$$\frac{da}{dt} = K_s a(1-a-\delta\int_0^t a(\tau)d\tau) + [\int_0^t a(\tau)e^{-\delta(t-\tau)}\psi(t-\tau)d\tau]CK_c(1-a-\delta\int_0^t a(\tau)d\tau) - \delta a \tag{3.3}$$

and the steady state solution for this case is:

$$a(\infty)_{SIR} = \frac{[K_s + CK_c\int_0^\infty \frac{\psi(\tau)}{e^{\delta\tau}}d\tau] - \delta}{[K_s + CK_c\int_0^\infty \frac{\psi(\tau)}{e^{\delta\tau}}d\tau](1+\delta.\infty)} = 0$$

Therefore, with the puppetnet enhancement, the final epidemic state for the SIR model is also zero.

Scanning performance: If the attacker relies on simple web requests, the scanning rate is constrained by the default browser connection timeout and limits imposed by the OS and the browser on the maximum number of outstanding connections. In our proof-of-concept attack, we have embedded a hidden HTML frame with image elements into a normal web page, with each image element pointing to a random IP address with a request for the attack URL. Note that the timeout for each round of infection attempts can be much lower than the time needed to infect all possible targets (based on RTTs). We assume that the redundancy of the worm will ensure that any potential miss from one source is likely to be within reach from another source.

Experimentally, we have found that both MSIE and Firefox on an XP SP2 platform can achieve maximum worm scanning rates of roughly 60 scans/min, mostly due to OS connection limiting. On other platforms, such as Linux, we found that a browser can perform roughly 600 scans/min without noticeable impact on regular activities of the user. These measurements were consistent across different hardware platforms and network connections.

Impact on worm propagation: We simulate a puppetnet worm outbreak and compare results with the prediction of our analytical model. We consider CodeRed [80] as an example of a worm targeting web servers and use its parameters for our experiments. To simplify

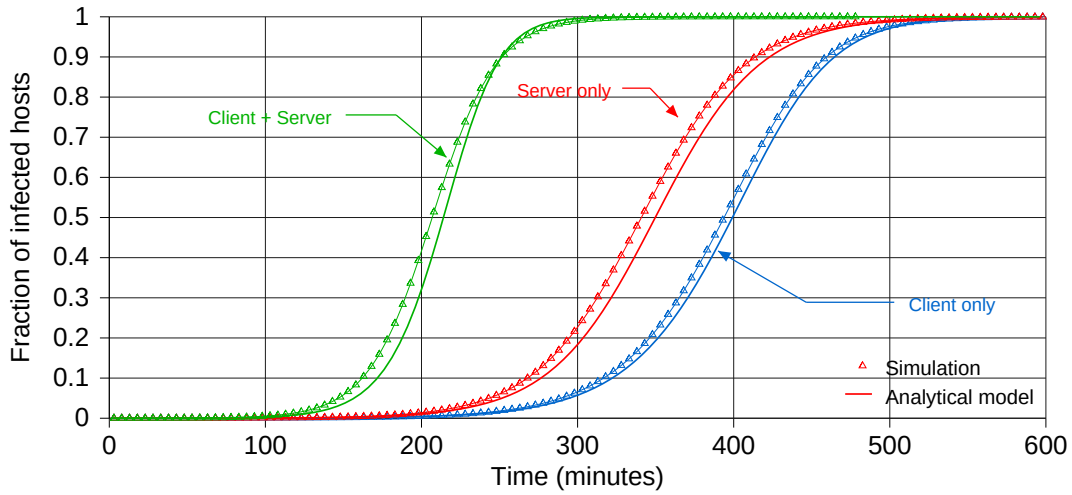


Figure 3.9: **Worm propagation with puppetnet**

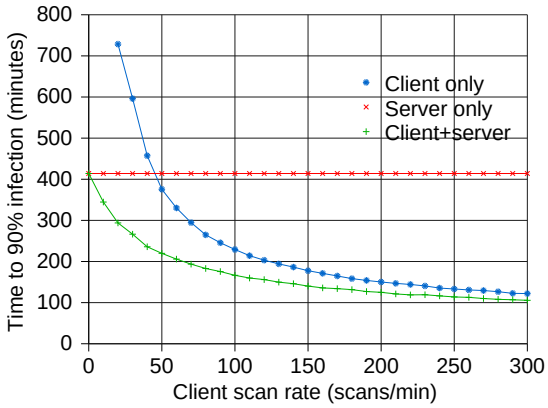


Figure 3.10: **Worm infection for different browser scan rates**

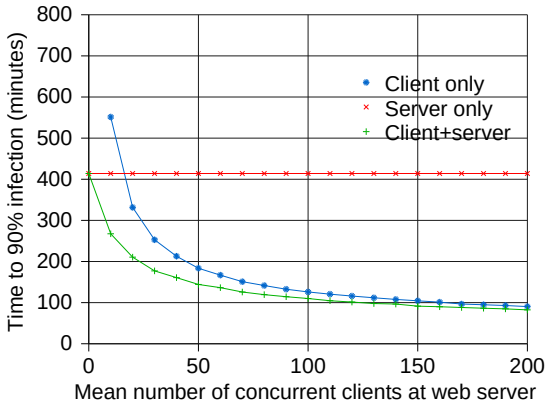


Figure 3.11: **Worm infection versus popularity of web servers**

the analysis, we ignore possible human intervention such as patching, quarantine, and the potential effect of congestion resulting from worm activity.

We examine three different scenarios: (a) a normal worm where only compromised servers can scan and infect other servers, (b) a puppetnet-enhanced worm where both the compromised servers and their browsers propagate the infection, and (c) a puppetnet-only worm where servers only push the worm solely through puppets to achieve stealth or bypass defenses.

We have extended a publicly available CodeRed simulator [185] to simulate puppetnet worms. We adopt the parameters of CodeRed as used in [185]: a vulnerable population of 360,000 and a server scanning rate of 358 scans/min. In the simulation, we directly use these parameters, while in our analytical model, we map these parameters to analytical model parameters and numerically solve the differential equations. Note that our model is a special case of the Kephart-White Susceptible-Infected-Susceptible (SIS) model [112] with no virus

curing. The compromise rate is $K = \beta \times \langle k \rangle$ where β is the virus birth rate defined on every directed edge from an infected node to its neighbors, and $\langle k \rangle$ is the average node out-degree. Assuming the Internet is a fully connected network, $\langle k \rangle_{CodeRed} = 360,000$ and $\beta_{CodeRed,server} = 358/2^{32}$, we have $K_s = 0.03$. Our simulation and analytical model also include the delay in accessing a web page as users have to click or reload a newly-infected web page to start participating in worm propagation.

We obtain our simulation results by taking the mean over five independent runs. For this experiment, we use typical parameters measured experimentally: browsers performing 36 scans/min (*i.e.,* an order of magnitude slower than servers), and web servers with about 13 concurrent users, and an average page holding time of 15 minutes. To study the effect of these parameters, we vary their values and estimate worm infection time as shown in Figures 3.10 and 3.11.

Figure 3.9 illustrates the progress of the infection over time for the three scenarios. In all cases the propagation process obeys the standard S-shaped logistic growth model. The simulated virus propagation matches reasonably well with the analytical model. Both agree on a worm propagation time of 50 minutes for holding times in the order of $t_h = 15min$ (that is, compared to the case of zero holding time). A client-only worm can perform as well as a normal worm, suggesting that puppetnets are quite effective at propagating worm epidemics.

Figure 3.10 illustrates the time needed to infect 90% of the vulnerable population for different browser scanning rates. When browsers scan at 45 scans/min, the client-only scenario is roughly equivalent to the server-only scenario. At the maximum scan rate of this experiment (which is far more than the scan rate for MSIE, but only a third of the scan rate for Linux), a puppetnet can infect 90% of the vulnerable population within 19 minutes. This is in line with Warhol worms and an order of magnitude faster than CodeRed.

Figure 3.11 confirms that the popularity of compromised servers plays an important role in worm performance. The break-even point between the server-only and client-only cases is when web servers have 16 concurrent clients on average. For large browser scanning rate or highly popular compromised servers, the client-only scenario converges to the client-server scenario. That means that infection attempts launched from browsers are so powerful that they dominate the infection process.

Finally, in a separate experiment we found that if the worm uses a small initial hitlist to specifically target busy web servers with more than 150 concurrent visitors, the infection time is reduced to less than two minutes, similar to flash worms [160].

3.1.3 Reconnaissance probes

We discuss how malicious or subverted websites can orchestrate distributed reconnaissance probes. Such probes can be useful for the attacker to locate potential targets before launching an actual attack. The attacker can thereby operate in stealth, rather than risk triggering detectors that look for aggressive opportunistic attacks. Furthermore, as in worm propagation, puppets can be used to scan behind firewalls, NATs and detection systems. Finally, probes may also enable attackers to build *hitlists* that have been shown to result in extremely fast-spreading worms [160].

As with DDoS, the attacker installs a web page on the website that contains a hidden HTML frame that performs all the attack-related activities. The security model of modern browsers imposes restrictions on how the attacker can set up probing. For instance, it is not possible to ask the browser to request an object from a remote server and then forward the

response back to the attacker's website. This is because of the so-called "same domain" (or "same origin") policy [148], which is designed to prevent actions such as stealing passwords and monitoring user activity. For the same reason, browsers refuse access to the contents of an inline frame, unless the source of the frame is in the same domain with the parent page.

Unfortunately, there are workarounds for the attacker to indirectly infer whether a connection to a remote host is successful. The basic idea is similar to the timing attack of [97]. We "sandwich" the probe request between two requests to the attacker's website.

We can infer whether the target is responding to a puppet by measuring the time difference between the first and third request. If the target does not respond, the difference will be either very small (*e.g.,* because of an ICMP UNREACHABLE message) or very close to the browser request timeout. If the target is responsive, then the difference will vary but is unlikely to coincide with the timeout.

Because browsers can launch multiple connections in parallel, the attacker needs to serialize the three requests. This can be done with additional requests to the attacker's website in order to consume all but one connection slots. However, this would require both discovering and also keeping track of the available connection slots on each browser, making the technique complex and error-prone. A more attractive solution is to employ JavaScript, as modern browsers provide hooks for a default action after a page is loaded (the *onLoad* handler) and when a request has failed (the *onError* handler). Using these controls, the attacker can easily chain requests to achieve serialization without the complexity of the previous technique. We therefore view the basic sandwich attack as a backup strategy in case JavaScript is disabled.

We have tested this attack scenario as shown in Figure 3.12. In a hidden frame, we load a page containing several image elements. The script points the source of each image to the reconnaissance target. Setting the source of an image element is an asynchronous operation. That is, after we set the source of an image element, the browser issues the request in a separate thread. Therefore, the requests to the various scan targets start at roughly the same time. After the source of each image is set, we wait for a timeout to be processed through the *onLoad* and *onError* handlers for every image. We identify the three cases (*e.g.,* unreachable, live, or non-responsive) similar to the sandwich attack but instead of issuing a request back to the malicious website to record the second timestamp we collect the results through the *onLoad/onError* handlers.

After the timeout expires, the results can be reported to the attacker, by means such as embedding timing data and IP addresses in a URL. The script can then proceed to another round of scanning. Each round takes time roughly equal to the timeout, which is normally controlled by the OS. It is possible for the attacker to use a smaller timeout through the *setTimeout()* primitive, which speeds up scanning at the expense of false negatives. We discuss this trade-off in Section 3.1.3.

There are both OS and browser restrictions on the number of parallel scans. On XP/SP2, the OS enforces a limit of no more than ten "outstanding"[5] connection requests at any given time [62]. Some browsers also impose limits on the number of simultaneous established connections. MSIE and Opera on Windows (without SP2), and browsers such as Konqueror on Linux, impose no limits, while Firefox does not allow more than 24. The attacker can choose between using a safe common-denominator value or employing JavaScript to identify the OS and browser before deciding on the number of parallel scans. The code for scanning

[5]A connection is characterized outstanding when the SYN packet has been sent but no SYN+ACK has been received.

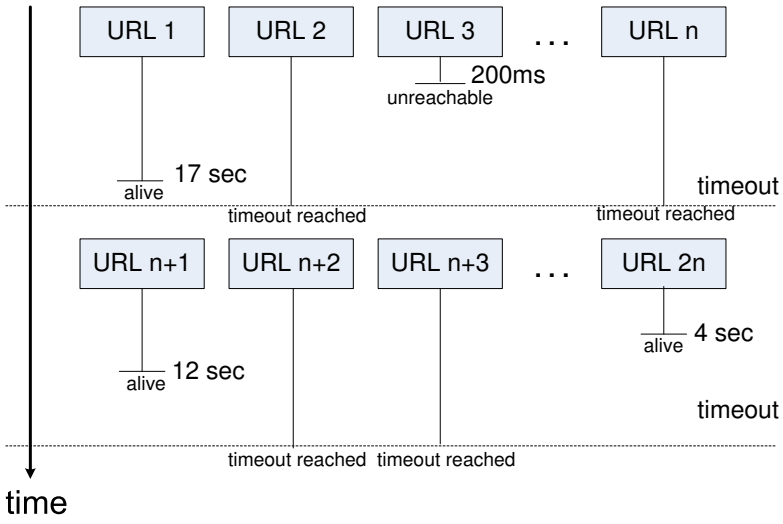


Figure 3.12: **Illustration of reconnaissance probing method.**

is shown in Figure 3.13.

We also considered the case where JavaScript is disabled. In that case, the effectiveness of scanning is constrained by the maximum number of parallel connections allowed by the browser and the operating system. With JavaScript-disabled, the attacker has to revert to the sandwich attack and make assumptions or take measurements to determine the timeout values and maximum connections for different browsers and operating systems. Moreover, the attacker cannot cancel the timeout and move on to the next request as is the case with JavaScript. We have experimentally found that for both Windows and Linux systems, a web connection times out after 22 seconds. That is we can have roughly 3 scanning sessions per minute. A simplifying assumption for the attacker would be to assume that all users have XP/SP2 installed on and that all connection slots in the browser are free. As the number of outstanding connections is limited by the operating system to 10, then the attacker is limited to scan for around 30 hosts per minute. Without SP2, this number increases up to 72, as attacker has 24 available slots to search for hosts.

The same process can be used to identify services other than web servers. When connecting to such a service, the browser will issue an HTTP request as usual. If the remote server responds with an error message and closes the connection, then the resulting behavior is the same as probing web servers. This is the case for many services, including SSH: the SSH daemon will respond with an error message that is non-HTTP-compliant and cannot be understood by the browser, the browser will subsequently display an error page, but the timing information is still relevant for reconnaissance purposes. This approach, however, does not work for all services, as some browsers block certain ports: MSIE blocks ports FTP, SMTP, POP3, NNTP and IMAP to prevent spamming through web pages (we return to this problem in Section 3.1.4); Firefox blocks a larger number of ports [54]; interestingly, Apple's Safari does not impose any restrictions. The attacker can rely on the "User-agent" string to trigger browser-specific code.

As described so far, puppetnets are limited to determining only the *liveness* of a remote target. As the same-domain policy restricts the attack to timing information only, the attack script cannot relay back to the attacker information on server software, OS, protocol versions,

```
[fontshape=sl,fontfamily=courier,frame=single,fontsize=\footnotesize]

<script language="javascript">
function printTimes() {
    var i=0;
    var curdate = new Date();
    var start_time= curdate.getTime();

    for(i=0;i<parallel_scans;i++) {
        if(loadTime[i]>0)
          info = info+"url="+victim[i]+",time="+loadTime[i]+"\n";
        victim[i]="http://"+createRandomIP();
        document.getElementById("frame"+i).src=victim[i];
        loadTime[i]=0;
        scanStarted[i]=start_time;
    }
    setTimeout('printTimes()',20000);
}

function startScanning() {
    var i=0;
    init();
    var curdate = new Date();
    start_time= curdate.getTime();

    for(i=0;i<parallel_scans;i++) {
       scanStarted[i]=start_time;
       /* stopTimer calculates the difference between the time
       we issued the request and the time requested URL was
       loaded or we had an error (not found) */
       document.write("<img id=frame"+i+" src=\""+victim[i]+"\"
       onload=\"stopTimer("+i+")\"
       onerror=\"stopTimer("+i+")\"></img>");
    }
    setTimeout('printTimes()',20000);
}

</script>
```

Figure 3.13: **Scanning for liveness of hosts through puppets**

etc., which are often desirable. Although this is a major limitation, distributed liveness scans can be highly valuable to an attacker. An attacker could use a puppetnet to evade detectors that are on the lookout for excessive numbers of failed connections, and then use a smaller set of sources to obtain more detailed information about each live target. Recent work by Bortz

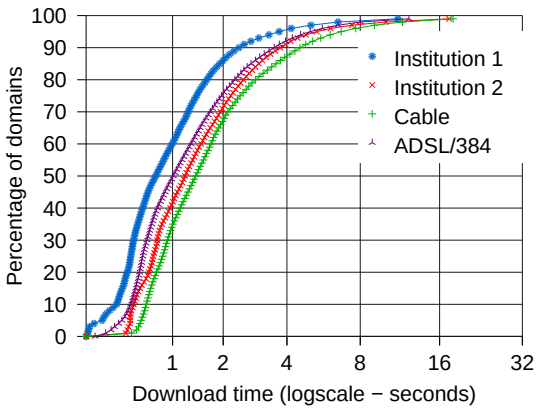


Figure 3.14: **Time to get main index from different sites.**

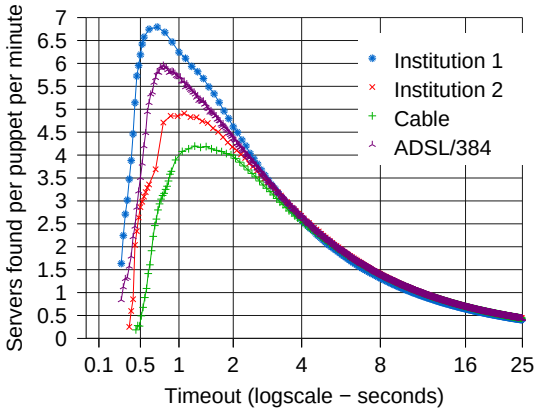


Figure 3.15: **Discovery rate, per puppet.**

et al. [77] has also shown that timing attacks can extract private information such as if a user is logged in at a certain site or way to learn the number of objects in the user's shopping cart. The authors of [77] use a similar technique based on JavaScript timing functions, as it is described in the following section.

Analysis of reconnaissance probing

There is a subtle difference between worm scanning and reconnaissance scanning. In worm scanning, the attacker can opportunistically launch probes and does not need to observe the result of each probe. In contrast, reconnaissance requires data collection and reporting.

There are two parameters in the reconnaissance attack that we need to explore experimentally: the timeout for considering a host non-responsive, and the threshold for considering a host unreachable. The attacker can tune the timeout and trade off accuracy for speed of data collection. The unreachable threshold does not affect scanning speed, but if it is too large it may affect accuracy, as it would be difficult to distinguish between unreachable hosts and live hosts. Both parameters depend on the properties of the network through which the browser is performing the scans.

In our first experiment we examine how the choice of timeout affects reconnaissance speed and accuracy and whether the unreachable threshold may interfere with reconnaissance accuracy. As most browsers under the attacker's control are expected to stay on the malicious page only for a few minutes, the attacker may want to maximize the scanning rate. If the timeout is set too low, the attacker will not be able to discover hosts that would not respond within the timeout.

Note that in the case of XP/SP2, the timeout must be higher than the default connection timeout of the OS, which is 25 seconds. The reason is that the scanning process has to wait until outstanding connections of the previous round are cleared before issuing new ones. The analysis below is therefore more relevant to non-XP/SP2 browsers. We succeeded, in part, in circumventing the XP/SP2 connection limit by having the browser "refresh" the scan page. This caused connection state to be released, presumably because of a request to close the socket. Oddly, this workaround only worked for small numbers of parallel connections. At this point we do not fully understand how the connection limiter operates, and whether the

workaround can be perfected.

We measure the time needed to download the main index file for roughly 50,000 unique websites, obtained through random queries to a search engine. We perform our measurements from four hosts in locations with different network characteristics. The distributions of the download times are presented in Figure 3.14. We see that in all cases, a threshold of 200-300 msec would result in a loss of around 5% of the live targets, presumably those within very short RTT distance from the scanning browser. We consider this loss to be acceptable.

Recall that the goal of the attacker may be speed rather than efficiency. That is, the attacker may not be interested in finding all servers, but finding a subset of them very quickly. We use simulation, driven by our measured distributions, to determine the discovery rate for a puppet using different timeout values, assuming 200 msec as the unreachable threshold. The results are summarized in Figure 3.15. For the four locations in our study, the peak discovery rate differs in absolute value, and is maximized at different points, suggesting that obtaining optimal results would require effort to calibrate the timeout on each puppet. However, all sources seem to perform reasonably well for small timeouts of 1-2 seconds.

In our second experiment we look at a 2-day packet trace from our institution and try to understand the behavior of ICMP unreachable notifications. We identify roughly 23,000 distinct unreachable events. The RTTs for these notifications were between 5 msec and 18 seconds. Nearly 50% responded within 347 msec, which, considering the response times of Figure 3.14, would result in less than 5% false negatives if used as a threshold. The remaining 50% of unreachables that exceed the threshold will be falsely identified as live targets, but as reported in [73], only 6.36% of TCP connections on port 80 receive an ICMP unreachable as response. As a result, we expect around 3% of the scan results to be false positives.

3.1.4 Protocols other than HTTP

One limitation of puppetnets is that they are bound to the use of the HTTP protocol. This raises the question of whether any other protocols can be somehow "tunneled" on top of HTTP. This can be done, in some cases, using the approach of [169, 81]. Briefly, it is possible to craft HTML forms that embed messages to servers understanding other protocols. The browser is instructed to issue an HTTP POST request to the remote server. Although the request contains the standard HTTP POST preamble, the actual post data can be fully specified by the HTML form. Thus, if the server fails gracefully when processing the HTTP part of the request (*e.g.,* ignoring them, perhaps with an error message, but without terminating the session), all subsequent messages will be properly processed. Two additional constraints for the attack to work is that the protocol in question must be text-based (since the crafted request can only contain text) and asynchronous (since all messages have to be delivered in one pass).

In this scenario, SMTP tunneling is achieved by wrapping the SMTP dialog in a HTTP POST request that is automatically triggered through a hidden frame on the malicious web page. For IRC servers that do not require early handshaking with the user (*e.g.,* the *identd* response), a browser can be instructed to login, join IRC channels and even send customized messages to the channel or private messages to pre-selected list of users sitting in that channel. This feature enables the attacker to use puppetnet for certain attacks such as triggering botnets, flooding and social engineering. The method is pretty similar to SMTP. An example of how a web server could instruct puppets to send spam is provided in [120].

Although this vulnerability has been discussed previously, its potential impact in light of a

```
[fontshape=sl,fontfamily=courier,frame=single,fontsize=\footnotesize]
<FORM METHOD="POST" NAME="myform"
      onSubmit="return nextjob()"
      enctype="multipart/form-data"
      ACTION="http://mailserver:25/foo">
<TEXTAREA name="thetext" rows="0" cols="0">

MAIL FROM: <spammer@marketing.com>
RCPT TO: <victim@target.com>
DATA
Subject: viagra

ok lah
.
QUIT

</TEXTAREA>
</FORM>
<body onLoad="document.myform.submit()">
```

Figure 3.16: **Spam transmission through puppets**

puppetnet-like attack infrastructure has not been considered, and vendors may not be aware of the implications of the underlying vulnerability. We have found that although MSIE refuses outgoing requests to a small set of ports (including standard ports for SMTP, NNTP, *etc.*) and Firefox blocks a more extensive list of ports, Apple's Safari browser as well as MSIE5.2 on Mac OSX do not impose *any* similar port restrictions[6]. Thus, although the extent of the threat may not be as significant as DDoS and worm propagation, popular websites with a large Apple/Safari user base can be easily turned into powerful spam conduits. The example code for sending spam can be seen in Figure 3.16.

3.1.5 Exploiting cookie-authenticated services

A large number of Web-based services rely on cookies for maintaining authentication state. A typical example is Web-based mail services that offer a "remember me" option to allow return visits without re-authentication. Such services could be manipulated by a malicious site that coerces visitors to post forms created by the attacker with the visitors' credentials. There are three constraints for this attack. First, the inline frame needs to be able to post cookies; this works on Firefox, but not MSIE. Second, the attacker needs to have knowledge about the structure and content of the form to be posted, as well as the target URL; this depends on the site design. Finally, the attacker needs to be able to instruct browsers to automatically post such forms; this is possible in all browsers we tested.

We have identified sites that are vulnerable to this attack.[7] As proof-of-concept, we have

[6] We have informed Apple about this vulnerability.

[7] These sites include a very popular Web-based mail service, the name of which we would prefer to disclose only upon request.

successfully launched an attack to one of our own accounts on such a site. Although this seems like a wider problem (*e.g.,* it allows the attacker to forward the victim's email to his site, *etc.*), in the context of puppetnets, the attacker could be on the lookout for visitors that happen to be pre-authenticated to one of the vulnerable websites, and could use them for purposes such as sending spam or performing mailbomb-type DoS attacks. Once again, puppetnets here only amplify an existing flaw that in many scenarios may be regarded as one of minor importance, as it may not be potent enough to cause significant damage in isolated incidents. However, in some cases this flaw can be a serious concern as it enables targeted attacks. The most common defense to this kind of attack is to require session-specific state in any form submission request which is only known to the relevant browser windows or frames, or to perform strict referrer checking on the server side.

Given the restriction to Firefox and the need to identify visitors that are pre-authenticated to particular sites, it seems that this attack would only have significant impact on highly popular sites, or moderately popular sites with unusually high session times, or sites that happen to have an unusually large fraction of Firefox visitors. Considering these constraints, the attack may seem weak compared to the ubiquitous applicability of DoS, scanning, and worm propagation. Nevertheless, none of these three scenarios can be safely dismissed as unlikely.

3.1.6 Distributed malicious computations

So far we have described scenarios of puppetnets involved in network-centric attacks. However, besides network-centric attacks, it is easy to imagine browsers unwillingly participating in malicious computations. This is a form of Web-based computing which, to the best of our knowledge, has not been considered as a platform for malicious activity. Projects such as RC5 cracking [100], use the Web as a platform for distributed computation but this is done with the users' consent. Most large-scale distributed computing projects rely on stand-alone clients, similar to SETI@home [116]. This particular form of abuse differs from the attacks we described previously as it does not directly involve browser resources being abused to attack a third-party, as the victim may be "offline". Furthermore, there is no well-defined policy as to what is legitimate and what may be harmful in terms of computation – this may well be subjective, and there may be no way to ensure that CPU cycles are used for benign purposes. This makes it difficult to defend against such abuse, but also illustrates a larger point regarding puppetnets, in that they involve *amplification* of threats through large-scale abuse of browser resources.

It is easy to instruct a browser to perform local computations and send the results back to the attacker. Computation can be done through JavaScript, Active-X or Java applets. By default, Active-X does not appear attractive as it requires user confirmation. JavaScript offers more stealth as it is lightweight and can be made invisible. Sneaking Java applets into hidden frames on malicious websites seems easy, and although the resources needed for instantiating the Java VM might be noticeable (and an "Applet loaded" message may be displayed on the status bar), it is unlikely to be considered suspect by a normal user.

To illustrate the extent of the problem we measured the performance of JavaScript and Java applets for MD5 computations. On a low-end desktop, the JavaScript implementation can perform around 380 checksums/sec, while the Java applet within the browser can compute roughly 434K checksums/sec – three orders of magnitude faster than JavaScript. Standalone Java can achieve up to 640K checks/sec. In comparison, an optimized C implementation

computes around 3.3M checks/sec. Hence, a 1,000-node puppetnet can crack an MD5 hash as fast as a 128-node cluster.

3.2 Defenses

In this section we examine potential defenses against puppetnets. The goal is to determine whether it is feasible to address the threat by tackling the source of the problem, rather than relying on techniques that attempt to mitigate the resulting attacks, such as DDoS, which may be hard to implement right at a global scale.

We discuss various defense strategies and the tradeoffs they offer. We concentrate on defenses against DDoS, scanning and worm propagation. Detecting malicious computations seems hard, and well beyond the scope of this work. Cookie-authenticated services seem trivial to protect by adding non-cookie session state that is communicated to the browser when the user wishes to re-authenticate.

Disabling JavaScript The usual culprit, when it comes to Web security problems, is JavaScript, and it is often suggested that many problems would go away if users disable JavaScript and/or websites refrain from using it. However, the trade-off between quality content and security seems unfavorable: the majority of websites employ JavaScript, there is growing demand for feature-rich content, especially in conjunction with technologies such as Ajax[99], and most browsers are shipped with JavaScript enabled. It is interesting to note, however, that a recently-published Firefox extension that selectively enables JavaScript only for "trusted" sites [125] has been downloaded 7 million times roughly one month after its release on April 9th, 2006.

In the case of puppetnets, disabling JavaScript could indeed alter the threat landscape, but it would only reduce rather than eliminate the threat. The development of our attacks suggests that even without JavaScript, it would still be feasible to launch DDoS, perform reconnaissance probes and propagate worms. We have found that DDoS is almost as potent without JavaScript. On the other hand, scanning and worm propagation are much slower and more complicated. Considering these observations, disabling JavaScript does not seem like an attractive proposition towards completely eliminating the puppetnet threat.

Careful implementation of existing defenses We observe that in most cases the attacks we developed were quite sensitive to minor tweaks. That is, although simple versions of the attack were quite easy to construct, maximizing their effectiveness required a lot more effort. Particularly the connection rate limiter implemented in XP/SP2 had a profound effect on the performance of worm propagation and reconnaissance. Unfortunately, we were able to demonstrate that the rate limiter can be partially bypassed. In particular, by reloading the core attack frame we were able to clear the TCP connection cache, presumably because a frame reload results in the underlying socket being closed and the TCP connection state entry being removed.

In this particular case it appears easy to address the problem by means of separating the actual connection state table from the state of the connection rate limiter. The rate limiter could either mirror the regular connection state table but choose to retain entries for closed sockets up to a timeout, or keep track of aggregate connection state statistics. This could reduce the effectiveness of worm propagation of up to an order of magnitude.

Another case that suggests that existing defenses are not always properly implemented is the Spam distribution attack described in Section 3.1.4. Although both MSIE and Firefox have mitigated this problem, at least in part, through blocking certain ports, Apple's Safari and the OSX version of MSIE5.2 did not properly address this known vulnerability.

However, careful implementation of existing defenses is insufficient for addressing the whole range of threats posed by puppetnets.

Filtering using attack signatures We consider whether it is practical to develop IDS/IPS signatures for puppetnet attacks. In some cases it seems fairly easy to construct such a signature. For example, in the case of puppetnet-delivered spam it is easy to scan traffic for messages to the SMTP port that contain evidence of both a HTTP POST request *and* legitimate SMTP commands. This should cover most other protocols tunneled through POST requests.

Can we develop signatures for puppetnet DoS attacks? We could consider signatures of malicious web pages that contain unusually high numbers of requests to third-party sites. However, our example attack suggests that there are many possible variations to the attack, making it hard to obtain a complete set of signatures. Additionally, because the attacks are implemented in HTML and JavaScript, it appears unlikely that simple string matching or even regular expressions would be sufficient for expressing the attack signatures. Instead, more expensive analyzers, such as HTML parsers, would be needed.

Furthermore, obfuscation of HTML and JavaScript seems to be both feasible and effective [164, 146], allowing the attacker to compose obfuscated malicious web page on-the-fly. For example, one could use the *document.write()* method to write the malicious page into an array in completely random order before execution. This makes the attack difficult to detect using static analysis alone, a problem that is also found in shellcode polymorphism [82, 118, 140]. Although we must leave room for the possibility that such a unusual use of *document.write()* or similar approaches may be amenable to detection, such analysis seems complex and is likely to be expensive and error-prone.

Client-side behavioral controls Another possible defense strategy is to further restrict browser policies for accessing remote sites. It seems relatively easy to have a client-side solution deployed with a browser update, as browser developers seem to be concerned about security, and the large majority of users rely on one among 2-3 browsers.

One way to restrict DoS, scanning and worm propagation is to establish bounds on how a web page can instruct the browser to access "foreign" objects, *e.g.,* objects that do not belong to the same domain. These resource bounds should be persistent, to prevent attackers from resetting the counters using page reloads. For similar reasons, the bounds should be tied to the requesting server, and not to a page or frame instance, to prevent attackers from evading the restriction through multiple frames, chained requests, *etc.*

We consider whether it makes sense to impose controls on foreign requests from a web page. We attempt to quantify whether such a policy would break existing websites, and what impact it would have on DDoS and other attacks. We first look at whether we can limit the total number of different objects (*e.g.,* images, embedded elements and frames) that the attacker can load from foreign servers, considering all servers to be foreign except for the one offering the page. This restriction should be enforced across multiple automatic refreshes or frame updates, to prevent the attacker from resetting the counters upon reaching the limit.

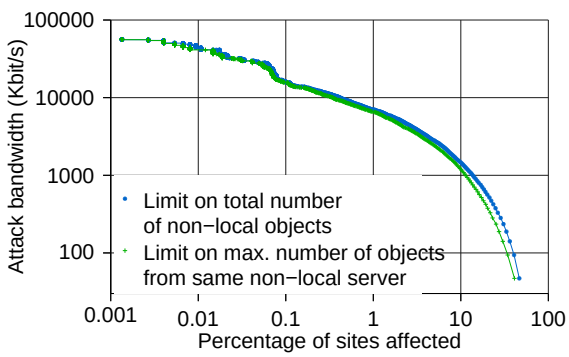


Figure 3.17: **Effectiveness of remote request limits**

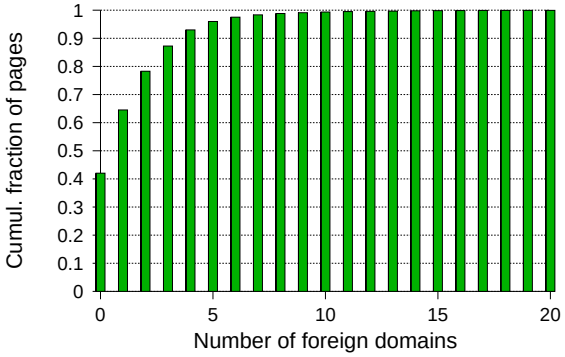


Figure 3.18: **Cumul. histogram of foreign domains referenced by websites**

(Counters would only be reset only when a user clicks on a link.) Of course, this is likely to "break" sites such as those that use automatic refresh to update banner ads. Given that ads are loaded periodically, *e.g.,* one refresh every few minutes, it seems reasonable to further refine the basic policy with a timeout (or leaky bucket mechanism) that occasionally resets or tops-up the counters.

To evaluate the effectiveness of this policy, we have obtained data on over 70,000 web pages by crawling through a search engine. For each web page we obtain the number of non-local embedded object references. We then compute for each upper bound N of non-local references, the fraction of sites that would be corrupted should such a policy be implemented, against the effective DoS bandwidth of a 1000-node puppetnet under the same policy. A variation of the above policy involves a cap on the maximum number of non-local references to the same non-local server.

The results are shown in Figure 3.17. We observe that this form of restriction is somewhat effective when compared to the DDoS firepower of Figure 3.1, providing a 3-fold reduction in DDoS strength when trying to minimize disruption to websites. The strength of the attack, however, remains significant, at 50 Mbit/s for 1000 puppets. Obtaining a 10x reduction in DDoS strength would disrupt around 0.1% of all websites, with DDoS reduced to 10 Mbit/s. Obtaining a further 10x reduction seems impractical, as the necessary request cap would negatively affect more than 10% of web pages. The variation limiting the max. number of requests to the same non-local server also does not offer any notable improvement. Given the need to defend against not only 1000-node but even larger puppetnets, we conclude that although the improvement offered is significant, this defense is not good enough to address the DDoS threat.

The above policy only targets DDoS. To defend against worms and reconnaissance probes, we look at the feasibility of imposing limits on the number of distinct remote servers to which embedded object references are made. The cumulative histogram is shown in Figure 3.18. We see that most websites access very few foreign domains: around 99% of websites access 11 or less foreign domains; around 99.94% of websites access less than 20 foreign domains. Not visible on the graph is a handful of websites, typically "container" pages, that access up to 33 different domains. Based on this profile, it seems reasonable to implement a restriction of around 10 foreign domains, keeping in mind that the limit should be set as low as possible,

given that a large fraction of puppets have a very short lifetime in the system. Note that sites that are out of profile could easily avoid violating the proposed policy, by proxying requests to the remote servers. We repeated the worm simulation of Section 2.2.1 to determine the impact of such a limit on worm propagation. As expected, this policy almost completely eliminates the speed-up for the client-server worm compared to server-only, as puppets can perform only a small fraction of the scans they could perform without this policy. Similar effects apply to scanning as well.

Unfortunately, the above heuristic can be circumvented if the attacker has access to a DNS server. The attacker could map all foreign target hosts to identifiers that appear to be in the same domain but are translated by the attacker's DNS server to IP addresses in the foreign domain. Attacks aiming at consuming egress bandwidth from servers that rely on the "Host:" tag in the HTTP request would be less effective, but all other attacks are not affected.

Server-side controls and puppetnet tracing Considering the DoS problem and the difficulty in coming up with universally acceptable thresholds for browser-side connection limiting, one could argue that it is the Web developers who should specify how their sites are accessed by third parties.

One way for doing that is for servers to use the "Referrer" tag of HTTP requests to determine whether a particular request is legitimate and compliant, similar to [174]. The server could consult the appropriate access policy and decide whether to honor a request. This approach would protect servers against wasting their egress bandwidth, but does not allow the server to exercise any control over incoming traffic.

Another use of referrer information can be to trace the source of the puppetnet attack, and take action to shutdown the control website. That is, puppetnets have a single point of failure. However, this process is relatively slow as it involves human coordination. Thus, attackers may already have succeeded in disrupting service. Moreover, even when the controlling site has been taken down, existing puppets will continue to perform an attack – the attack will only subside once all puppet browsers have been pointed elsewhere, which is likely to be in the order of 10-60 minutes, based on the viewing time estimates of Section 2.1.1.

However, as shown in [120], we have been able to circumvent the default behavior of browsers that set referrer information, making puppetnet attacks more difficult to filter and trace. It is unclear at this point if minor modifications could address the loss of referrer-based defenses. Thus, referrer-based filtering does not currently offer much protection and may not be sufficient, even in the longer-term, for adequately protecting against puppetnet attacks.

Server-directed client-side controls To protect against unauthorized incoming traffic from puppets, we examine the following approach. If we assume that the attacker cannot tamper with the browser software, a server can communicate site access policies to a browser during the first request. In our implementation, we embed Access Control Tokens (ACTs) in the server response through an extension header ("X-ACT:") that is either a blanket "permit/deny" or a JavaScript function, similar to proxy autoconfiguration[131]. This script is executed on the browser side for each request to the server to determine whether a request is legitimate or not. The use of JavaScript offers flexibility for site developers to design their own policies, without having to standardize specific behaviors or a new policy language.

The scope of ACTs is a sensitive issue. If a server is able to specify ACTs for the IP

Distribution	Firepower
KDDCUP	47.03
Google/WT	14.39
JSTracker	3.05
Web track	2.87

Figure 3.19: **Impact of ACT defense on 1000-puppet DDoS attack**

address it is responding to, this gives any hosted server on that same IP control over all other hosted servers, opening up an opportunity for abuse. The hosting system would have to perform scope checks on ACTs or otherwise ensure that servers cannot abuse ACTs. If a web server can only set policies for its own "domain" similar to the same-origin principle, then it may possible for a malicious or subverted server to create ACTs that result in denial-of-service on sub-domain servers. If this becomes an issue, it is straightforward to restrict the scope of ACTs to the same web server only, at the expense of having to set separate ACTs for each server involved in a service (e.g., image server, etc.). Another problem with domain-level ACTs is that the attacker can still use aliasing as discussed previously to amplify his attack. The use of certificates binding IP addresses to server or domain names, as we will discuss further in this section, is one way to address this problem.

Perhaps the simplest policy would be to ask browsers to completely stop issuing requests if the server is under attack. More fine-grained policies might restrict the total number or rate of requests in each session, or may impose custom restrictions based on target URL, referrer information, navigation path, etc. One could envision a tool for site owners to extract behavioral profiles from existing logs, and then turn these profiles into ACT policies. For a given policy, the owners can also compute the exposure in terms of potential puppetnet DDoS firepower, using the same methodology used in this work. The specifics of profiling and exposure estimation are beyond the scope of this thesis.

ACTs require at least one request-response pair for the defense to kick in, given that the browser may not have communicated with the server in the past. After the first request, any further unauthorized requests can be blocked on the browser side. Thus, ACTs can reduce the DoS attack strength to one request per puppet, which makes them quite attractive. On the other hand, this approach requires modifications to both servers and clients.

To illustrate the effectiveness of this approach, we estimate, using simulation, the firepower of a 1000-puppet DDoS attack where all users support ACTs on their browsers. The puppet viewing time on the malicious or subverted site is taken from the distributions shown in Figure 3.2. The victim site follows the most conservative policy: if a request comes from a non-trusted referrer then user is not allowed to make any further requests. The results are summarized in Table 3.19. As the attack is restricted to one request per user, the firepower is limited to only a few Kbit/sec.

In theory, it is possible to prevent the first unauthorized request to the target if policies are communicated to the browser out-of-band. One could directly embed ACTs in URL references, through means such as overloading the URL. Given that an ACT needs to be processed by the browser, it must fully specify the policy "by value". To prevent the attacker from tampering with the ACT, it must be cryptographically signed by the server. Besides being cumbersome, this also requires the browser to have a public key to verify the ACTs.

While (most) SSL websites already have certificates in place, this proposal is less attractive for all other non-SSL websites.

3.3 Publications

The work related to puppetnets was published to the following places:

- V. T. Lam, Spiros Antonatos, Periklis Akritidis and Kostas G. Anagnostakis. "Puppetnets: Misusing Web Browsers as a Distributed Attack Infrastructure." In ACM Transactions on Information and System Security (TISSEC), Vol. 12, Issue 2, December 2008
- V.T. Lam, S. Antonatos, P. Akritidis and K.G. Anagnostakis. "Puppetnets: Misusing Web Browsers as a Distributed Attack Infrastructure." In the Proceedings of the ACM Conference on Computer and Communications Security (CCS), November 2006

Chapter 4

The Network of Affined Honeypots

NoAH is focused on honeypots that listen to unused IP address space and analyze and/or interact with malicious traffic. The architecture of NoAH presents a flexible design for deployment and collaboration of honeypots. NoAH is not restricted to a single type of honeypot but tries to combine the good characteristics of both types. Its modular architecture permits the construction of a network of honeypots with minimal overhead and affordable administrative overhead.

Honeypots in NoAH are deployed inside the "NoAH core", the center of decisions. Apart from honeypot deployment, services like automated signature generation for zero-day attacks run inside the core. The NoAH core is not a centralized farm of honeypots. On the contrary, it is a distributed set of honeyfarms that can collaborate. Inside the core, both low-interaction and high-interaction honeypots are deployed. Low-interaction honeypots are used as a traffic filter. Therewith, activities like port-scanning can be effectively detected by LI honeypots and stop there. Traffic that cannot be handled by LI honeypots is handed over to HI honeypots. In this case, LI honeypots are used as proxies whereas HI honeypots offer the optimal level of realism. To prevent the HI honeypots from infections, a containment environment is used. Any containment environment can be used, like VMware, Xen or Qemu virtual machines. Another proposed environment is Argos

The address space covered by NoAH core can be further extended. Institutes, campuses or organizations can collaborate with NoAH by deploying a "plug" to NoAH core. This "plug" is actually a tunnel to NoAH core. All traffic going to dark space of a collaborative party is tunneled to honeypots in the NoAH core for processing. Replies from honeypots are tunneled back and injected to party's network. Using tunneling, honeypot deployment is not needed and thus the administrative overhead is minimal.

Apart from organizations and institutes, simple home users can help NoAH. Home users or small size enterprises can share their black address (or port) space in a similar way as the participating organizations described before. To do so, they install Honey@home. We will further describe the Honey@home approach at Section 5.

Overall, NoAH glues together various network and host components to form a flexible network of honeypots. Although the NoAH core is the main component of NoAH architecture, NoAH is more than a set of honeyfarms. Approaches like tunneling and honey@home extend NoAH far beyond its core and have the potential for wide address space coverage with minimal overhead. In the next Sections, each component of the architecture will be examined thoroughly.

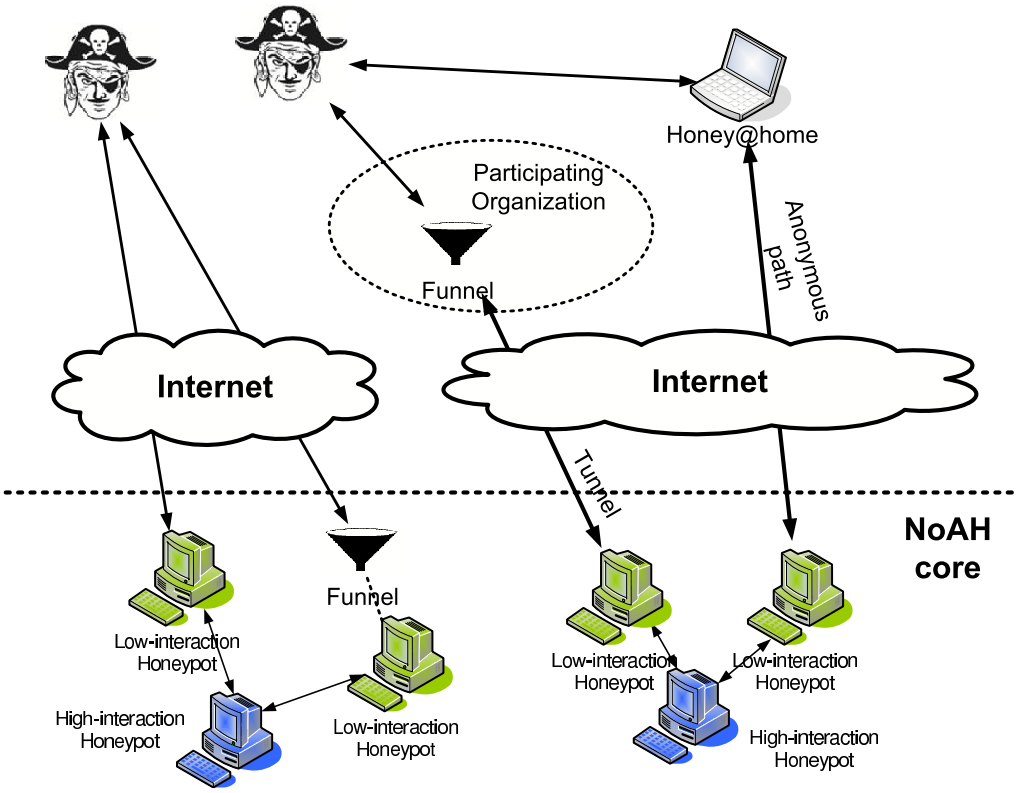


Figure 4.1: The overall architecture of NoAH. We can identify three main types of interaction with attackers: attackers communicate directly with NoAH honeypots, attackers aim at black space of cooperating organizations or finally they attack on home/enterprise black space (Honey@home). In the last two cases, traffic is forwarded to NoAH honeypots.

4.1 NoAH Core

The NoAH core is composed by both low- and high-interaction honeypots. It would be limiting to restrict the NoAH architecture to a single type of honeypot as both types present complementary characteristics. Low-interaction honeypots can be assisted by high-interaction ones in order to provide an advanced level of realism, while maintaining their security properties. In Figure 4.1, the overall architecture of NoAH can be seen at a glance. This Section aims at describing the NoAH core and more specifically, defining the level of interaction between low- and high-interaction honeypots along with the functionality of each honeypot. Before we proceed to the interaction between low- and high-interaction honeypots, a definition of their architecture needs to be established.

4.2 Low-interaction honeypots inside NoAH core

Low-interaction honeypots are basically a set of software-based components that emulate a workstation or home PC. The most popular tool for deploying low-interaction honeypots is honeyd. Honeyd is a small daemon that creates virtual hosts on a network and it can run on

both Unix and Windows platforms. It can be configured to run arbitrary services on multiple IP addresses. Honeyd is open-source and its distribution comes with a large set of scripts emulating services. Furthermore, its modularity permits the authoring of customized service emulation. The extended functionality of honeyd, its portability and its wide acceptance from the honeypot community are the main criteria for choosing it as the primary tool for setting up a low-interaction honeypot inside NoAH. For performance and security reasons, it is suggested that honeyd is installed on a Unix-like operating system.

A complete picture of how honeyd works was shown in Figure 2.2. The IP stack is emulated in user space and packets are delivered to TCP, UDP or ICMP handlers. The TCP handler is the component that performs TCP stack emulation. Generally, handlers forward packets to the appropriate services emulated by scripts. Packets with destination port 80 are e.g. handed over to the web server emulation script. These scripts can be either external programs or proxies to real services. The personality engine component is the one that is responsible for setting up a behavior for an emulated IP address. As described before, honeyd can emulate various operating systems at network level. Each configuration in the form of "this IP address will virtually belong to a machine running AIX 4.0" is called a personality. The personality engine writes the packet in such a way that its TCP/IP protocol fields appear as having been sent by a real AIX system.

For a low-interaction honeypot, it is necessary to specify a set of applications and services to emulate. To start with, services and applications running on a typical high-interaction honeypot can be emulated. For example, suppose we need to build a low-interaction honeypot that imitates Windows XP. A fresh installation of Windows XP in a real-machine or a containment environment environment, like VMware, will provide an initial list of services and open ports to emulate in the low-interaction honeypot. This procedure is done once and then customization is applied to each honeypot. This customization installs specific services/applications to specific honeypots, e.g. installation of IIS to honeypot A, mail server to honeypot B and so on. To blur attackers, tricks like setting up some low-interaction machines to behave like unpatched Windows and some others having installed SP2 can be used. After deciding which services will run on which honeypots, the partitioning of services must be viewed as a whole. It would be unrealistic, for example, to run more than ten IIS servers in the same /24 subnet. Typical views of services running on a subnet can be obtained through full TCP scanning in existing local subnets. Testing the diversity in some local –lightly loaded– subnets, we counted more than 40 services. We expect this number to increase with the number of machines in a subnet.

4.3 Medium-interaction honeypots inside the NoAH core

Amun [3] is a medium-interaction honeypot designed to capture autonomous spreading malware in an automated fashion. Medium-interaction honeypots are more preferable due to their processing speed and acceptable level of emulation. Amun operates in a way very similar to Nepenthes, and also bases its functionality on the same five types of modules. The main difference between the two is that Amun may have more than one vulnerability module for a specific port, allowing the interaction with a wider range of attacks. Furthermore it is more up-to-date than Nepenthes with a richer selection of vulnerability modules. Finally, Amun is written in Python and therefore allows the easy integration of new features. We integrated Amun with the NoAH infrastructure. Amun processes the traffic coming from the

Honey@home clients and one of our sensors.

We forward traffic from one of our sensors to the Amun honeypot. Some collection statistics will be presented in Section 4.7.

4.4 High-interaction honeypots inside the NoAH core

High-interaction honeypots are ordinary machines that run real services. Services can run either on top of the operating system or inside a containment environment. High-interaction honeypots of NoAH core run a variety of operating systems, either real or virtual ones, from Windows to most well-known Linux flavors. To facilitate the deployment of machines, automatic installation through images retrieved from systems like Quattor [35] can be used. In any case, high-interaction honeypots should be instrumented to log traffic and/or system calls. Tools like Sebek [38] can help to this direction. All services and applications that will run on high-interaction honeypots do not need any instrumentation. Furthermore, no special software component needs to run on each operating system. High-interaction systems will run as they would in a real environment.

The containment environment that is mainly used is Argos, developed by Vrije University. As virtual machine environments are heavyweight processes (a performance comparison between VMware, Xen and UML can be found at [49]) only a few can run per physical machine and this fact has put us several constraints during the deployment phase.

An overview of the Argos architecture is shown in Figure 4.2. The full execution path consists of five main steps, indicated by the numbers in the figure which correspond to the circled numbers in this section. Incoming traffic is both logged in a trace database, and fed to the unmodified application/OS running on our emulator (step 1). In the emulator we employ dynamic taint analysis to detect when a vulnerability is exploited to alter an application's control flow (step 2). This is achieved by identifying illegal uses of possibly unsafe data such as the data received from the network [134]. These three steps are accomplished by tagging data originating from an unsafe source as tainted, tracking tainted data during execution and finally identifying and preventing unsafe usage of tainted data.

Data originating from the network is marked as tainted, whenever it is copied to memory or registers, the new location is tainted also, and whenever it is used, say, as a jump target, we raise an alarm. Thus far this is similar to approaches like [87] and [134]. As mentioned earlier, NoAH Containment differs from most existing projects in that we trace physical addresses rather than virtual addresses. As a result, the memory mapping problem disappears, because all virtual address space mappings of a certain page, refer to the same physical address. There are various options for determining which behavior should trigger an alert. Argos opts for illegitimate use of tainted data, including loading the processor's program counter register (EIP) with a tainted value, or loading EIP with an address whose contents are tainted. Unsafe usage of tainted data is prevented because Argos detects the exploit before it can take control of the vulnerable program. We also plan to check the arguments of some system calls for use of tainted data and may add other methods later. When a violation is detected, an alarm is raised which leads to a signature generation phase (steps 3 to 5). To aid signature generation, Argos will first dump all tainted blocks and some additional information to file, with markers specifying the address that triggered the violation, the memory area it was pointing to, etc. Since we have full access to the machine, its registers and all its mappings, we are able to translate between physical and virtual addresses as needed. The dump therefore contains

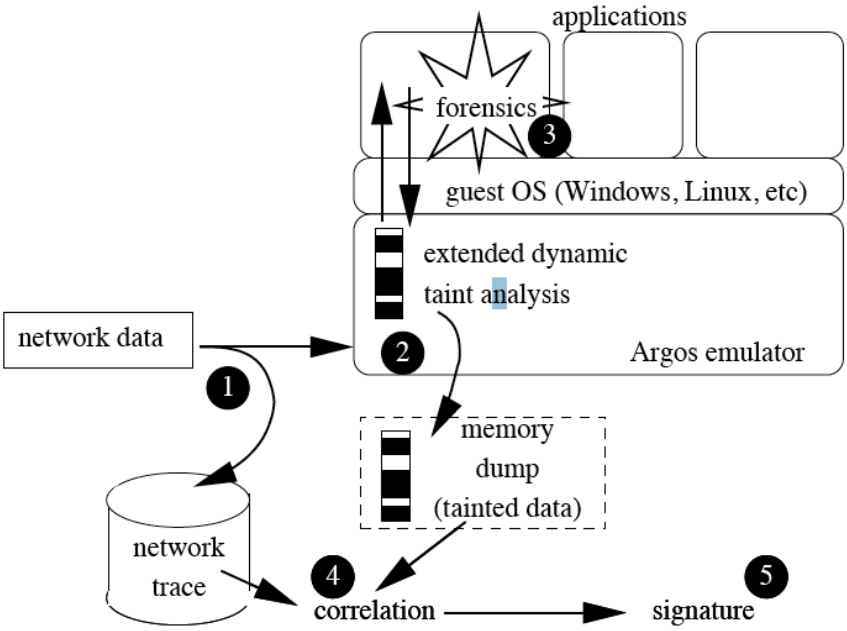


Figure 4.2: High-Level overview of Argos (1) Network data arrives and is both logged and sent to the emulator. (2) The emulator tags data from the network as tainted. (3) An alert is triggered if tainted data is used in ways that violate the security policy and forensics about the process under attacks is logged. (4) The memory log, forensics data, and network trace are correlated and (5) a signature is generated.

```
|98 91|KCBJ7J|99 98|G|F8|@HHA?IK7N|9B F8|N|97 9F 9F|
?JI0|EB 10 EB 19 9F| u| 18 00| #7| F3| w| EB E0 FD 7F|
A|F5|A|FC|F|90 9B|C?|D6 98 91 FC 93 98 92 F9|K|FC|J
|9B 92|H|FC 99 D6|A|96|0J|93|N|FC|O|FD|CC|97 96|J|91
92|JAKI?|27|B@|99|G|99 F5|I|93 F8|C|D6 27|07|90 91|
7O|D6 99|@H?|FD 27 91|BI|F9 97|H|D6 96 98|?|91 93 97
F8 FD EB 04 FF FF FF FF|J|92|G|93 92|7|9F 98 EB 04
EB 04 92 9F|?@|EB 04 FF FF FF FF 97|CI|F8 F5 FC|FKK@
0JHF|96|GHN|92 9B|K|93|F7|0A|
```

Figure 4.3: An example of a signature generated by Argos. The highlighted part of the signature is the invariant part of the attack vector.

registers, physical memory blocks and specific virtual address, as explained later, and in fact contains enough information not just for signature generation, but for, say, manual analysis as well.

The dump of the memory blocks (tainted data, registers, etc.) plus the additional information obtained by our shellcode is then used for correlation with the network traces in the trace database (step 4). In case of TCP connections, we have to reconstruct flows prior to correlation. We use the correlation to generate signatures. An example of a signature generated by Argos can be seen in Figure 4.3. The highlighted part of the signature is the

invariant part, this is the part of the signature that remains constant after multiple instances of the attack have been seen. We should observe that many different signature generation can be plugged in the NoAH back-end. Two approaches are used for signature generation. The first one locates the longest common sub-sequence between the memory footprint and the network trace. The second one, called CREST, finds the memory location that allowed the attacker to take control of the system. This memory location is found using the physical memory origin and value of EIP register, that is the instruction pointer register. The value of EIP register is located inside the trace and then trace is extended to the left and right. The extension stops when different bytes are encountered. The resulting byte sequence, along with protocol and port number, is used as a signature. For both approaches, a network trace is useless if data in it is encrypted, for example it is a HTTPS connection. Latest advances of Argos allow it to correlate memory dump with unencrypted data, as Argos comes with modified versions of secure socket libraries for some guest operating systems.

4.5 Interaction between low and medium/high-interaction honeypots

Low-interaction honeypots are used as the front-end of the NoAH core. Nobody is able to directly communicate with high-interaction honeypots, except the low-interaction ones. By assigning low-interaction honeypots the role to handle the communication with attackers, the integrity of NoAH core is assured as low-interaction honeypots never run real services and high-interaction honeypots are inside a containment environment. To reduce the potential damage of high-interaction honeypots, they run inside a containment environment. All connections are recorded for further analysis from deployed detection mechanisms. An alternative solution is to deploy reflectors to the subnet of high-interaction honeypots. Reflectors provide the illusion that a connection is established to a machine outside the subnet but in reality they redirect the connection to another honeypot [173].

Honeyd scripts for emulating various services, like a Web server, are not able to interact with attackers past a certain level. Emulating a Web or FTP server is much more trivial than emulating more complicated services, like RPC which presents increased protocol complexity. Certain attacks trigger the real exploit after extended communication with their target. As an example, RPC attacks may require more than ten message exchanges with the target before they send their real exploit. Scripts cannot interact up to this depth and would stop being realistic after a few message exchanges. They may even not be able to provide an answer, e.g. when they see an abnormal request or unidentifiable protocol versions. In such cases, the assistance of high-interaction honeypots is required. Another problem of scripts is that for every service that a high-interaction honeypot provides, a script that emulates that service should exist or be written. If we consider that there are hundreds of services, and several versions of each service, it is impractical to write, test and maintain a huge number of emulation scripts. Approaches like ScriptGen [121] try to automatically create scripts for honeyd but there is room for enhancement.

Low-interaction honeypots in the NoAH core are used as lightweight responders. These responders will reduce the traffic that is destined to the high-interaction honeypots inside NoAH core. The architecture is illustrated in Figure 4.4. A few low-interaction honeypots use the services running on high-interaction honeypots to provide realistic responses to attackers. High-interaction honeypots are not directly accessible by attackers and if they get infected,

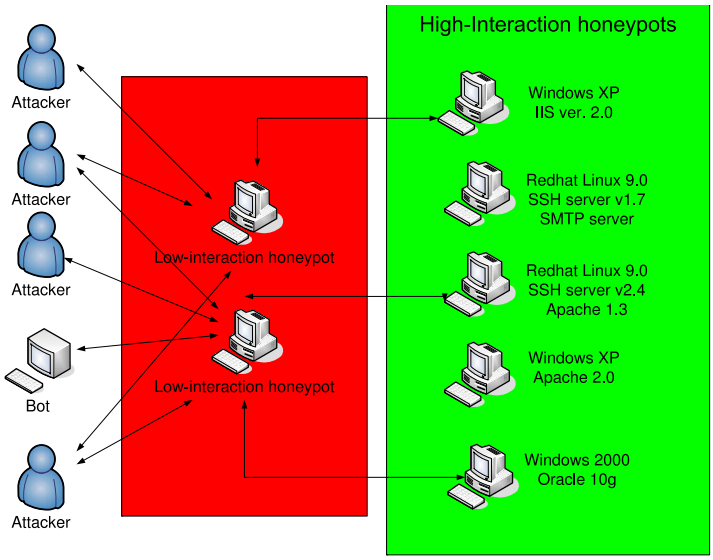


Figure 4.4: Cooperation between low- and high-interaction honeypots. Low-interaction honeypots are accessible by attackers, while high-interaction ones are placed in a private subnet. Attackers reach low-interaction honeypots which, in their turn, use high-interaction ones to provide real responses.

the infection is contained in their private subnet.

How low-interaction honeypots can act as lightweight proxies is described at the following section.

4.5.1 Lightweight proxies

The initial part of conversation with an attacker is performed by a low-interaction honeypot. The low-interaction honeypots inside NoAH core act as lightweight proxies. These proxies are honeyd instances that listen to specific ports as specified in the configuration file of honeyd. Each open port is mapped to a specific service of a high interaction honeypot. For example, if a low-interaction honeypot imitates a Windows machine, then port 80 is mapped to a high-interaction honeypot that runs the Windows operating system and that has e.g. IIS installed.

The lightweight proxies inside the NoAH core respond only to TCP SYN requests to ports that are open. For any other ports, it just absorbs and records the packets received. When the three-way handshake has completed properly between the attacker and the low interaction honeypot, the connection must be handed-off to the appropriate high interaction honeypot. At this point, also referred as zero point, the low interaction honeypot sets up a connection with the high interaction honeypot. The low-interaction honeypot now acts like a relay agent. Any application-level data coming from attacker is forwarded to the high-interaction honeypot and vice versa , until the connection is terminated. This behavior is embedded to the honeyd implementation, known as proxy mode. The proxy mode is instrumented to record the message exchanges, for further analysis purposes. Handoff is useful in cases of port scanning, where low-interaction honeypots will absorb all incoming connections without

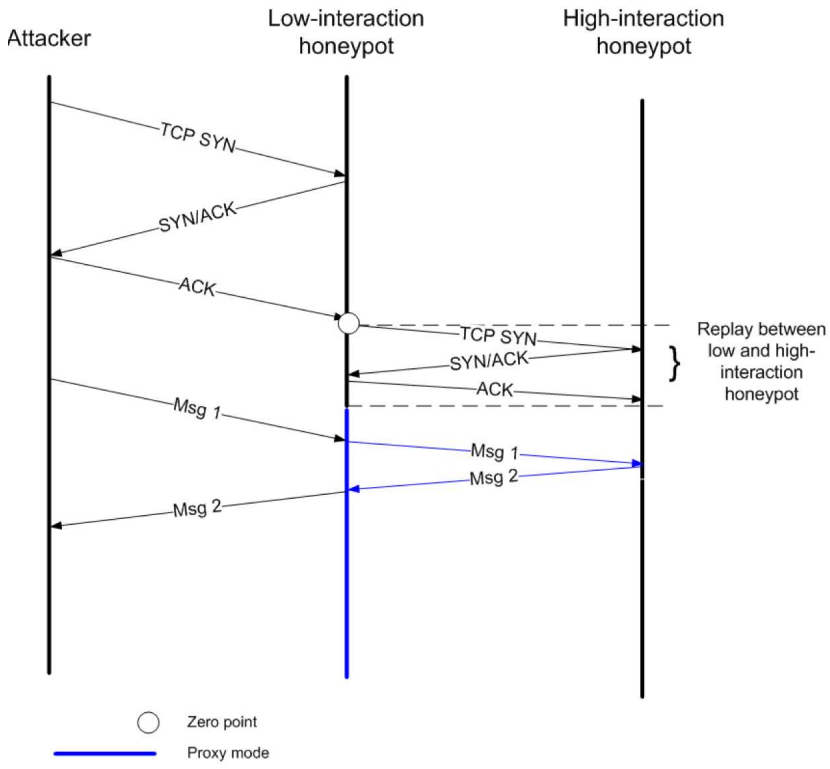


Figure 4.5: An example of connection handoff. Upon connection establishment, low-interaction honeypot becomes a proxy between the attacker and the high-interaction honeypot

disturbing high-interaction honeypots.

An example of connection handoff is illustrated in Figure 4.5. Initially, the attacker sends a TCP SYN packet to the low-interaction honeypot. If the honeypot is configured to listen to this port, then it sends a SYN/ACK packet and waits to receive the next packet. If that packet is not an ACK then the low-interaction honeypot assumes that it was a port scan and the connection is dropped (or ignored based on the configuration settings). If the third packet received is ACK then it is a valid TCP connection and the zero point is reached. Thus the low-interaction honeypot connects with the high-interaction honeypot running the requested service. Then after the connection establishment the low-interaction honeypot continues to work as a proxy. As low- and high-interaction honeypots belong to the same local network, no additional delay will be perceived by the attacker.

Using the above scheme for the communication between low and high interaction honeypots we achieve a number of goals. First, we need to maintain only a small number of high interaction honeypots since the portion of the traffic is routed to them is limited. All port-scan attempts or connections to ports that are not open will be stopped by the low-interaction honeypots. Second, the high interaction honeypots are placed in a monitored network. Thus if a honeypot gets infected, the infection rate is controllable either through limiting bandwidth or traffic reflection. Also, since honeyd can emulate different machines running in a network, we can map several machines which run the same operating system and similar services to a

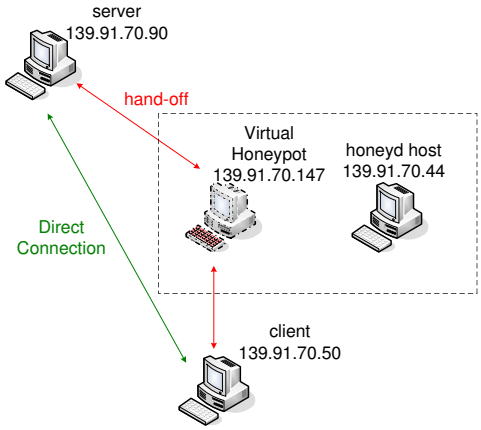


Figure 4.6: Experimental setup for measuring the overhead of connection handoff. In the case of handoff, client downloads a file from virtual server 139.91.70.147 but her connection is proxied to real server at 139.91.70.90

single high-interaction honeypot.Finally, the addition of new services to the high-interaction honeypot is facilitated we only should open appropriate port at the low-interaction honeypot and set up the mapping.

4.5.2 Overhead of connection handoff

We measured the overhead added by the handoff mechanism. We used a normal client, a Web Server running Apache and a low-interaction honeypot running honeyd with the handoff mechanism enabled. The experiment setup can be seen in Figure 4.6. All three machines belong to the same subnet based on 100Mbit/s links. In our experiment, the client performed a series of HTTP requests for files of variable size, from 10 MBytes up to 1 GByte. We measured the time needed to download a file with and without (direct connection) connection hand-off enabled. In the case of hand-off, client asks to download a file from a virtual host (139.91.70.147) that does not run a web server, but instead is emulated by honeyd. The honeyd hands the TCP connection off to the real server giving the client the illusion that she downloads a file from 139.91.70.147.

We found that the added overhead is independent of the file size, as expected, and it is ranged between 130 and 140% (the measured overhead for each filesize is an average of ten runs). We verified these numbers by doing the same experiment using *scp* instead of the HTTP requests. There were no noticeable difference.

4.6 Dark space monitoring

In this Section, we will present the mechanisms that will be used to redirect traffic to the NoAH core. Our approach covers both those organizations that host their own NoAH honeypots (*Core Organizations*) as well as those that only wish to donate some of their IP space to the NoAH infrastructure (*Cooperating Organizations*).

Regarding the Core Organizations, we should stress that while the NoAH core (as described in Section 4.1) is tiered in a front-end of low-interaction honeypots assisted by a backend of high-interaction honeypots, the NoAH architecture is not strictly bound to this scheme. Core Organizations may set up their own farm of honeypots, either low- or high-interaction ones. Thanks to the modularity of the NoAH architecture, they have the flexibility to take their own decisions (and risks as well).

4.6.1 Dark space

Typically a respectful portion of the IP address space that is allocated to an organization remains unused. This unused IP address space, also referred as *Dark Space*, can be utilized by the NoAH infrastructure to gather traffic for the NoAH core honeypots. However, the number of honeypots in the NoAH core is limited compared to the size of the dark space. The dark space of a medium-sized organization typically measures a few hundreds of IP addresses, while for larger organizations it may measure thousands of IP addresses. It is unreasonable to expect that for each IP address in the dark space a physical machine, which may be either a high- or a low-interaction honeypot, will be available.

The solution to this size mismatch is to have each physical honeypot handle traffic for a range of IP addresses instead of only one. E.g. a single low-interaction honeypot can be configured to handle traffic for a whole /16 dark subnet, or a high-interaction honeypot can have its network interface configured with multiple IP addresses.

In general, software for handling traffic from multiple IPs on a single host is readily available (e.g. honeyd[142]). The open issue that we have to address is how will the desired traffic be gathered to this host. Our solution to this problem is *funneling*.

4.6.2 Funneling

The main abstraction of *funneling* is that traffic going to a set of IP addresses will end up to be processed by a single machine. The goal is to create a virtual *funnel* that concentrates traffic from portions of dark space to a limited number of honeypots. An example of how funneling works is illustrated in Figure 4.7. In our example, we have a single honeypot and we map the dark space IP addresses 11.12.0.1 to 11.12.255.255 to this honeypot. The honeypot host is reachable within the subnet using the IP 11.12.1.1.

Funneling is a two step process. First, the dark IP address space traffic is redirected to the honeypot location within the Core Organization and secondly, each honeypot claims a portion of this traffic. In practice, the first step involves configuring the organization's routers, so that the dark space traffic ends up to the local area network where the honeypots are plugged. In the second step, the honeypots are configured to claim their respective share of dark space traffic. We suggest the shares to be allocated statically: the dark space is divided by the number of honeypots and each honeypot gets traffic from an equal number of dark IP addresses.

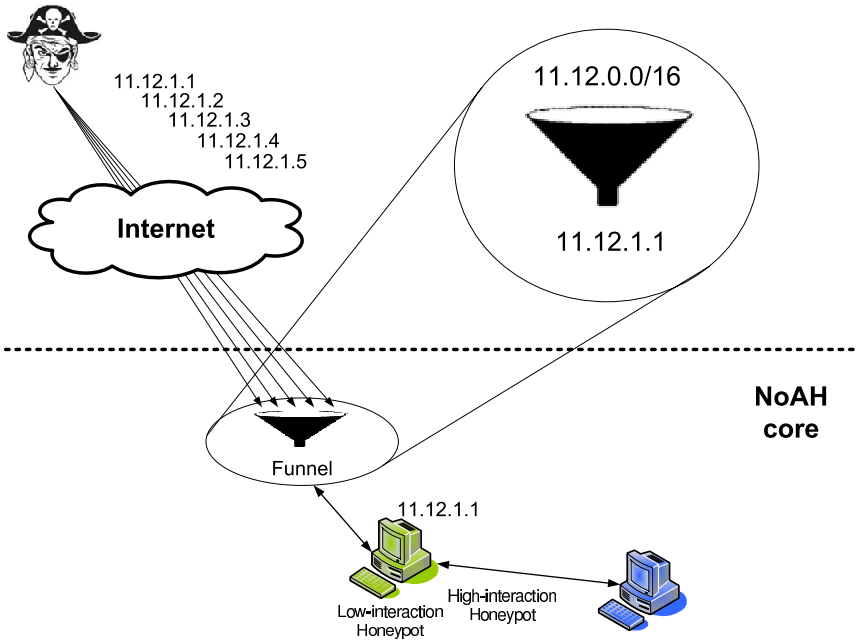


Figure 4.7: Funneling IP addresses from 11.12.1.1 to 11.12.1.5 to a single low-interaction honeypot

Using arpd for funneling

When we want a single honeypot to claim traffic for a few tens of IP addresses, it is impractical and resource consuming to do it with means of configuring its network interface for all the addresses. The solution to this inefficiency is *arpd*. Arpd is a user-space daemon that responds to ARP requests arriving to the network interface of the honeypot. ARP requests are broadcast packets used to discover which machine (more specifically which MAC address) has a specific IP address. For a given IP address in a LAN, the LAN gateway directs the traffic to this IP address to the host that replied to the corresponding ARP request. Under normal circumstances, it is the operating system that takes care of responding to ARP requests. By having arpd reply to these requests, we can effectively direct the traffic for any IPs in the LAN to the host we want, without actually configuring the host network interface for all these addresses.

In our example of Figure 4.7, the arpd running on honeypot will respond to any request for addresses 11.12.1.1 to 11.12.1.5. So, the LAN gateway will forward packets for this range to the honeypot. The packets arriving at the honeypot are handled by the interaction software which responds to the received traffic. Such software is for example *honeyd*. Honeyd can be easily configured to handle traffic for a subnet and respond to it.

We experimented with low-interaction honeypots running honeyd and using arpd for funneling. In our experiments, no overhead was experienced when we tried to emulate a whole /24 or a whole /16 subnet. We artificially tried to stress honeyd by doing requests to the full spectrum of IP addresses it handled. Apart from the initial cost of arpd claiming the IP addresses by responding to ARP requests, response time for requests was the same in any case.

Alternative funneling methods

We should note that arpd only works with low-interaction honeypots. For high-interaction honeypots the applications within the honeypot are bound to specific sockets. In turn, sockets receive traffic only for addresses for which the network interface of the honeypot is configured so the arpd approach does not work. To have a portion of the dark space handled by a single high-interaction honeypot, we can either configure its interface for multiple IP addresses or have a low-interaction honeypot relay specific traffic to it as described in Section 4.4.

Finally, for completeness reasons, we should mention that the second step of funneling can be completely implemented by means of router configuration, that is by mirroring traffic to the port where the honeypot is connected, so there is no need for arpd to run. This approach though is more cumbersome because it involves contacting network administrators every time we need to change the honeypot setup. Additionally, not all routers and switches support traffic mirroring, so this approach will not work in all cases.

4.6.3 Monitoring dark space of cooperating organizations

The honeypots deployed inside the NoAH core monitor a limited, and in most cases contiguous, address space. The more dark address space we monitor, the more accurate is the view of the suspicious traffic we obtain. In our effort to extend the available dark space, organizations (other than NoAH partners) can cooperate with NoAH and offer a portion of their dark space. Honeypot deployment and maintenance is an additional administrative overhead, undesirable to most organizations that do not have the expertise or the resources for such a deployment. Requiring from organizations that wish to cooperate with NoAH to bear this overhead would be impractical.

For this reason, the NoAH architecture proposes redirecting traffic arriving at the dark space of the organization to the NoAH core. By having the traffic redirected and processed in the NoAH core, we avoid requiring the cooperating organization to locally deploy fully fledged honeypots. They will only have to install and run a funnel, which maps all the dark space traffic to a packet forwarding component. This component will be responsible for forwarding packets to and from the NoAH core. Since this forwarding scheme ultimately aims to relieve the cooperating organization from the honeypot related maintenance burden, the forwarding component itself should require minimal configuration and resources to work.

As cooperating organizations are formal entities, like institutes or universities, there is a level of trust between the organization and the NoAH. Thus, there is a minimal guarantee for the normal operation of the traffic redirection and the proper network configuration. Furthermore, we can trust that all traffic coming to the NoAH core from the organization is not artificially generated by the organization itself. Distributed platforms, like Planetlab, can be used as an example.

4.6.4 Tunneling

As described above, cooperating organizations will act as relay agents: they will forward traffic directed to their dark space to the NoAH core and accordingly the responses sent by the NoAH core back to the original sender of the traffic. A possible solution to achieve this would be to perform address rewriting on the packets, so that they are routed to the NoAH core. This solution has the advantage that is simple and that it keeps unchanged the payload of the packets. On the downside, this approach would diminish the dynamics of connections,

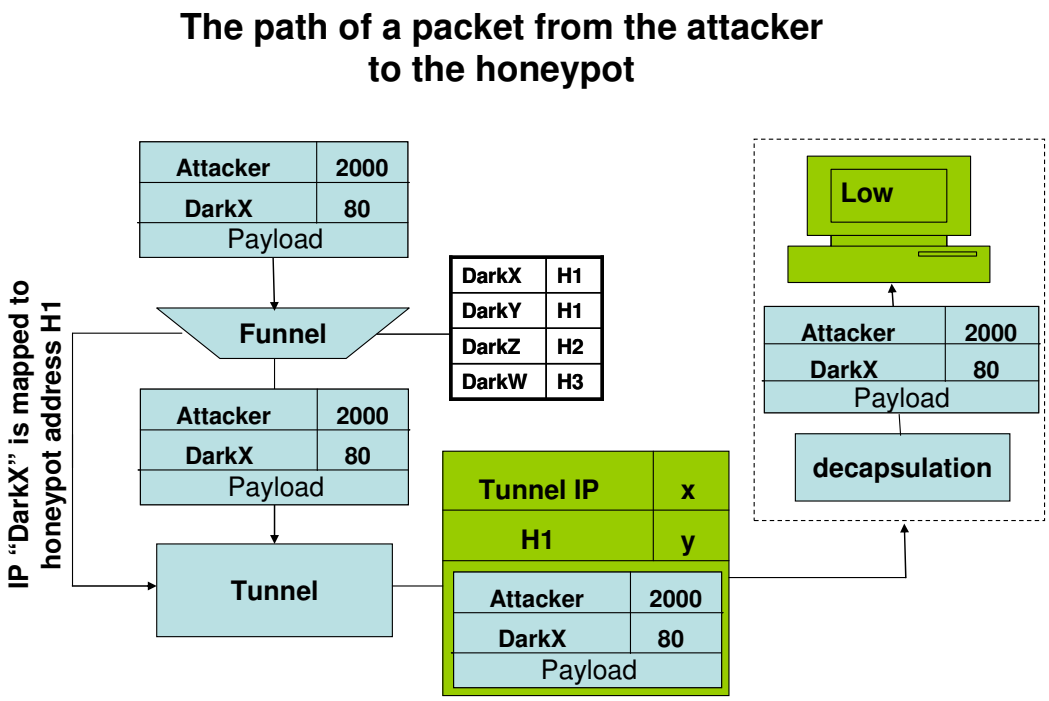


Figure 4.8: Funneling and tunneling. A packet from attacker destined to darkX address is tunneled to honeypot H1.

making detection of certain type of events harder or even impossible. For example, consider an attacker who performs scanning on the dark space of the cooperating organization. Following the address rewriting approach would require to rewrite all destination IP addresses with the address of the honeypot and inject them to the network. However, in the NoAH core we wouldn't be able to detect this scan as we only see traffic going to a single IP address, the honeypot. Another drawback is communications having the IP address as data in a higher level protocol. In this case, rewriting address will not work because addresses will no longer match after translation.

Therefore, the traffic directed to dark space must arrive at the NoAH core unchanged. A more appropriate solution to this problem is tunneling. The concept of tunneling is used mainly on virtual private networks. With tunneling, every packet directed to the dark space of the organization is encapsulated in one or more packets and subsequently sent to the NoAH core. Tunneling preserves the single packets as they are received by the tunnel entry-point (no fragmentation is performed by the tunneling mechanism). At the side of the NoAH core, the original packet is decapsulated and sent to a honeypot. The response of the honeypot follows the reverse procedure: it is encapsulated on the side of NoAH core, decapsulated at the organization and then sent back to the original sender.

The encapsulation/decapsulation procedure, along with the funnel concept, is illustrated in Figure 4.8. The funnel component concentrates packets from a range of the dark space of the organization to a host running the tunnel component. Each tunnel component is configured to forward packets to a single honeypot. Subsequently that honeypot is free to choose whether to locally process the packet or relay it to another honeypot.

So, a packet from the attacker that is destined to IP DarkX in this range will be delivered to the tunnel component, that has the responsibility to forward the packet to the honeypot with IP H1. The tunnel component encapsulates the original packet to a packet destined to IP H1 and sends it to the Internet. As the destination IP of this packet is H1, it will be routed normally from Internet routers and will finally reach the low interaction honeypot at IP H1. There it will be decapsulated and the honeypot will receive the original attack packet unmodified.

In order to implement the funnel, the arpd approach is used similarly to the case of local honeypot deployment so as packets arriving at dark space are mapped to the machine which runs the tunneling software.

Tunneling technologies

A possible solution for implementing the tunnel component would be to use IPSec/GRE[10]. However this solution does not quite match the needs of our architecture. IPSec/GRE is not yet supported by all routers. Also, the IPSec/GRE traffic can be easily identified and it is commonly blocked by intermediate routers[47]. Furthermore, since IPSec/GRE is not yet standardized the process of setting up an IPSec/GRE tunnel changes among different implementations (operating systems). Tools like CIPE [7] do not offer advanced functionality.

For this reason, we turned down the use of IPSec/GRE in favor of OpenVPN[30]. OpenVPN is an open-source, cross-platform, user-space VPN solution. OpenVPN has several advantages over IPSec/GRE. First, the OpenVPN configuration is not specific to the underlying operating system. This greatly eases the deployment of OpenVPN tunnels in heterogeneous environments. Then, OpenVPN traffic is encrypted and cannot be easily identified or blocked. Furthermore OpenVPN is able to tunnel Network Layer 2 traffic as well. This means that the arpd used to implement the funnel component can be moved to the NoAH core side, since the ARP requests/replies will also be tunneled. Having the arpd on the honeypot side is a great advantage, because any administration costs after the tunnel setup will not burden the cooperating organization. Finally, OpenVPN provides the abstraction of a virtual network interface. This guarantees that numerous network-security related programs will be able to transparently run unaltered on the honeypot side.

OpenVPN requirements

OpenVPN requires from the cooperating organization minimal (if any at all) investment in hardware equipment. Actually it can operate on an active workstation without obstructing it's user from his work. If the workstation has only one network adapter OpenVPN can be bridged with it. This way the traffic directed to the unused IP addresses of the workstation's subnet will be tunneled to the NoAH core. This approach can be valuable for the NoAH infrastructure because it makes harder for attackers to use blacklisting to avoid stumbling on a honeypot. Of course cooperating organizations may not feel comfortable with having a NoAH funnel in their live corporate networks. In this case, OpenVPN can still run on an

active workstation that has an additional commodity network interface added. Alternatively, OpenVPN can run without problems on now obsolete hardware.

The software requirements for OpenVPN are minimal too. It has been ported and can operate on all the popular operating systems, including Microsoft Windows family (versions 2000 and later), Linux, Mac OSX etc. What is also important is that the OpenVPN configuration files are simple enough to be automatically generated. This should ease its deployment in the context of the NoAH infrastructure as well as the propagation of possible changes in the configuration.

OpenVPN performance

We measured the overhead added by tunneling. We performed HTTP requests for various file sizes between a client and a server on the same subnet, with and without tunneling. Both client and server were running OpenVPN 2.0 with encryption enabled. Overhead was found to be around 20%, without any noticeable variation. It is expected this overhead to decrease when client and server belong to remote subnets, e.g. client is a broadband user.

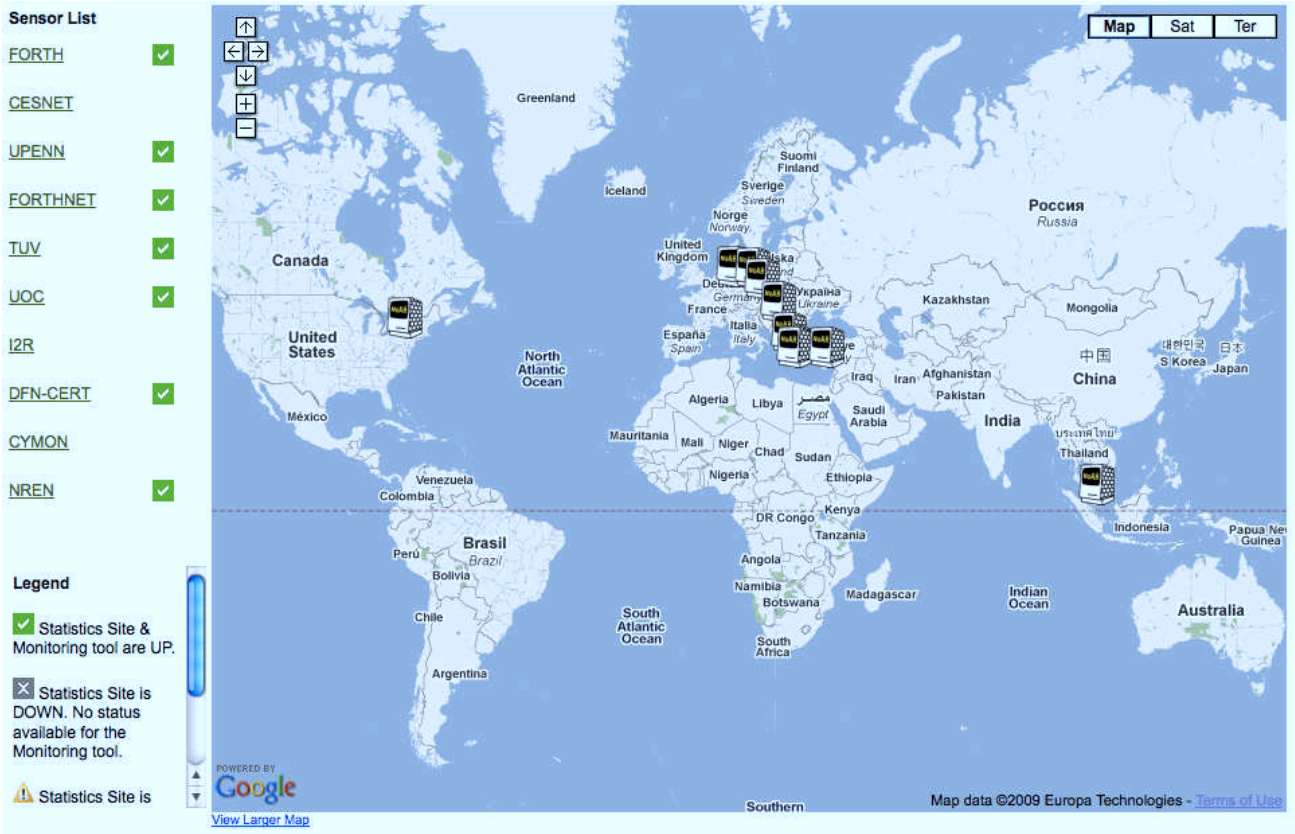


Figure 4.9: The NoAH sensor deployment map

4.7 Pilot deployment and operation of the NoAH infrastructure

The NoAH infrastructure includes so far ten static sensors that monitor more than nine thousand unused IP addresses. The static sensors are geographically distributed and monitor unused addresses from diverse environments; from universities and institutions to ISPs and medical centers. Figure 4.7 displays the location of the deployed NoAH sensors printed on a Google map. In average, the high-interaction honeypots process around half a million packets per day. Such a number is impossibly to be inspected manually by hand, thus automatic mechanisms that will display statistics and trends about received traffic is needed. We present what types of statistics are gathered by each sensors and how they are visualized.

4.7.1 Sensor statistics

Each sensor runs three software components. The first one is a minimal daemon based on the pcap library[127] that listens to an interface and captures packets going to a given unused

Source IP Addresses

Source IP	Packet Count	Country
91.41.222.13 (?)	2269	
174.37.228.48 (?)	1017	
116.53.68.74 (?)	653	
70.44.1.34 (?)	606	
93.190.137.241 (?)	569	
89.248.172.196 (?)	412	
88.80.5.157 (?)	235	
64.18.128.86 (?)	156	
121.12.109.36 (?)	122	
111.92.244.76 (?)	116	?

Destination TCP/UDP Ports

Destination Port	Packet Count	Trend
445 (?)	48794	▲
135 (?)	2272	▲
1434 (?)	1092	▲
3072 (?)	824	▼
1024 (?)	788	▼
2967 (?)	653	▲
23 (?)	606	▲
1720 (?)	569	▲
5900 (?)	148	▼
137 (?)	107	▲

Figure 4.10: The top 10 source IP addresses and destination ports as monitored by a NoAH sensor for one day

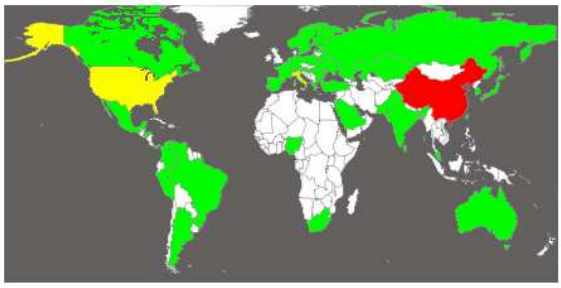


Figure 4.11: The geographical distribution of attackers as monitored by a NoAH sensor for one day

IP address space. Specific pieces of information for the captured packets are stored to a local Postgres database - the second software component -. This information includes the packet protocol, source and destination IP addresses, source and destination ports, flags in the case of a TCP packet and finally the timestamp of when the packet was captured. The last component is a set of PHP files that retrieve and render various statistics for the traffic received. These statistics are:

- *Top source IP addresses.* By default the top 10 source IP addresses that sent most packets for the last 2 hours is displayed. For each IP address the number of packets it sent and its geographic location is also displayed. The geographic location is retrieved by a local MaxMind database. Additionally, each IP address is clickable. By clicking it, user is redirected to a web page that displays all packets sent by that IP address for a configurable time period. User is able to select that time period which varies from two hours up to the last month.

- *Top destination ports.* The top destination ports targeted by attackers is displayed. For each port the number of packets and a trend indication is also shown. The trend indication represents whether the sensor received more or less packets to that port in comparison with the previous time period. Again, the user can configure the time period up to the last month. By clicking a port, a web page containing all traffic sent to that port is sent. A screenshot for the top 10 source IP addresses and destination ports as observed by a NoAH for one day can be seen in Figure 4.10.

- *Attack maps.* This page includes two earth maps. The first map displays the geographic distribution of distinct source IP addresses. Each country is colored based on how many IP addresses are hosted in that country. Countries that host no attackers are colored as white, low activity countries are colored as green while countries that host lots of attacking IP addresses are red. An example of such a map is displayed in Figure 4.11. The second map looks alike the first one but is based on the number of packets sent by each country. The scale of both maps is calculated dynamically based on the traffic volume. Maps are generated once per day as they require significant processing power and it is not feasible to create them on demand.

- *Attack graphs.* This page includes three graphs. The first one is a breakdown of the TCP ports while the second one is a breakdown of UDP ports for the last two hours.

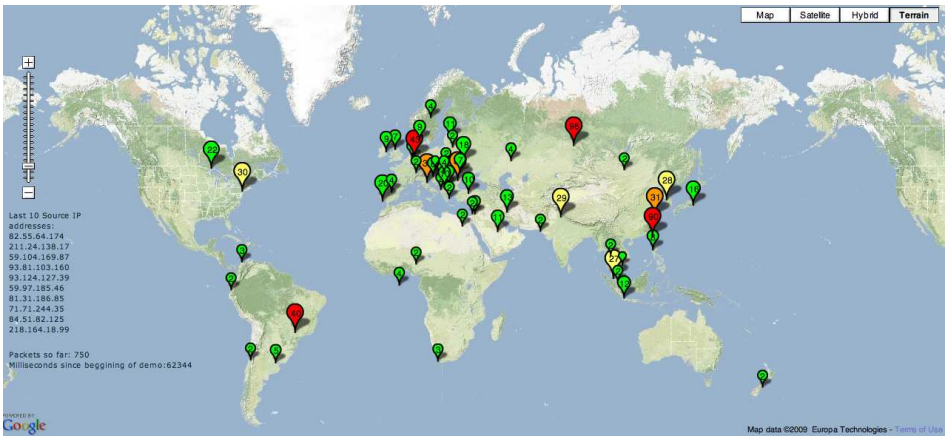


Figure 4.12: Screenshot from TrGeo, a Flash application that displays the geographic origin of attackers

The third one is a breakdown of traffic type in terms of how much TCP, UDP and ICMP traffic was received during the last day.

- *Backscatter traffic.* Each sensor receives unsolicited traffic, that is traffic that comes in response to spoofed attacks. It is trivial to identify such traffic by inspecting the TCP flags of each packet. A plot for the number of backscatter packets received over the last week is displayed on a separate page.

Furthermore, the user has several options to customize the displayed information. First of all, she can select traffic coming from or going to a specific IP address. Second, the time frame can be changed. The user can view traffic from the last 2 hours up to the last 1 month. Additionally, she can define an arbitrary date interval. Third, the number of top source IP addresses and top destination ports can be altered. By default, only 10 of the most active source IP addresses and targeted destination ports are displayed. These lists can be expanded up to 100 entries. The final filtering option is to display traffic going to specific destination ports. All of the above options can be combined. Thus, user can query our statistics database for more complicated questions, like "what traffic have you received that originates from IP address X, targets IP address Y to destination port Z from last Monday to last Thursday?" and many more.

4.7.2 Data aggregation

All individual sensors send daily a list of the top 100 source IP addresses and top 100 destination ports they observed to the main stats.fp6-noah.org site. The lists are then aggregated and displayed in a similar template to the one of individual sensors. The role of aggregation is twofold. First, it permits us to view at a quick glance what traffic is sent to the honeypot sensors and see if trends have changed over the last monitoring periods. Second, it allows us to correlate traffic among various sensors. For example, we can check if two sensors receive traffic on a specific port and if their most attacked ports are the same. As all aggregated data are public, the source IP addresses are anonymized after the aggregation process. However, the geolocation of each source IP address is calculated before the anonymization

process. The anonymization function applied to source IP address is the replacement of the address with random numbers. We have also considered other anonymization functions, such as prefix-preserving, but they present a high risk of revealing the honeypot topology.

In our effort to present an overview of what traffic our honeypots capture daily, we have implemented TrGeo. TrGeo is a platform for geographic visualization of packets captured by the NoAH infrastructure. The basic concept behind TrGeo is to track locations of the attackers and display the traffic volume they send on the earth map. For the purposes of this work, we have implemented TrGeo as an Adobe Flash application that renders desired data on top of Google maps. On each source location a balloon is drawn that represents how much traffic the location sent in terms of packets. As time passes, the height of these bars changes according to the traffic they sent. In fact, TrGeo implements the visualization of a sliding time window. For example, if it has not observed packets from a location for a long time period, the balloon for that location will have its counter and size decreased. The information about packet volume and geographical origin is extracted from queries done to the stats.fp6-noah.org site. The aggregation is done at the level of countries, this means multiple attackers from different cities of a country are mapped to a single random location within the country. A screenshot of TrGeo is shown in Figure 4.12.

4.7.3 Sensor monitoring

To monitor the availability of NoAH sensors, a web page that shows the sensors drawn on a Google map was constructed. A screenshot can be seen in Figure 4.7. Each NoAH logo represents a sensor. On the left of a map, a list of the sensors is displayed. Next to the name of each sensor there is a status icon. A green tick means that the sensor is up and displays statistics. A yellow exclamation mark means that the server is up but no statistics are displayed in its pages, indicating possibly the collection component is down. A red cross icon means that the server is either down or not accessible. The status of each sensor is inspected at the background by an external script. Furthermore, the script is linked to an alerting module that automatically sends notification e-mail when a sensor is down for more than 2 hours. All sensor monitoring pages are password protected and accessible by few IP addresses. This security measure ensures the confidentiality of sensor owners.

4.7.4 Conversation and malware statistics

The aim of this Section is to provide an overview of the statistics collected by the NoAH infrastructure during the last two months of the NoAH deployment (July to September 2009). These statistics are for monitoring an unused IP address space of around two and a half thousand IP addresses.

The distribution of the conversations handled by the NoAH honeypots are shown in Figure 4.13. By conversations we mean the connections that were established with the honeypots and sent application data that could be characterized as exploitation attempts. During a 2 month deployment period, from the end of July to September 2009, the infrastructure handled a total of 153,082 conversations with attackers that targeted the NoAH sensors. The maximum number of conversations handled in one day was 22,268 which occurred on the 17th of September.

Table 4.1 presents the top 10 source countries of attackers that initiated conversations with the NoAH sensors. The aggregated conversations from these 10 countries amount to

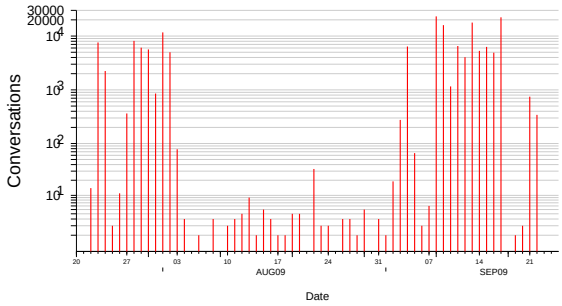


Figure 4.13: Conversations with attackers

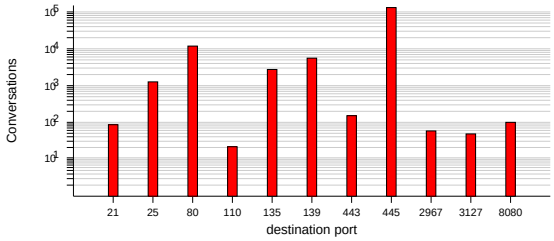


Figure 4.14: Top 10 targeted destination ports

Country	# Conversations
USA	56,361
Russia	22,823
Taiwan	18,858
AR	9,327
Japan	6,662
MY	4,193
MD	3,474
PT	3,061
BG	2,663
China	2,646

Table 4.1: Top 10 source countries of attackers that targeted the NoAH sensors

IP Address	# Conversations	Country	Days
99.14.215.83	14,499	USA	1
173.24.92.207	7,862	USA	2
89.178.247.14	7,279	Russia	1
63.151.109.189	7,260	USA	2
114.137.101.25	5,327	Taiwan	2
122.125.98.156	4,853	Taiwan	1
190.50.56.118	3,798	Argentina	1
96.243.101.211	3,743	USA	3
190.176.42.29	3,628	Argentina	1
12.183.171.197	3,580	USA	2

Table 4.2: Top 10 attackers that targeted the NoAH sensors.

84.8% of the total conversations handled by our honeypots. Results show that the United States are the country that initiated the largest number of conversations with the NoAH sensors. During this period the number of attacks received from Greece are negligible. In Table 4.2 we can see the statistics of the top attackers for the whole duration of the two month deployment period. It is interesting to note that all top attackers were active for a few days, and in all cases they were consecutive. In Figure 4.14 we can see which ports were targeted the most by attackers. Port 445 was again the most targeted port receiving over 130,000 conversations initiated by attackers. The second most popular port is again port 80 which attracted over 11 thousand conversations. We can identify two other interesting ports. Attack traffic targeting port 2967 exploits a vulnerability in the Symantec antivirus. Attacks with a destination port of 3127 are very popular amongst worms.

In Figure 4.15 we can see all malware binaries collected during the 2 month deployment period. The green portion of the bars indicate the number of malware samples received on that day, that had not been collected before. Overall, 60 samples of malicious software were downloaded based on information extracted from the incoming attack traffic, 36 of which had not been collected before. The largest number of unique samples downloaded in one day, was on the 23rd of July when 5 different binaries were downloaded, while the day with largest number of malware samples overall was on the 29th of August with 9 samples totally.

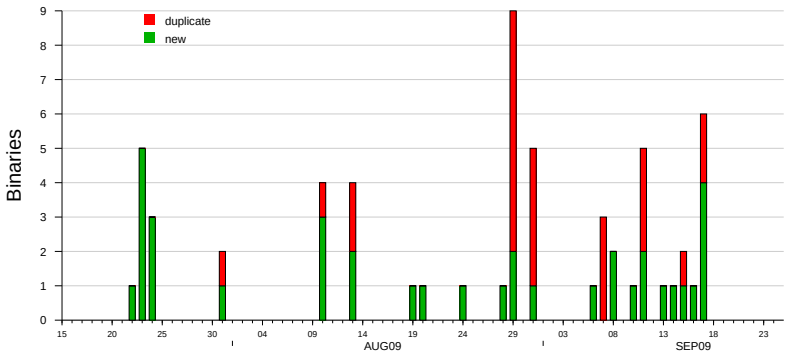


Figure 4.15: Binaries collected by the NoAH infrastructure

Chapter 5

The Honey@home approach

Honeypots have been shown to be very useful for accurately detecting attacks, including zero-day threats, at a reasonable cost and without false positives, unlike IDS's and ADS's. However, there are two pressing problems with existing approaches. First, the effectiveness of honeypots heavily depends on the unused IP address space they cover. Unused IP address space can be found in almost every organization, institution and public body due to under-utilized or even totally empty subnets. However, the deployment of honeypots requires both administrative expertise and dedicated resources that many organizations cannot afford. The second problem is that attackers are evolving, and it has been shown that it is not difficult for them to identify honeypots and develop blacklists to avoid them when launching an attack. Although there are approaches to harden the identification of honeypots, recent work has shown that it is relatively straightforward for attackers to detect the placement of certain types of sensors [75, 152].

In response to these problems, we propose a new architecture that enables large-scale deployment at low cost, while making it harder for attackers to maintain accurate blacklists. The Honey@home architecture relies on communities of regular users installing a lightweight honeypot that monitors unused addresses. Because it does not require the static allocation of valuable chunks of network address space, and considering the success of other community-based approaches such as seti@home, our approach is well-suited for creating a large-scale honeypot infrastructure at low cost. Since participation in the system is dynamic as users come and go, it becomes harder for attackers to maintain accurate blacklists. Users only need to install a lightweight daemon that runs in the background and is responsible for grabbing an unused IP address, forwarding the traffic of that space to a honeypot core and injecting the responses coming from that core in order to respond to the attackers.

In the following Sections we discuss the current design of the Honey@home architecture, a preliminary implementation, the design issues that we faced especially with respect to infrastructure robustness, and discuss detectability and effectiveness issues.

5.1 Honey@home design

Honey@home is designed simple and lightweight, as it mainly targets on typical home users or administrators unfamiliar with honeypot technologies. In a nutshell, Honey@home forwards traffic to unused IP addresses or unused ports to a honeypot farm and sends the answers provided by the honeypots back to the attacker. It is a software package that needs no

special configuration to run and can run on both Unix and Windows platforms. It is also non-intrusive as it runs in the background with minimal CPU, memory and network overhead. Our measurements have shown that Honey@home client requires less than 2% CPU utilization and nearly 10MB of main memory for Windows versions. Similar requirements apply for the Linux version as well.

5.1.1 Design requirements

Honey@home is a software client that is accessible by everyone, including malicious users. With that in mind, we need to fulfill three major requirements:

- The location of honeypots must remain hidden. If honeypots become known, attacker can try to manually compromise them, evade their detection mechanisms or flood them with junk traffic, forcing them to waste their time on useless traffic instead of serving Honey@home clients.

- The identity of Honey@home clients must also remain hidden. Once it is known, the attacker may blacklist them and make them "blind" during the time of a real attack outbreak. Honey@home should be undetectable – or at least very hard to detect – by attackers. We discuss about detectability issues in Section 5.3.

- Attackers must be prevented from automatically installing Honey@home in the machines they own. In case that the attackers own a botnet, they must be forced to setup Honey@home manually to each one of the bots.

These challenges make Honey@home a more complex architecture than a simple packet forwarder. We address all of the above issues in Sections 5.2 and 5.3. Before we proceed to the challenges, we describe the architecture thoroughly at Section 5.1.2. Apart from the three main requirements, we also need to address the issue that malicious traffic must not be able to infect any part of the architecture, namely Honey@home clients, honeypots and intermediate components.

5.1.2 Core architecture

Every Honey@home client is responsible for a single unused IP address (unused IP addresses are also referred as *dark*) or the unused port space of the machine it is installed on. All the traffic received by the client is tunneled to the centralized honeypots of the NoAH core through the Tor anonymization network[92]. The Tor network provides all the desired anonymity for both Honey@home users and honeypots. Details about Tor are provided in Section 5.2. Responses coming from the core are injected by Honey@home to the network so as to reach the originators of the traffic. The architecture of Honey@home is represented in Figure 5.1. Honey@home clients are connected through an SSL connection to the *SSL server* component. The SSL connection is passing through the TOR anonymization network. The server component handles the client connections and it is responsible for validating users and forwarding their packets to honeypots. Users are validated by supplying a key to the SSL server. They key is obtained after the user registers at the official website of Honey@home. All registration information and user keys are stored in a MySQL database. After the user is validated, her packets are sent to honeypots and responses from honeypots are sent back to the user.

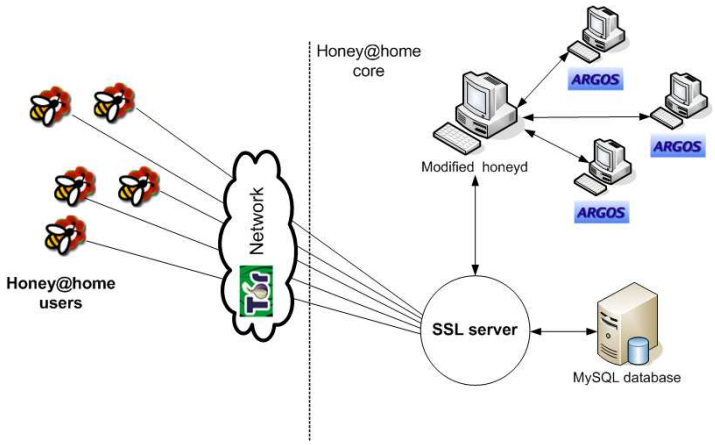


Figure 5.1: The Honey@home architecture

All traffic exchanged between Honey@home clients and honeypots is recorded to a central database for further analysis and visualization purposes.

We run both low- and high-interaction honeypots to handle user's packets. *Honeyd*[142] is used as low-interaction honeypot. Honeyd is a very popular and lightweight system with many interesting properties, such as network stack emulation. We use honeyd as a mechanism to filter out uninteresting traffic, such as TCP connections that do not complete the three-way handshake or attacks that can be easily emulated, for example SSH brute-force attacks. All the other traffic is forwarded to high-interaction honeypots. The forwarding is performed by a hand-off mechanism[69] we implemented inside honeyd. Honeyd creates a new connection with the high-interaction honeypot and sends all the application content it receives. The handoff is based on the destination port. For example, if an attacker wants to connect to port 445 or 139, the connection is forwarded to a high-interaction honeypot emulating the Windows XP operating system. As the choice of high-interaction honeypot is static, this means we may loose attacks for services that can run on multiple platforms. Examples of such services are web servers (Linux Apache or Windows IIS?), SMB sharing (Linux SAMBA or Windows sharing?) and many more. However, our choice is currently made based on popularity of applications in terms of users and attack instances. For our prototype system, we emulate a limited number of services and applications and more specific Linux telnet daemon, Microsoft Internet Information Server, MS-SQL server and the default Windows services. We intentionally run unpatched versions of applications and services so we can observe attacks taking place. [1]

We chose Argos emulator[141] for high-interaction honeypot. Argos is using memory-tainting techniques and is able to track both known and unknown exploits. Argos is based on the idea that code coming from the network should never be executed. Once data from the network are treated as executable code, Argos raises an alert containing all relevant information about the attack and vulnerable application is restarted. As only the vulnerable application is restarted and not the whole virtual machine, the downtime of the vulnerable application is minimal. As Argos detects the attack before its code is executed, we have a core that is hard to infect. The only way to infect an Argos honeypot is to trigger an exploit on the underlying Qemu emulator [34] but this issue is beyond the scope of this thesis. In

[1] We plan to run fully patched versions in our production system so as to capture only fresh attacks.

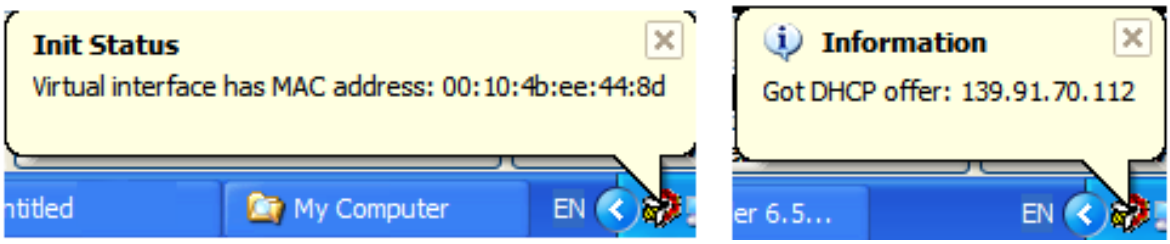


Figure 5.2: Honey@home client in action: Creating an pseudo-interface (left) and getting an IP address through DHCP server (right)

contrast with other approaches, like Honeynets[15], that use bandwidth control and extrusion detection to mitigate effects of infection, our core is able to run with minor administration overhead. To improve the scalability of our architecture, we run multiple Argos systems inside the NoAH core, each one being responsible for specific applications. As Argos has inherently slow performance ,around 30x slowdown of applications, we need multiple instances in order to serve all clients. We also consider using traditional load balancing techniques to share load for the same target applications (like in Web server farms) and offload Argos by using low-interaction honeypots as filtering mechanisms, such as honeyd. Honeyd is not a bottleneck as it can serve thousands of TCP requests per second, around 2000 requests according to [142].

5.1.3 Unused IP address space monitoring

Each Honey@home client is requesting an IP address from the local DHCP server (optionally it can be set to listen to a static IP address). Most broadband connection routers, like ADSL routers, organizations and institutes use DHCP servers to assign addresses. Every time Honey@home client starts, it requests an address from the local DHCP server. The main advantage of this approach is that user does not need to statically set an IP address, which may be even hard for him to find. Honey@home client can be stopped or started any time without interrupting the normal operation of user's network. Upon the client exit, the clients informs the DHCP server that the IP address is released. The procedure of how a Honey@home obtains an IP address is shown in Figure 5.1.3. The client first creates a pseudo-interface with a random MAC address and broadcasts a DHCP request (left part of the Figure). When the DHCP response is received, client configures itself to wait for packets destined for the given address, 139.91.70.112 in our example (right part of the Figure).

The Honey@home client can be configured to claim an IP address statically or by using a BPF filter. Static allocation and BPF filter features are mainly targeted for more advanced users. For example, an administrator can setup a single Honey@home client in an unused subnet and set the BPF filter to cover all the addresses of that subnet. In that way, the unused subnet becomes useful in a few steps. As long as the client runs, the subnet monitored by the Honey@home client will contribute to the overall infrastructure.

As a short notice, Honey@home clients also receive legitimate traffic, such as broadcast ping or SMB queries. This traffic has to be white-listed and the decision can be made either locally or at the core. The Honey@home client can be configured not to forward traffic sent to specified ports. Additionally, Honey@home client can be configured to forward traffic sent only to a specific set of ports. By default, all traffic destined to the unused address claimed

by Honey@home is sent to core. As Honey@home captures traffic directed to an unused IP address and not the packets destined for the actual IP address of the host running the tool, there are no privacy concerns. All traffic destined to unused addresses is by default suspicious. The only legitimate traffic that is destined for unused addresses, according to our traffic traces, is attempts to connect to peer-to-peer ports. As peer-to-peer programs tend to have some sort of memory, like host caches of Gnutella, it is observed sometimes that some external hosts try to reconnect to an address that was used in the past, participated in a peer-to-peer network, but is not used anymore and thus claimed by Honey@home. If the user is concerned about privacy issues, she can configure the client not to forward traffic directed to these ports.

Honey@home can work even behind NAT. Hosts behind NAT cannot accept incoming connections from external hosts, only for ports that are explicitly forwarded to them through the router setup. For this case, Honey@home clients can automatically configure the local router to forward specific ports to the physical machine on which client is running using the UPnP protocol[43]. UPnP provides an API to configure the router to forward packets for specific address and ports and is supported by the majority of routers. However, modern routers have UPnP disabled by default for security issues as malware can also use it and make infected machines act like servers. For those users that are not privileged to change the router configuration, the Honey@home client is limited to capture suspicious traffic that is generated by internal infected hosts, for example local scans.

5.1.4 Unused port monitoring

Regular home users can usually claim one IP address. However, they can also contribute to the honeypot infrastructure by offering their unused port space. Ordinary users usually run a few applications that listen to specific ports. These applications include messengers that bind a port for incoming file transfers and peer-to-peer clients. It is, thus, rare to find typical users that run a web server to their machine or an SMTP server.

Based on this observation and given that already infected hosts keep searching for new victims, it is expected that we may see connections destined to unused ports of the user. Honey@home client can be configured to listen to all unused port space (default) or to a set of ports specified by the user. It examines all incoming traffic (through the *pcap* library) and searches for connection attempts on the unused port space. If such an attempt is found then the traffic going to this port is forwarded to the core. Periodically, Honey@home client scans the host to obtain a list of used ports, similar to the way the netstat tool does, and update the list of unused ones.

The major concern around unused port monitoring is possible deniability. As an example, a user was running a peer-to-peer application for several hours ago and then disconnected from the peer-to-peer network. However, as such systems tend to have a sort of "memory", incoming connections to the ports of his peer-to-peer application may still be arriving. Honey@home client sees that ports previously used by peer-to-peer applications are unused and forwards their traffic despite the fact that this traffic is legitimate. We are currently investigating these issues and we are experimenting on hold-on timers. Hold-on timers remember that a port was used k hours ago and even if it is currently unused, it is not forwarded. Another solution is port sampling, that is forward traffic from only a few unused ports. By default, Honey@home clients do not forward traffic destined to known sensitive ports, like Gnutella and eMule.

5.2 Dealing with challenges

In this Section, we provide details on how challenges described in 5.1.1 can be dealt with and specifically how we can hide honeypots (Section 5.2.1) and prevent automatic installation of Honey@home clients (Section 5.2.2). Detectability issues of Honey@home clients will be discussed in Section 5.3.

5.2.1 Hiding honeypots

Hiding honeypots is essential for the viability of the architecture. The reason is twofold. First, we want to protect the core honeypots from immediate exposure. Although, we cannot protect them from Denial-of-Service attacks (the attacker can easily do DoS on the anonymization network), we make hard for the attacker to manually compromise our honeypots or use techniques that evade the deployed detection mechanisms. For Denial-of-Service attacks, the only measure we could take is not to accept clients that send data above a certain rate. By default, each Honey@home clients sends a few kilobytes per minute and it is easy to detect misbehaving clients. Second, we envision Honey@home as an architecture that will act as a feeding mechanism for other systems as well. Currently, our prototype implementation is based on honeyd and Argos system but our architecture is flexible enough so other systems can plug in as well. A direct exposure of honeypots to users would possibly provide hints for detection mechanisms used in the core.

Anonymous routing

Before we proceed on the approach we use for hiding honeypots, a brief introduction to anonymous routing is provided. The goal of anonymous routing is to protect the privacy of both the sender and the recipient of a message, providing protection for eavesdropping in parallel. Messages are transferred from sender to recipient through an anonymization network and none in the middle of the path can identify the role of each participant (sender, receiver or forwarder).

The anonymization scheme we selected is onion routing, a technique based on a set of dedicated servers that route packets anonymously. Clients simply participate in the network without having to share their resources or help the network route packets. Onion routing uses dedicated software-based routers, called onion routers, that perform the anonymization process. We have also considered peer-to-peer anonymization schemes but they can lead to immediate exposure of Honey@home clients as they become the entities who form the anonymization network. Additionally, to the best of our knowledge, there are no widely deployed peer-to-peer anonymization networks.

The advantage of onion routing is that it is not necessary to trust each cooperating router. The concept behind onion routing is the routing onion. Routing onions are used to create paths through which many messages can be transmitted. In order to create a path, the client at the head of a transmission randomly selects a number of onion routers and generates a message for each one, providing them with symmetric keys for decrypting messages, and instructing them which router will be next in the path. Each of these messages, and the messages intended for subsequent routers, is encrypted with the corresponding router's public key. This provides a layered structure, in which it is necessary to decrypt all outer layers of the onion in order to reach an inner layer. As each router receives the message, it "peels" a layer

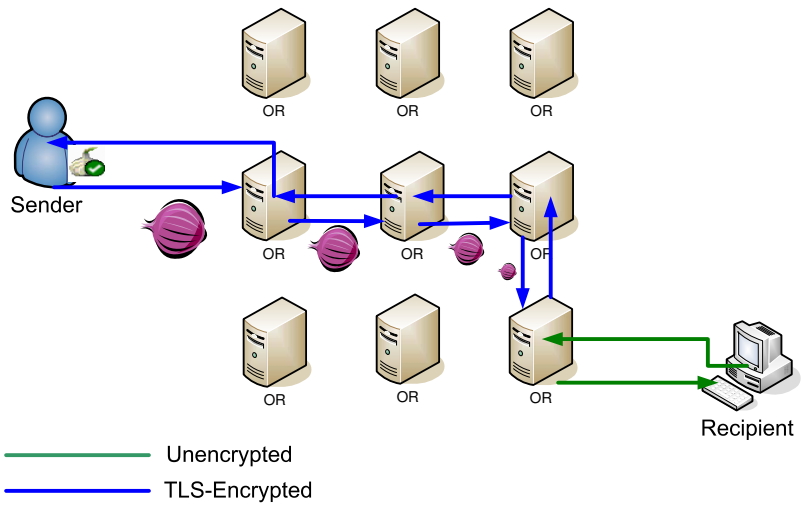


Figure 5.3: An overview of how Tor works. Client establishes a path of onion routers and sends onions, messages encrypted with public keys of all path's routers. At each router the onion is piled off -decrypted by router's public key- and forwarded to the next router. The last router has fully decrypted content and communicates directly with recipient through a standard TCP/IP connection.

off of the onion by decrypting with its private key, thus revealing the routing instructions meant for that router, along with the encrypted instructions for all of the routers located farther down the path. Due to this arrangement, the full content of an onion can only be revealed if it is transmitted to every router in the path in the order specified by the layering.

Tor is the most popular and widely used implementation of onion routing and is used for anonymizing any application that uses the TCP protocol. The Tor network has several thousands users and around 400 onion routers are deployed. Figure 5.3 illustrates how Tor works. At the sender side, user installs a SOCKS proxy that is used by the applications. This specialized proxy connects to Tor network to create an onion routing path. All communication between sender and onion routers and among routers is done using the TLS protocol (blue line). At the server side no deployment is needed. The last router of the path communicates with the recipient using the standard TCP/IP protocol (green line). Sender sends onions which are piled off along the path. The last router receives an onion with non-encrypted content and forwards it to the recipient. The response of the recipient follows the same path backwards.

Using Hidden Services

The way Tor was described earlier assumes that every sender knows the address of the recipient. This is not the case for our system, where address of honeypots must remain hidden. Tor offers a functionality called "hidden services", that permits the recipient to hide its address. Hidden services work as follows. Initially , the recipient gets a descriptor for its hidden service from a centralized service lookup server. This descriptor is a DNS-like name, in the form of "xyz.onion", where the .onion domain can be resolved only inside the TOR network. After-

ward, it creates onion paths to several introduction points. An introduction point can be any onion router. It then advertises the descriptors of introduction points and addresses to service lookup repository. The client only needs to know the service descriptor. For example, when a client browses a hidden web server it types to her browser a URL like "http://xyz.onion", where xyz.onion is the service descriptor. When clients lookups for xyz.onion at the directory server, a set of introduction points is returned. The client creates an onion path to a "rendezvous point" and requests one of the introduction points to connect to a server. The introduction point passes the request to the server (remember that introduction point and server are connected through an onion path) along with the address of the rendezvous point. If the server wants to connect to the client, it creates a path to the rendezvous point and then the point mates the two paths. We have implemented Honey@home core as a hidden service. Honey@home clients only know the service descriptor of the core and will connect to it using a "xyz.onion" name. We use two safety features to protect our honeypots from attacks against TOR. First, the entry TOR node from honeypots to the introductory point is a trusted node, maintained by Honey@home developers. This measure protects our infrastructure from Sybil attacks, where attackers users flood the TOR network with malicious servers so they can control the entry and exit nodes of TOR users. Second, we use SSL connection over the TOR network to ensure that even if the trusted router of the path is compromised, it cannot inspect the contents of the communication or understand that is a Honey@home client-Honey@home core communication.

5.2.2 Preventing automatic installations

In the case that an attacker owns a botnet, she can automatically install Honey@home clients in all bots if no measure for preventing automatic installation is taken. Massive deployment of Honey@home client to a botnet will cause deniability issues at the core. While the core does not suffer from false positive problems (specially crafted traffic that can trigger false alerts), its capacity in processing power is limited. To prevent attackers from massive installations, Honey@home clients are verified to the core by providing a registration key. The registration can be done at the official site of Honey@home. We employ Enhanced CAPTCHAs[65] techniques for the registration process. Additionally, two or more users with the same registration key cannot be connected with the core at the same time.

5.3 Detectability of Honey@home clients

Honey@home clients are part of a distributed honeypot infrastructure and although they do not present any special functionality themselves, they are considered as detectors from an attacker point of view. Detectability of honeypots is an open issue and has been studied partially in [94, 2, 93] but those techniques focus more on detecting the underlying software system, like honeyd or Sebek. In this work we do not study how to secure the core honeypots from exposure but we try to identify how an attacker can understand if a host is running the Honey@home client.

A Honey@home client relies on the Tor network to forward its packets to the core honeypots. The attack scenario is as follows. An attacker wants to identify whether some hosts in a given subnet are running Honey@home. To do so, she performs a TCP scan for specific ports at this subnet. Some hosts will respond in time $t_{respond}$. Honey@home clients will need additional time t_{tunnel} to forward the packet through Tor, wait for the response and inject

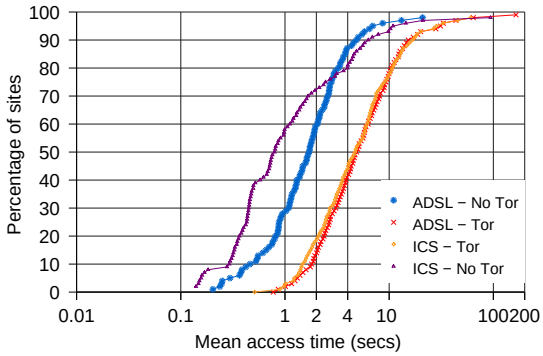


Figure 5.4: Mean access time for retrieving the index file of top500 sites with and without Tor

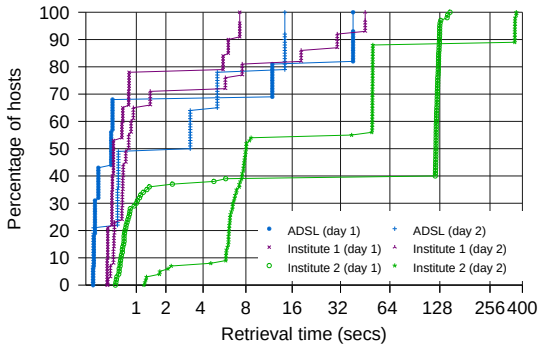


Figure 5.5: Cumulative distribution function of retrieval time

it back to the attacker. If t_{tunnel} is much larger than $t_{respond}$ then it is an indication that Honey@home is running. Time t_{tunnel} is the response time between Honey@home client and the core honeypots multiplied by a delay factor due to the Tor network. We measured the delay introduced by Tor in terms of time needed to browse several popular web pages (only their main index file). In our experiment, we downloaded the index file of 500 web pages with and without TOR from two different environments: our institute's 1GBps line and an ADSL connection of 1MBps. The results are summarized in Figure 5.4. For 90% of the cases, we needed up to 4 seconds to retrieve the index file without Tor for both environments and up to 15 seconds in the presence of Tor. This gives us a slow factor of 3 to 4 when using Tor thus making the time t_{tunnel} noticeable.

Honey@home clients are designed to run by normal users. As we cannot reduce the delay of forwarding packets (even the absence of Tor would still yield a significant overhead), we rely on users' behavior to hide Honey@home clients. We performed a scan to port 80 in ten different subnets. Two of them belong to institutes, one in U.S. and one in Singapore, and the rest belong to ADSL users of two major Greek ISPs. For those hosts that had port 80 open, we measured the time to retrieve the index file. We performed the same experiment for the next 2 days after the initial measurement and took again the times. The results are summarized in Figure 5.5. To avoid making the figure crowded, we plotted the results from the first two days. The results for ADSL subnets are merged and plotted in a single line. We made three interesting observations. First, the hosts that were found to have port 80 open differ slightly from day to day. Some hosts never reappear with port open and some new hosts respond to the port. In fact, only 7% of the hosts were persistently responding to the port all the 3 days. Second, hosts that persistently respond to port 80 present a variation in their response time. Third, a significant percentage of hosts respond in time greater than 5 seconds, with some hosts needing even more than 30 seconds. This delay response allows us to hide Honey@home clients as it is unclear for attacker if it is a host responding very slow or a Honey@home node that forwards traffic to core honeypots.

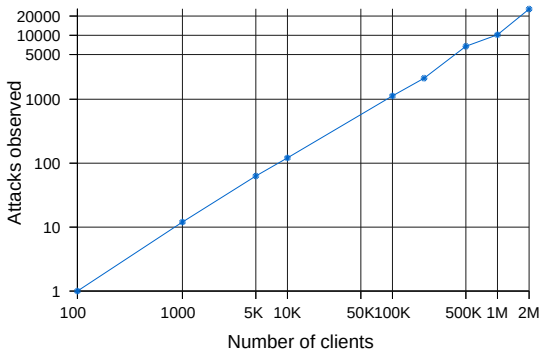


Figure 5.6: Instances of an attack observed as a function of Honey@home population

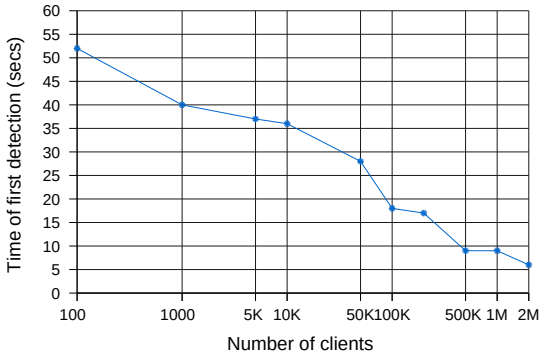


Figure 5.7: Time needed to detect first instance of an attack as a function of Honey@home population

5.4 Simulations

As Honey@home is not yet largely deployed, we performed simulation to measure its effectiveness in terms of detection rate and speed. Our simulated scenario involved the spread of a worm, where each infected hosts scans 10 other hosts per second and the vulnerable population is 2% of total IP address space (around 80 million hosts). We assumed that infected hosts do not scan at the loopback (127.0.0.0/8) and NAT subnets (192.168/16, 172.16/12 and 10/8 as defined in [1]) and the initial number of infected hosts was arbitrarily set to 10. Infected hosts were doing random scanning without any topological information included in the scanning process. We consider that the population of Honey@home clients remains unchanged throughout the one simulated minute.

Our initial measurement counted the number of worm instances observed as a function of Honey@home population within the first minute of the worm spread. Results are summarized in Figure 5.6. As the number of Honey@home clients increases, the number of observed attacks is also increased in a linear fashion. For example, with 10,000 clients we can observe 100 instances of the attack, while we need more than 100,000 clients to have 1000 samples. Taking into consideration that attacks may have polymorphic or metamorphic properties, we need multiple observations to characterize the attack or even generate signatures for it.

Our second measurement focused on the detection delay of the first worm instance, that is the time needed to see at least one attack instance in one of the deployed Honey@home clients. The results are displayed in Figure 5.7. Linearity does not apply for the detection delay. Doubling the number of clients does not necessarily mean that we will see the first instance of the attack in half time. With 100 clients, we need around 52 secs to observe the first instance, while to see it in half time (nearly 25secs) we need around 50k clients. In order to have a detection time of less than 10 seconds we need more than a half million clients. We would like to note here that the term client does not directly refer to a person that will install Honey@home. We use the term client here to denote an unused IP address. As Honey@home is designed to monitor arbitrarily large unused IP address space, the number of actual installations may be significantly less than the number of monitored addresses.

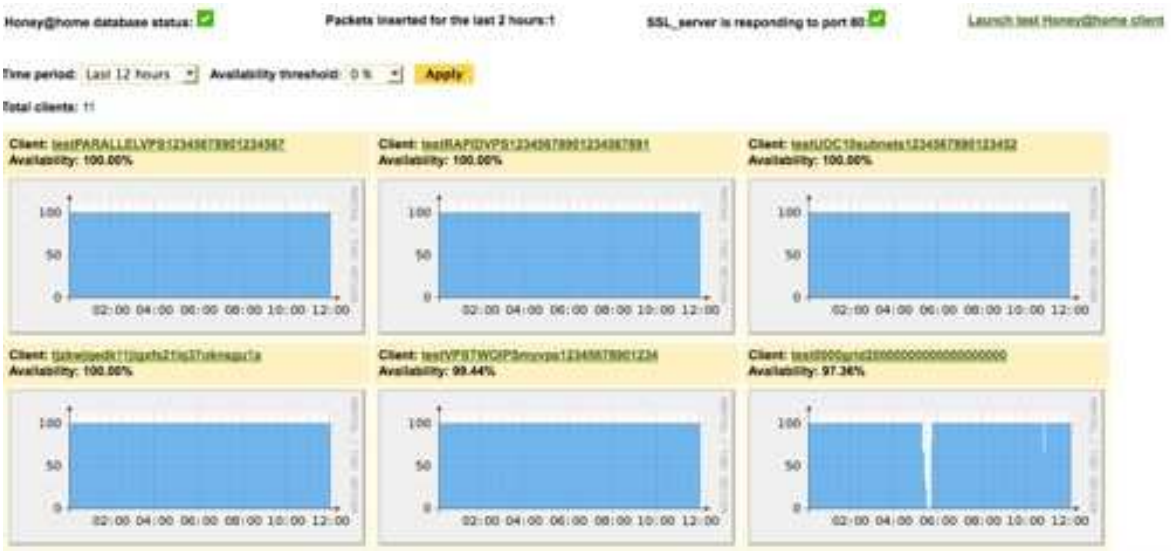


Figure 5.8: Uptime graphs for Honey@home clients

5.5 Deployment and monitoring

Several tens of Honey@home clients have been deployed so far, covering around five thousand unused IP addresses. Both Unix and Windows versions are stable enough. One of the issues we faced during the deployment is that Unix clients consume all the CPU when they are installed inside a virtual machine. The problem was proved to be caused by the library used to capture packets from the virtual interface.

Another deployment challenge is the efficient and scalable monitoring of Honey@home clients. Honey@home clients are designed to send a heartbeat pulse to the NoAH core every minute as an indication of their uptime status. Using the heartbeat information, we know which client are online and for how long. We reconstruct the heartbeat information as SNMP-like graphs which are published in a private web page. Figure 5.8 is a screenshot of the uptime graphs. By default, we plot only the graphs for those clients that were active at least for one minute during the last twelve hours. We have two parameters to customize the displayed graphs. The first one is the monitoring period. Changing this period, we can see which clients were active at least for one minute up to one year ago. The second one is the availability threshold. By default this threshold is 0%, meaning that clients that send at least one heartbeat are displayed. Setting this threshold to 100% only the clients that were up for the whole monitoring period (as defined by the first parameter).

The graphs are consisted of two parts. The first one is the key of the client. As the monitoring page is available only to NoAH administrators, there is no need to anonymize the client keys. Clicking on the client key the user is redirected to a page of the stats.fp6-noah.org domain which displays traffic statistics for the specific client. The page contains similar statistics as described in Section 4.7. Additionally to these statistics, the attack conversations and malware instances captured by the client are displayed. The second part is the SNMP-like graph that displays the availability of the client as defined by the two parameters described before. The graph is also clickable, redirecting the user to a page containing the availability graphs for all monitoring periods. In that way, we can observe the

behavior of the client for different time periods.

5.6 Publications

The following publications were made for the Honey@home architecture:

- Spiros Antonatos, Kostas G. Anagnostakis and Evangelos P. Markatos. "Honey@home: A New Approach to Large-scale Threat Monitoring." In Proceedings of the 5th ACM Workshop on Recurring Malware (WORM'07), November 2007, Alexandria, VA, USA
- S. Antonatos, M. Athanatos, G. Kondaxis, J. Velegrakis, N. Hatzibodozis, S. Ioannidis and E. P. Markatos. "Honey@home: A New Approach to Large-Scale Threat Monitoring." In Proceedings of the 1st WOMBAT workshop on Information Security Threats Data Collection and Sharing (WISTDCS). April 2008, Amsterdam, The Netherlands
- Elias Athanasopoulos and Spiros Antonatos. "Enhanced CAPTCHAs: Using Animation To Tell Humans And Computers Apart." In Proceedings of the 10th IFIP Open Conference on Communications and Multimedia Security (CMS'06), Heraklion, Greece, October 2006

Chapter 6

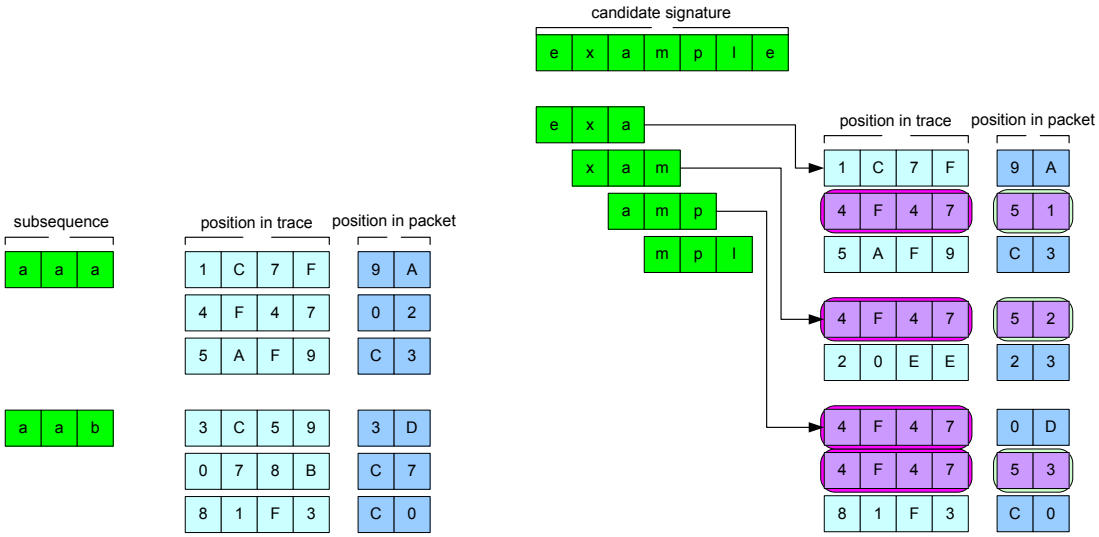
Attack signature validation

The NoAH architecture, apart from collecting attack information and malware vectors, is also able to produce signatures for the exploits that targeted their high-interaction honeypots. As described in Section 4.4, the Argos system is able to produce signatures that can be used to filter out malicious input. Besides Argos, the intrusion detection industry is also developing intrusion prevention systems that can block suspicious traffic using the most reliable detection heuristics available. Similarly, techniques such as Microsoft's Shield provide lightweight vulnerability-specific filters that can be implemented on the end-host by intercepting and analyzing incoming protocol messages. In both cases, the signatures or filters to be distributed to users are reasonably small to be quickly pushed to a large number of sites, and much easier to compose than a permanent fully-blown update or patch. Since the inexact nature of new signatures introduces the risk of accidentally blocking legitimate traffic, the accuracy of these signatures need to be tested extensively. This testing procedure usually takes hours or even days because it needs to collect a large enough amount of traffic data to validate new signatures.

To speed up the process of signature validation, a solution is to collect and store recent network traffic in advance and use it to validate new signatures. This solution, however, has two main challenges: (1) how to collect and store a huge amount of network traffic efficiently and (2) how to retrieve stored data to perform fast signature validation. The first challenge can be dealt with the use of either huge data-centers or distributed solutions, like peer-to-peer systems. However, this problem is out of the scope of this work. For the second challenge, inspired by work in the database community, to achieve fast signature validation, we trade space for time by creating indices for every piece of data. In this way, the system only needs to perform signature validation over data indices, which is fast. Since network data are binary while existing indexing techniques employed in databases only work for text data, we also propose a method that creates indices for binary data.

6.1 Signature Validation Algorithm

As mentioned previously, a major concern in our work is the time it takes to complete signature validation. Since it is desirable to perform a fast signature validation over a large amount of traffic data, we need to design an efficient algorithm. In our approach, we incorporate time-space trade-off techniques that are well understood in the context of databases. These techniques allow users to quickly scan for the existence of a key in a database. Since we



(a) A snapshot from the index file for the 3-byte subsequences "aaa" and "aab". For each subsequence we store the position in trace (4 bytes) and position in packet (2 bytes)

(b) An example that demonstrated how our algorithm matches signature "example" against the index file

Figure 6.1: Outline of the indexing algorithm

expect from our algorithm a response time only in a few seconds, if our algorithm does not use any pre-processing of the captured traffic, only the time required to read the captured traffic from the storage device would be potentially great if the amount of data is big. As a consequence, we have to pre-process the captured traffic in order to minimize the data that our algorithm must read from the storage devices (hard disks). To be more specific, our algorithm has two phases, an "offline" phase where the captured traffic (trace) is processed and an index file is created, and an "online" phase where the algorithm uses the index file to validate the signature. We respectively present these two phases in subsequent parts of this section.

6.1.1 Offline phase

In the "offline" phase, the basic idea of the algorithm is to index every *n-byte* sequence appearing in the captured traffic. For every appearance of each sequence a 6-byte record is kept: 4 bytes for the packet number on which the sequence was found and 2 bytes for the position of the sequence inside the packet. As an example, Figure 6.1(a)) shows indices of the 3-byte subsequences "aaa" and "aab". The aim of our algorithm is to retrieve only the information stored on the index, eliminating the need to search on the real captured traffic. Thus, the size of information for each sequence must be as little as possible. By increasing n, the information stored for each sequence is reduced but the number of sequences we have to index increases. For example, choosing 1 as n, we only have 256 indices but each index is several megabytes (assuming a 1 GB input trace). Choosing 4 as n, each index contains a few records, but we have 2^{32} indices.

6.1.2 Online phase

Since the main idea to validate a signature is to count the number of occurrences of the signature pattern in the stored network traffic, the "online" phase is actually a search phase that performs exact matching based on the information stored in the indices. Initially, we retrieve the indices for the n-byte sub-sequences that form the pattern. We then correlate the retrieved information to find packets in which all sub-sequences are found and their positions are adjacent. Specifically, we first correlate the index of the first subsequence with the index of the second subsequence. Then, we check all 6-byte records to find those that have a common packet number. For instance, if a record of first index indicates that the first subsequence is found in packet A and packet A never appears on the records of the second index, then this record is dropped. For the records that have the same packet number, positions are checked. If in the first index there is a record saying "packet A position B", then we try to find if there is a record in second index that says "packet A position B+1". If such a record is found then the record of the second index is checked against the index of the third subsequence in order to locate a record "packet A position B+2" and so on. If the checks are successful up to the index of the last subsequence, then we have a match in packet A at position B. Note that we can apply the same technique to index header fields, such as IP addresses or TCP/UDP ports.

Figure 6.1(b) illustrates an example where a candidate signature is validated. Assume that we have $n = 3$, we need to search for the indexes of 3-byte sequences that form the signature, that is "exa", "xam", "amp", "mpl" and "ple". Note that to be neat, we only display indices of the three first subsequences in the figure. Our first step is to retrieve indices of "exa" subsequence. Since this subsequence is found in three packets 1C7F, 4F47 and 5AF9, these packets are candidates to include the signature. Our second step is to retrieve the indices of the next subsequence "xam". This subsequence is found in two packets 4F47 and 20EE. Since the subsequence is no found in packets 1C7F and 5AF9, we exclude these two packets from the packet candidate list. We further check packet 4F47 and since the subsequence appears in the adjacent position (52) in relation to first subsequence, packet 4F47 is still kept in the candidate list. As a third step, we retrieve indices for subsequence "amp". This subsequence is found in packet 4F47 and in an adjacent position in relation to the previous subsequence so this packet continues to be a candidate. This process continues until the last subsequence and if all subsequences are contained in packet 4F47 and in adjacent positions, we can say for sure that this packet contains the signature "example". Furthermore, we are able to answer the position of the signature inside the packet, that is "example" is found at packet X in position Y. This is useful as many signatures contain information about depth and offsets of the string they search for.

Our approach needs 6 times more space than the original trace in worst case. However, during the index creation we perform several optimizations. If we have a sequence that contains the same character, for example ten consecutive appearances of character A, then we do not store 10 indices but a single one that denotes we see character A at a given offset and for 10 times. This optimization has decreased our storage needs to a factor of 5.2. A second optimization that improves our speed is a in-memory bitmap that keeps track if a subsequence has stored information or not. In cases where we choose to index 4-byte sequences. many subsequences do not appear in the traffic trace. The in-memory bitmap provides a fast way to find out if a subsequence has stored indices or not; if the signature contains a subsequence that has no stored indices, then we know for sure that this signature will never match. Before

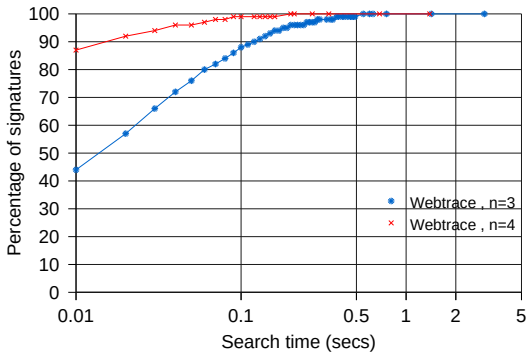


Figure 6.2: Cumulative distribution function of search time for n=3 and n=4 for a trace containing web traffic

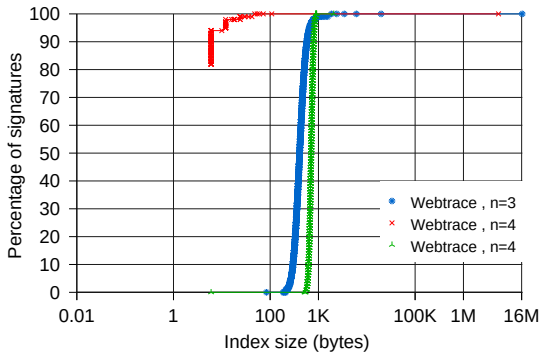


Figure 6.3: Cumulative distribution function of sizes for indices for a trace containing web traffic and a trace with random payload

we start the matching algorithm, we check that all subsequences that form the signature have stored indices. If not, the searching algorithm as described earlier is invoked.

6.2 Experimental Study

To evaluate the performance of our proposed signature validation service, we implemented a prototype of the system and tested the prototype on signatures found on Snort [145], a popular intrusion detection system. We tested these signatures against the two real network traces: FORTH.webtrace and NLANR.MRA. Both traces contain 3GB of data. While FORTH.webtrace is a trace captured during a portal mirroring, NLANR.MRA is a trace with random payload. Since we only have two real network traces, we just tested the prototype on two computers, each processed a network trace. Note that even though we used only two computers to test, the scalability of the system is still guaranteed since we employed P2P architecture for our service.

6.2.1 Overhead Cost

We evaluated the time to process network traces to create indices and the storage required to hold indices. In terms of processing time, our method takes less than an hour to generate indices for these experimental network traces. In terms of storage cost, our method requires an amount of storage 10 times greater than the size of the network traces (2 bytes for packet position, 4 bytes for packet number, and 4 bytes for next pointer) plus a fixed 64MB pointer table (3-byte sequences) or 16GB pointer table (4-byte sequences).

6.2.2 Signature Validation Performance

The results for 3- and 4-byte sequences are summarized in Figure 6.2. For almost 95% patterns we tested, our algorithm achieves sub-second search time. Our preliminary results also indicate that hotspots presented in the rest 5% of patterns are one to two orders of magnitude more effective than traditional linear searches. The dominant cost of our approach is the size of the indices we need to retrieve. The size of each index for two traces is presented

in Figure 6.3. For almost 95% of sequences, we need to retrieve up to 1 Kbyte, although the maximum value reaches 16 Mbytes due to some popular sequences like consecutive zeros found in JPEG images. In the ideal case, this of random payload, each index is 500 to 900 bytes long. As the data we have to retrieve from disk are only a few kilobytes for the 3 GBytes trace, we can expect that time for searching on Terabyte traces is also near one second. Either fetching a few Kilobytes or a few Megabytes (e.g. less than 20MB) from a local hard disk requires almost the same time.

For comparison purposes we used Snort to validate some of its own signatures. The result shows that without considering the time to collect Snort required around 80 seconds to validate a signature on a 3 Gbytes trace. Most of the time is spent on reading the packet trace from the disk, while the user time is nearly 3 seconds. Doing the same validation with our algorithm takes around 1 second for 80% of the possible patterns.

6.3 Publications

An extended version of this part of work was published in:

- Spiros Antonatos and Vu Quang Hieu. "Harnessing the power of P2P systems for fast attack signature validation". In Proceedings of 3rd International Conference on Network & System Security (NSS 2009). October 2009, Gold Coast, Australia

Part of this work was patented under the title "Apparatus and Method For Analysis of Data Traffic" (K.G. Anagnostakis, S. Antonatos) (PCT/SG2008/000040, November 2008, previously also US Provisional Patent Application No.60/900,342)

Chapter 7

Related work

Web security has attracted a lot of attention in recent years, considering the popularity of the Web and the observed increase in malicious activity. Rubin *et al.* [147] and Claessens *et al.* [84] provide comprehensive surveys of problems and potential solutions in Web security, but do not discuss any third-party attacks like puppetnets. Similarly, most of the work on making the Web more secure focuses on protecting the browser and its user against attacks by malicious websites[119, 105, 96, 83, 106].

The most well-known form of HTML tag misuse is known as cross-site scripting (or XSS) and is discussed in a CERT advisory in 2000 [79]. The advisory focuses primarily on the threat of attackers injecting scripts into sites such as message boards, and the implications that such scripts could have on users browsing those sites, including potential privacy loss. Although XSS and puppetnet attacks both exploit weaknesses of the Web security architecture, there are two fundamental differences. First, puppetnet attacks require the attacker to have more control over a web server, in order to maximize exposure of users to the attack code. Injecting puppetnet code on message boards in a XSS fashion is also an option, but is less likely to be effective. The second important difference is that puppetnets exploit browsers for attacking third parties, rather than attacking the browser executing the malicious script. Several proposal for defending against such attacks have been recently explored[143, 109, 172].

During the course of our investigation we became aware of a report [171] describing a DDoS attack that appears to be very similar to the one described in this thesis. The report, published in early December 2005, states that a well-known hacker site was attacked using a so-called "xflash" attack which involves a "secret banner" encoded on websites with large numbers of visitors redirecting users to the target. According to the same report, the attack generated 16,000 SYN packets per second towards the target. As we have not been able to obtain a sample of the attack code, we cannot directly compare it to the one described here. However, from the limited available technical information, it seems likely that attackers are already considering puppetnet-style techniques as part of their arsenal.

Another example of a puppetnet-like attack observed in the wild is so-called "referer spamming" [102], where a malicious website floods some other site's logs to make its way into top referer lists. The purpose of the attack is to trick search engines that rank sites based on link counts, since the victims will include the malicious sites in their top referer lists.

The reconnaissance technique relies on the same principle used for timing attacks against browser privacy [97]. Similar to our probing, this attack relies on timing accesses to a particular website. In our case, we use timing information to infer whether the target site exists or

is unreachable. In the case of the Web privacy attack, the information is used to determine if the user recently accessed a page, in which case it can be served instantly from the browser cache.

Puppetnets are malicious distributed systems, much like reflectors and botnets. Reflectors have been analyzed extensively by Paxson [138]. Reflectors are regular servers that, if targeted by appropriately crafted packets, can be misused for DDoS attacks against third parties. The value of reflectors lies both in allowing the attacker to bounce attack packets through a large number of different sources, hereby making it harder for the defender to develop the necessary packet filters, as well as acting as amplifiers, given that a single packet to a reflector can trigger the transmission of multiple packets from the reflector to the victim.

There are several studies discussing botnets. Cooke *et al.* [86] have analyzed IRC-based botnets by inspecting live traffic for botnet commands as well as behavioral patterns. The authors also propose a system for detecting botnets with advanced command and control systems using correlation of alerts. Other studies of botnets include [168, 122]. From our analysis it becomes evident that botnets are much more powerful than puppetnets and therefore a much larger threat. However, they are currently attracting a lot of attention, and may thus become increasingly hard to setup and manage, as end-point and network-level security measures continue to focus on botnets.

Honeypot technologies have also attracted the interest of researchers over the last decade. An overview of the work related to the NoAH infrastructure and the Honey@home approach follows.

In [163], the authors describe the risk to the Internet due to the ability of attackers to quickly gain control of vast numbers of hosts. They argue that controlling a million hosts can have catastrophic results because of the potential to launch distributed denial of service (DDoS) attacks and access any sensitive information that is present on those hosts. Their analysis shows how quickly attackers can compromise hosts using "dumb" worms and how "better" worms can spread even faster. In subsequent work [159], the same authors show how a worm using pre-compiled lists of IP addresses known to be vulnerable can infect one million hosts in half a second.

Collapsar[108] proposes a decentralized architecture composed of a large number of honeypots deployed in different network domains. This approach tries to address the problem that centralized honeypot farm have limited view of Internet activity. The core idea of Collapsar is to deploy traffic redirectors in multiple network domains and examine the redirected traffic in a centralized farm of honeypots. This approach has the benefit that we can deploy honeypots in many networks without the need of honeypot experts on each network. All the processing and detection logic will be done in the centralized honeypot farm, also referred as Collapsar center.

An overview of the Collapsar architecture is shown in Figure 7.1. The Collapsar architecture has three parts. The first part is the traffic redirector. The traffic captures all packets and afterwards filters them, according to rules specified by the network administrator. All packets that pass the filter are encapsulated and sent to the Collapsar center. The redirection can be done either through the Generic Routing Encapsulation (GRE) tunneling mechanism of a router or using an end-system-based approach. The redirector is implemented as a virtual machine running an extended version of User-Mode Linux[44] (UML), using libpcap,libnet and specialized kernel modules.

The second part is the front-end of the Collapsar center. It receives encapsulated packets from redirectors, decapsulates them and dispatches them to honeypots of the Collapsar center.

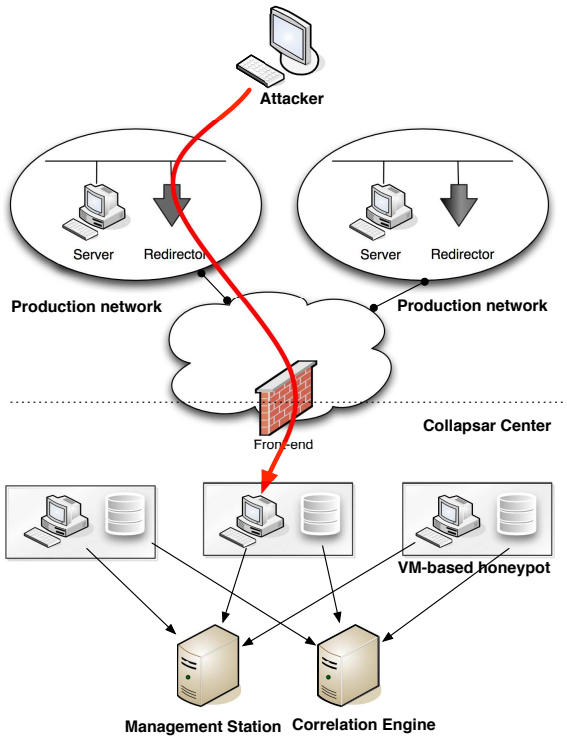


Figure 7.1: Architecture of Collapsar

It also takes responses from honeypots and forwards them to the originating redirectors. Upon receipt, redirectors will inject the responses into their network. In that way, an attacker has the sense that communicates with a host in the network of the redirector but in reality she communicates with the Collapsar center. However, the front-end does more than packet dispatching. Its role is extended to assure that traffic from honeypots will not attack other hosts on the Internet. To prevent such malicious activities, introduces three assurance models: logging, tarpiting and correlation. The logging module is embedded in the honeypots guest OS as well as log storage in the physical machine's host OS in order to be invisible to the attacker. The tarpiting module throttles outgoing traffic from honeypots by limiting the transmission rate and also scrutinizes outgoing traffic based on known attack signatures. Snort-inline performs this task. Snort-inline is a modified version of Snort[145], a very popular intrusion detection system. While Snort is a system that passively monitors traffic, Snort-inline intercepts traffic and prevents malicious traffic from being delivered to the protected network. The correlation module is able to detect network scanning by correlating traffic from honeypots that logically belong to different production networks.

The last part of the architecture is the Collapsar center. The center is a farm of high-interaction honeypots. Honeypots run services inside a virtual machine and have the same network configuration as other hosts in the production network, that is the hosts running the redirectors. Virtual machines used are VMware and UML, with UML being more preferable due to the fact that it is open-source and allows better customization, especially to network virtualization issues.

We can identify two major drawbacks in the approach proposed by Collapsar. The first one is that redirectors need a dedicated machine that communicates with a predefined set of front-ends, imposing administrative overhead for the maintenance of the redirector. Furthermore, it implies a level of trust between the redirectors and the Collapsar center Once the identity of front-ends is known, they are susceptible to direct attacks and then redirectors become useless. The second drawback is that traffic redirection adds almost double latency, according to paper measurements, that helps attackers to identify redirectors, e.g. by correlating response times from the redirector and other machines in its production network.

Honeypot farms usually require a large number of physical machines in order to run a few tens of virtual machines. Virtual machines in fact consume a large amount of physical memory and processing power and it is hard to run more than ten in the same physical machine. The Potemkin approach[173] proposes an architecture that overcomes this problem and improves honeypot scalability. The Potemkin architecture is based on two key observations. The first one is that most of a honeypot's processor cycles are wasted idling, as they usually wait for an adversary to connect to them. The second one is that, even when serving a request, most of a honeypot's memory is also idle.

On each physical machine, a virtual machine monitor (VMM) is running. When a packet arrives for a new IP address, the VMM spawns a new virtual machine. In this way, we have a virtual machine running only when needed, that is on a per-request basis. However, spawning a new VM for each request is an expensive operation. To reduce this overhead, the flash cloning technique is used in the Potemkin architecture. After the boot of the first VM instance, a snapshot of this environment is taken. This snapshot is then used to derive subsequent VM images. The process responsible for VM cloning is the cloning manager. It instructs Xen to create a VM based on the reference snapshot. After the VM is created and successfully resumed, the clone manager instructs the guest operating system to change its IP address based on the destination address of the request. During the cloning process the VMM stores packets destined for the VM. After cloning is finished, packets are flushed to the VM. Per-request VM cloning solves the problem of wasted processor time spent by a honeypot on waiting for requests. To overcome the problem of large memory consumption, the delta virtualization technique is used. The notion behind delta virtualization is that most of the memory pages among VMs are common, for example pages of operating system, and thus can be shared. This technique follows the copy-on-write approach for pages that need to be changed by a VM.

Leurre.com[21] is a distributed honeypot environment that operates a broad network of honeypots covering around 30 countries. Honeypots run a modified version of honeyd and emulate three different operating systems; two from the Windows family (98 and NT server) and Redhat 7.3. Traffic and security logs are retrieved daily and stored into a centralized database. Apart from logs, raw traffic is also analyzed, mainly to derive information about attackers and specifically IP geographical location, DNS names, OS fingerprinting and TCP stream analysis. In the Leurre.com terminology, a single host running the modified honeyd is called a platform. As honeyd emulates three operating systems, each platform needs 3 dark IP addresses to listen to. These IP addresses are consecutive and each emulated OS is assigned to listen to one of them. The reason for listening to consecutive IP addresses is to identify attackers that scan subnets. If all three emulated OSes are contacted by an attacker, it is a strong indication of scanning. Participants in the Leurre.com project need to deploy a platform and in return they are granted access to the centralized database.

During the Leurre.com deployment, authors have gathered some interesting statistics con-

sidering attack sources. Note that as the paper was written in 2004, the statistical results may be invalid for present time. Most of the attacks, about 80% to 95%, come from Windows machines. 35% of the machines that launched attacks are clearly identified as personal computers. The identification was done by checking the reverse DNS name for patterns like "adsl" or "dialup". Considering most targeted ports, authors found that Windows RPC ports (135, 130 and 445) are the most popular ones.

A honeynet is an architecture proposal for deploying honeypots. Deployed honeypots can be both low- and high-interaction honeypots but honeynet architecture discusses mainly about high-interaction ones. According to the Honeynet Project[52] architecture, honeypots live in a private subnet and have no direct connectivity with the rest of the Internet. Their communication is controlled by a centralized component, actually the core of the architecture, called honeywall. Honeywall performs three operations: data capture, data collection and data analysis. Data capture mechanism monitors all traffic to and from the honeypots. The challenge here is that a large portion of the traffic is over encrypted channels (SSH, SSL, etc.). To overcome this problem, Sebek[38] was introduced to the honeynet architecture. Sebek is a hidden kernel module that captures all host activity and sends this activity to the honeywall. As Sebek runs in the host level, it can capture traffic after being decrypted. Detectability of Sebek is a challenge. The attacker must not be able that this module is running and he must not be able to see the traffic from the Sebek module to the honeywall. The detectability of Sebek is studied in [94]. The data control mechanism tries to mitigate risk from infected honeypots. As honeypots will eventually be compromised, they can be used for attacking other non-honeypot systems. Data control can be performed in many ways; limiting the number of outbound connections, removing attack vectors from outgoing traffic or limiting the bandwidth. The removal of attack vectors is performed by Snort-inline. The strategy followed is specified by each organization that deploys a honeynet. Data analysis processes all the information gathered by the data capture mechanism to collect useful statistics and attack properties. Data collection applies to organizations that have multiple distributed honeynets. Its main task is to gather and combine the data. The Honeynet alliance has currently 16 members, distributed in 3 continents. The honeynet community is very active and has published several "Know your enemy" white-papers available at project's website[1]. Honeynet farms were originally used to capture and analyze manual attacks. Recently, they have been used to track phishing attempts and botnets.

Provos et al. in [69] propose an architecture that combines the scalability of low-interaction systems with the interactivity of high-interaction ones. The architecture consists of three components: low-interaction (or lightweight) honeypots, high-interaction honeypots and a command and control mechanism. The role of low-interaction honeypots is to filter out uninteresting traffic. Connections that have not been established (the 3-way handshake was not completed) or payloads that have been seen in the past are part of the uninteresting traffic. Low-interaction honeypots maintain a cache of payload checksums. If the payload of the first packet (after connection is established) has not been seen in the past, it is considered as interesting. According to the measurements of the paper, around 95% of the packets with payload have been observed in the past. All interesting traffic is handed off to high-interaction honeypots. The handoff mechanism is implemented by a specialized proxy. Once a packet is marked as interesting, the proxy establishes a connection with the back-end and replays this packet. Next packets of the "interesting" connection will be forwarded to the

[1] `http://www.honeynet.org`

back-end by the proxy. The honeyd system is used as the main core of the low-interaction honeypots. The high-interaction honeypots run on VMware and form the back-end of the architecture. The back-end is set up not to be able to contact the outside world. Instead of blocking or limiting the outgoing connections, traffic generated by these honeypots is mirrored back to other honeypots. As long as there are uninfected machines, the infection will spread among the honeypots, allowing the capturing of exploits and payload delivery. However, this architecture does not work for malware that downloads its code from an external source, like a web site. To detect whether high-interaction honeypots are infected, their network connections are monitored as well as changes in their filesystem. The virtual machines of infected honeypots are returned to a known good state, through the snapshot mechanism of VMware. The command and control mechanism aggregates traffic statistics from low-interaction honeypots and monitors the load of high-interaction ones. It also analyzes all data from virtual machines to detect abnormal behavior such as worm propagation. The proposed hybrid infrastructure looks similar to the infrastructures proposed by Collapsar and NoAH. However, this approach focuses more on filtering the interactions before they reach the high-interaction honeypots and additionally the backend architecture is fundamentally different as in this approach mirroring is performed.

Architectures presented so far use honeypots as non-production systems, living at different network domains than production systems and listening to unused IP address space. Shadow honeypots[60] propose a different approach for detecting attacks that couples honeypots with production systems. The architecture consists of three components: a filtering component, a set of anomaly detectors and shadow honeypots. The filtering component blocks known attacks from reaching the network. Such a component can be either a signature-based detector, like Snort, or a blacklist of known attack sources. The array of anomaly detectors, each one running with different settings in respect to their sensitivity and configuration parameters, is used to classify which traffic is suspicious. The traffic that is characterized as anomalous is forwarded to shadow honeypots. Their main role is to offload the shadow honeypots as much as possible by forwarding them only the traffic that may include an attack.

Shadow honeypots are cloned instances of production servers that are heavily instrumented so as to detect attacks. Two types of shadow honeypots can be identified: loosely-coupled and tightly-coupled. Loosely-coupled honeypots are deployed on the same network of the protected server, running a copy of the protected applications but in a different machine without sharing state. However, the effectiveness of loosely-coupled shadow honeypots is limited to static attacks that do not require to build state at the application level. Tightly-coupled shadow honeypots run on the same machine as the protected applications and share their state. Shadow honeypots, as stated before, are instrumented versions of protected applications. The instrumentation allows the accurate detection of buffer overflow attacks and is based on the pmalloc() concept, as described in [153]. Pmalloc() is a replacement for malloc(), the standard memory allocation routine, and it works as follows. Before and after an allocated memory block requested to pmalloc(), read-only memory pages are placed. If an overflow attack is going to take place, it will try to write on the read-only pages and an exception will be thrown. The exception is caught by the pmalloc() routine, indicating the presence of an attack. (note: the concept of pmalloc() was extended to include statically allocated arrays). The major drawback of this instrumentation approach is the requirement for the application's source code.

When the shadow honeypot detects an attack, it state is rolled back as it was before the attack and the malicious content is not forwarded to the normal application. It also informs

the anomaly detectors about the attack so they can tune their detection models for better performance. If the shadow honeypot does not detect any attack, the request is handled to the normal application. Again, the anomaly detectors are informed that this was not an attack so they can update their models. Shadow honeypots can be also used to protect the client from client-side exploits, such as the buffer overflow in the JPEG handling routine of Internet Explorer. As an example, an instrument copy of Mozilla Firefox can handle web requests. According to the paper, the overhead of the instrumented version is around 20%.

Vigilante is an infrastructure that aims for worm containment. The architecture of Vigilante is based on the collaboration of end hosts and makes no assumptions that collaborating hosts trust each other. The proposed approach has preferred to move from network-level to host-level in order to eliminate problems like encrypted traffic or the lack of information about software vulnerabilities. End hosts act as honeypots; they run instrumented versions of software that normally wouldn't run on the host. For example, a host can run an instrumented version of a database, not a common application for a normal host. The alert generation is done using two techniques. The first uses non-executable pages around stack and heap pages to detect code injection attacks. If a buffer overflow tries to write on a non-executable page an exception is raised and caught. The second technique is the dynamic dataflow analysis. The concept of dataflow analysis is very similar to memory tainting. Data coming from the network is marked as dirty and if dirty data is going to be executed or used as critical arguments for a function, a signal for exploit is raised. Unlike approaches described in Section 2.3.3, which use specialized versions of virtual machines, Vigilante uses binary rewriting at load time. Every control transfer instruction and every data movement instruction is instrumented. A bitmap is kept to track dirty data throughout memory, one bit for each memory page. Additionally, information for where the dirty data came from is stored to help the analysis and alert generation.

The contribution of Vigilante is the concept of self-certifying alerts (SCAs). SCAs are distributed among the collaborating hosts and their novelty is that they can be verified by recipients. This property eliminates the need for trust between the hosts. Three types of SCAs can be identified: arbitrary execution control, arbitrary code execution and arbitrary function argument, with each type covering a different type of vulnerability. All types of SCAs contain some common information. This information is the identity of vulnerable service, verification information to assist alert verification and a sequence of messages that contain necessary data to trigger the vulnerability. Upon received a SCA, the host uses the verification information the sequence of messages to reproduce the vulnerability. The alert distribution is done using the Pastry system. There was no real deployment and authors did simulations to evaluate the architecture. According to their results, the generation time of a SCA varies from 18 up to 2667 milliseconds, depending on the worm instance, while verification time needs up to 75 milliseconds. With a very small fraction of detectors against the total vulnerable population, infection rate can reach up to 50%. When this fraction reaches 0.001, infection rate falls to 5% and drops to nearly zero when this fraction reaches 0.01. The total population simulated was 500,000 hosts but only a subset S was vulnerable. The choice of S was done based on real data gathered during the worm outbreaks.

The iSink architecture[182] aims at monitoring large unused IP address space, such as /8 networks. Although this work focuses on measuring packet traffic, we will study its design and properties, which are related to honeypot infrastructures. The design model of iSink is to respond to traffic that goes to unused IP address space. However, as iSink deals with large address space, a scalable architecture is needed. Authors considered four systems that can be

used as responders: Honeyd, honeynet, LaBrea and ActiveSink. ActiveSink is a framework written by authors using the Click router language[115]. This framework includes several responders, such as ICMP, ARP, Web, SMTP, IRC and NetBIOS responders. Additionally, responders for MyDoom and Beagle backdoors were also implemented. ActiveSink responders are stateless, and still accurate, in order to achieve a high degree of scalability. Even for complex protocols, it is possible to construct a response by looking at the last request packet. Furthermore, there is need for interacting with the attacker up to the point where an attack is detected. For example, if an attack is taking place on the fifth step of a very long conversation, there is no need to emulate further than this step. The four systems were tested along five main criteria: configurability, modularity, flexibility, interactivity and scalability. Honeynet was discarded because of low configurability and medium scalability. LaBrea, on the other hand, has high scalability but very low configurability, modularity and flexibility. Honeyd presents high configurability and flexibility but medium scalability. ActiveSink, finally, is highly scalable, configurable, modular and flexible, with only drawback depending its interactivity on responders (medium interactivity according to authors). The performance of iSink architecture was evaluated using TCP and UDP packet streams at rates up to 20,000 packets per second. Each packet was a connection attempt. iSink didn't suffer from losses at any rate for both protocols. The system was also deployed in four class B networks and one class A network. The amount of traffic received in these networks was large. iSink node for class A network was receiving between 4,000 and 20,000 packets per second.

The Internet Motion Sensor (IMS)[66] is a Internet monitoring system covering 28 unused IP blocks, ranging from /25 to /8. The IMS architecture consists of a set of blackhole sensors. Each sensor monitors a block of unused IP addresses and has both an active and passive component. The passive component is a traffic logger that records all traffic destined to the monitored address space. The active component is a lightweight responder, aiming at raising the level of interaction with the attacker. While UDP and ICMP are stateless and no response is needed, TCP needs a connection to be established until the data of the actual attack can be seen. The lightweight responder establishes the incoming TCP connections and captures the payload data. The responses are application-agnostic as they do not provide any protocol emulation. While this may not work in all cases, it is enough for many cases like the Blaster worm that did not wait for any application responses. To avoid the recording of unnecessary traffic, payload caching is used. The checksum of payload data is computed for each packet and if it is has been seen in the past the payload data are not stored. If not, the payload is stored along with its checksum. The observed hit rate in IMS sensors is approximately 96%, which yields to storage savings by a factor of two. Checksums also provide a convenient way to gather quick statistics about traffic and a way to filter out traffic from other components, like intrusion detection systems. The IMS systems tries to answer questions regarding worm demographics, virulence, propagation and the timescale of responses to attacks. The paper provides a number of traffic statistics over a 7-day period and uses the Blaster worm as an example.

HoneyStat[91] is an effort to compliment global monitoring strategies and provide an early worm detection system by inspecting local networks. Honeystat proposes minimal honeypots created in an emulator and multihomed to cover a large address space. The emulator used is the VMware GSX server V3, which supports up to 64 isolated virtual machines on a single hardware system. Most modern operating systems support multihoming, which is assigning multiple IP addresses to a single network interface. Windows NT allow up to 32 IP addresses while most flavors of Linux up to 65536. Each virtual machine, also called node,

was configured to have 32MB RAM and 770MB virtual drive. Nodes were instrumented to capture three types of events: memory, network and disk events. Memory events are considered any kind of alert by buffer overflow protection software, like StackGuard[88] or Windows logs. Each outgoing TCP SYN or UDP traffic is a network event. Disk events are generated by activities that try to write to protected file areas. Kqueue is a monitoring tool that performs this task, although protected file areas have to be listed manually (e.g. the c:/windows/system directory or the registry file). For each the OS and patch level were also recorded as well as associated data, like stack state for memory events, packets for network events and delta of the file changes for disk events.

All recorded events are forwarded to an analysis node. If the event is a network event then the reporting honeypot is reset. This action is taken to prevent the node from attacking other machines. Besides, by the time a network activity is initiated, enough information has been recorded to study the worm infection cycle. The analysis node is also responsible to decide if some nodes should be redeployed. For example, if a worm hits a vulnerable OS, it would make sense to redeploy some of the nodes with the vulnerable OS to capture more events for the worm. Finally, each event is correlated with all other events for detection of attack patterns. Event correlation is done using logistic regression, a method which is effective for data collected in short observation windows. Network traffic from 100 /24 darknets was used to evaluate the HoneyStat approach in terms of detection accuracy. The logit analysis eliminates noise from generated events when this traffic is injected to honeystat nodes and identifies correctly all worm activities. HoneyStat evaluation, in terms of false positives, was done using attack data from the Georgia Tech Honeynet project mixed with non-attack background traffic. Attack data was from July 2002 to March 2004. There are two reasons for a honeystat node to generate a false positive: a) normal background traffic is characterized as worm and b) repeated human break-ins are identified as a worm. Honeypot events produced by nodes did not produce any false positive. However, according to authors, the false positive rate may be different than zero in a larger dataset.

The HoneyTank project[170] aims at collecting and analyzing large amounts of malicious traffic. The data destined for the unused IP address space of a network are forwarded by cooperating network devices, like routers, and captured by an IDS called ASAX (Advanced Sequential Analyzer on Unix). The IDS emulates services on these addresses to capture data sent to services (e.g. web-server). The architecture consists of a single sensor on which ASAX is deployed and the cooperating devices. As the sensors are distributed,the authors propose to use mobile IPv4 (MIPv4) to redirect the traffic to the IDS sensor. The other alternative is to use ARP and GRE tunneling but GRE is not fully supported by routers. To integrate a new unused IP address, the IDS sends a request for this address to the router for traffic redirection.

All packets redirected to the central sensor are captured by the libpcap library. This data is imported and analyzed by the ASAX IDS. ASAX allows the flexible declaration of rules that are applied to the captured data. Each rule declaration is written in the "*RUSSEL*" language and is used to analyze the captured packet and to apply actions that depend on the result of the prior analysis. The rule declaration allows the authors to emulate the basic behavior of the TCP protocol itself and the protocols HTTP and SMTP protocol. The advantage of the proposed approach is that no protocol state is required, increasing the scalability of the architecture (similar to iSink responders that are also stateless). The stateless emulation of protocols is done by matching regular expressions. For example, if a request matches the "GET .* HTTP/1.1" expression, a HTTP reply with code 200 is returned. However,

although the level of emulation is adequate for automated programs, a manual intrusion is able to detect that services are emulated.

Authors evaluated their approach by deploying a HoneyTank prototype in their class B campus network. The ASAX IDS was configured to emulate HTTP and SMTP and to accept connections for all other TCP ports. The port and flow length distribution were calculated. HoneyTank was deployed for around 6 hours and all the traffic received from and sent by ASAX was recorded to a tcpdump trace. The trace was then given as input to a Snort system and the occurrences of the attacks were measured (20 attacks were detected). Authors also compared HoneyTank with Darknet[90]. The SYN packets of HoneyTank trace were given as input to the Darknet system and the trace produced by Darknet was tested with Snort. Darknet trace contained 5 out of 20 attacks, providing less visibility to attackers than HoneyTank.

Current practice suggests that NIDS signatures for new worms are usually available several hours or days after the initial worm outbreak, due to the manual signature generation process. This implies that, given the propagation speeds of worms so far, signatures are available after the worm has infected the majority of its victims. In such timescales, and with theoretic results showing that well-prepared worms can infect the majority of the vulnerable population in less than 10 minutes [161], it becomes clear that human-mediated containment methods cannot provide an effective defense against fast spreading worms [162].

Two of the earliest network level worm detection and signature generation systems, Earlybird [154] and Autograph [113], rely on the identification of invariant content that is prevalent among multiple worm instances, a technique called *content sifting*. When sufficient worm instances have been identified, these techniques generate NIDS signatures by finding the longest contiguous byte sequence that is present in all instances.

EarlyBird [154] is based on two key behavioral characteristics of Internet worms: the prevalence of invariant content among different worm instances, and the dispersion of the source and destination addresses of infected hosts. Earlybird operates on network packets captured passively at a single monitoring point, such as the DMZ network of an organization. For each packet, digests of all 40-byte substrings are computed incrementally using Rabin fingerprints. The most prevalent digests among all observed packets to the same destination port are identified using space-efficient multi-stage filters. The intuition behind grouping by port number is that repetitive worm traffic always goes to the same destination service, while other repetitive content such as popular web pages or peer to peer traffic often goes to ports randomly chosen for each transfer. Furthermore, in order to ascertain that packets with recurring contents belong to a current epidemic, the number of distinct source and destination IP addresses seen in these packets should reach a certain threshold before raising an alert.

Similarly to Earlybird, Autograph [113] is an automated signature generation system for previously unknown worms. In contrast to Earlybird, which inspects all monitored traffic, Autograph uses a first stage portscanning-based flow classifier for identifying potential malicious connections. Autograph performs TCP stream reassembly for the inbound part of TCP connections, which reconstructs the client-to-server stream from the incoming packet payloads. The resulting suspicious reassembled streams, grouped by destination port number, are fed to the second signature generation stage. The signature generation algorithm searches for repeated non-overlapping byte sequences across all suspicious streams using Rabin fingerprints.

Akritidis et al. present a worm detection and signature generation approach based on content sifting [57], similar to Earlybird and Autograph. The proposed technique operates

on reassembled TCP streams, but explicitly discards traffic sent from servers to a clients and processes only client requests which may carry a worm payload. Prevalent substrings are found using Rabin fingerprints and value-based sampling, which reduces significantly the processing overhead. Other optimizations over previous approaches include using substrings of 150–250 bytes instead of the 40-byte strings used by Earlybird, which significantly reduces false positives, and giving a higher weight to substrings found in the first few kilobytes of a new stream, which effectively discards repeated control messages of peer-to-peer protocols that appear like worm attacks, but are sent over already established connections that have already transferred a significant amount of data.

Polygraph [133] generates signatures for polymorphic worms by identifying common invariants, such as return addresses, protocol framing, and poor obfuscation, which the authors argue that are present among different polymorphic worm instances. The key difference to previous automated signature generation approaches is that Polygraph looks for *multiple disjoint* byte sequences, which may be of very small size, instead of single contiguous byte sequences. The authors explore the effectiveness of different classes of signatures. Specifically, among longest substring, best substring, conjunction, token subsequence, and Bayes signatures, token subsequence signatures are the most effective, with the lowest false positive rate. Token subsequence signatures consist of multiple disjoint substrings and can be expressed with regular expressions. Bayes signatures are similar to token subsequence signatures, but allow for probabilistic rather than exact matching, by assigning an occurrence probability to each token. Similarly to Polygraph, Hamsa [123] is a network-based automated signature generation system for polymorphic worms. Using greedy algorithms, Hamsa achieves significant improvements in speed, noise tolerance, and attack resilience over Polygraph.

Honeycomb [117] is probably the first automated system for the extraction of attack signatures from network traffic. Honeycomb has been implemented as an extension to the `honeyd` [142] low interaction honeypot, which groups connections to the same destination port, and attempts to extract patterns from the exchanged messages. Pattern detection is performed by applying the longest common subsequence (LCS) algorithm to the messages of the different connections in the same group, in two different ways. Given a sequence of messages, *horizontal detection* applies the LCS algorithm to the ith messages of all connections with the same destination number. In contrast, *vertical detection* first concatenates the incoming messages of individual connections, and then applies the LCS algorithm to the different concatenated streams.

Nemean [183] is a system for automated signature generation from honeynet traces, aiming to reduce the rate of false positives by creating semantics-aware signatures. The architecture consists of two components: the data abstraction component (DAC) and the signature generation component (SGC). The DAC takes as input a honeynet trace and first normalizes the packets for disambiguating obfuscations at the network, transport, and application layers (for HTTP and NetBIOS/SMB protocols). Then, it groups packets to flows, flows to connections (request-response message collections), and connections to sessions, which are defined as sequences of connections between the same pair of hosts. Normalized sessions are finally transformed into XML-encoded semi-structured session trees, suitable for input to the clustering module. In the SGC, a clustering module groups connections and sessions with similar attack profiles according to a similarity metric. Then, the automata learning module constructs an attack signature from a cluster of sessions using a machine-learning algorithm. Due to the hierarchical nature of the session data, Nemean produces separate signatures for connections and sessions, appropriate for usage with the Bro NIDS [137].

Wang *et al.* [175] propose applying content-sifting techniques within clusters of traffic, as created with header-based multi-dimensional flow clustering. In many cases, this will improve the purity of signature pools. Wang and Stolfo [176] use byte distributions (1-gram frequencies) in payloads to identify traffic that deviates from normal (based on training).

7.1 Comparison and advances from related work

The work that is most closely related to Puppetnets is a short paper by Alcorn [58] discussing "XSS viruses", developed independently and concurrently [55] to our investigation. The author of this work imagines attacks similar to ours, focusing on puppetnet-style worm propagation and also mentions the possibility of DDoS and spam distribution. The main difference is that our work offers a more in-depth analysis of each attack as well as concrete experimental assessment of the severity of the threat. For instance, a proof-of-concept implementation of an XSS virus that is similar to our puppetnet worm is provided albeit without analyzing its propagation characteristics. Similarly, DDoS and spam are mentioned as *potential* attacks but without any further investigation. The author discusses referrer-based filtering as a potential defense, which, as we have shown, can be currently circumvented and is also unlikely to be sufficient in the long term. One major difference in the attack model is that we consider popular malicious or subverted websites as the primary vector for controlling puppetnets, while [58] focuses on first infecting Web servers in order to launch other types of attacks. Similar ideas are also discussed in [128]. While the work of [58] and [128] are both interesting and important, we believe that raising awareness and convincing the relevant parties to mobilize resources towards addressing a threat requires not just a sketch or proof-of-concept artifact of a potential attack, but extensive analysis and experimental evidence. In this direction, we hope that our work provides valuable input.

An attack similar to puppetnets is described in [158]. That work was based on the observation that most users setup their home networks using inexpensive broadband users that offer wired and wireless networking capabilities. Malicious websites can trick users to open the configuration page of their home router and make them change network properties, especially the DNS server. That work focused primarily on how to mount pharming attacks that lead to denial of service, malware theft and identity theft.

The technique we used for sending spam through puppets was first described by Jochen [169], although we independently developed the same technique as part of our investigation on puppetnets. Our work goes one step further by exploring how such techniques can be misused by attackers that control a large number of browsers.

A scanning approach that is somewhat similar to how puppets could propagate worms is imagined by Weaver *et al.* in [180], but only in the context of a malicious Web page directing a client to create a large number of requests to nonexistent servers with the purpose of abusing scan blockers. The misuse of JavaScript for attacks such as scanning behind firewalls was independently invented by Grossman and Niedzialkowski [101] while our study was in review [55].

The NoAH architecture was one of the first to introduce the cooperation of low- and high-interaction honeypots. The closest to NoAH architecture is Collapsar[108]. However, Collapsar introduces a set of distributed high-interaction honeypots that act as forwarders. In NoAH, we alleviate this overhead as traffic forwarding is a simple task and solutions like VPN and Honey@home can achieve the same purpose. Furthermore, the Collapsar center does not

perform any kind of filtering and all incoming connections are handled by high-interaction honeypots. NoAH uses the low-interaction honeypots as a filtering mechanism, presenting a more scalable solution in cases of large volume of traffic. To the best of our knowledge, the Collapsar architecture has not been deployed yet.

Leurre.com[21], SGNET and Honeynets[15] are three infrastructures that have been deployed and are still active. In comparison with NoAH, the Leurre.com infrastructure uses only low-interaction honeypots. This approach is very limiting. Although honeypots do not have the risk of getting infected, the amount of information they can collect is limited. Leurre.com can identify scanners and naive attacks but cannot detect complex attacks that require high level of interactivity. The NoAH architecture runs both low and high-interaction honeypots and as a result it can identify all ranges of attacks; from scanners up to complex remote exploits. The SGNET infrastructure, successor of Leurre.com, takes advantage of the capabilities of both medim and high-interaction honeypots. Medium-interaction honeypots take care of known attacks while all unknown attack instances are forwarded to modified Argos instances. This approach is similar to what NoAH proposes. However, the SGNET infrastructure was designed and implemented two years after the NoAH infrastructure. Finally, Honeynets deploy high-interaction honeypots that are monitored by a centralized component called Honeywall. The scalability of Honeynets is very low as no aggregation and filtering mechanisms are employed and a large number of physical machines is needed for large deployment scenarios. The centralized component is a single point of failure as it tries to contain possible infections by running traditional intrusion prevention systems that are vulnerable to false positives and false negatives. The NoAH architecture argues that at the end only traffic forwarding should run as the scalability of high-interaction honeypots is too low and their cost is too high. Additionally, the Honeynet approach has the risk of honeypots getting infected while in NoAH that risk is eliminated thanks to the Argos technology.

The Honey@home approach was the first one to introduce the deployment of honeypots for users that are unfamiliar with such technologies. Prior to Honey@home, no similar tools or approaches had been proposed. Recently, the HoneyPoint tool[16] has been proposed but it is a stripped version of a low-interaction honeypot that runs to the end host, without any honeyfarm running at the backend but emulation scripts running at the HoneyPoint client.

Concerning the signature validation algorithm we presented in Chapter 6, it was the first effort to perform indexing on top of network traces. Although in the literature exist tens of approach on how to perform for text (human readable content), none of them apply to network traces that consist mostly of binary data. The tries structure was the closest that could fit in our work but tries do not provide any information about the packet number and position of the matches which is essential to achieve sub-second speeds. Architectures like Vigilante have proposed self-certifying alerts (SCAs), however this type of alerts is customized for the specific architecture and is targeted for host-level signatures. Sigval is a general technique to validate signatures based on raw network traces and can be used to test any form of sigantures; from widely used network intrusion detection systems signatures, like the ones used in Snort, to customized searches, for example an ISP searching for specific patterns on its own traffic.

Chapter 8

Summary and discussion remarks

The exponential increase of attack volume and sophistication makes the cyber-security a difficult and challenging research area. Besides traditional attack vectors, like worms, viruses and Denial-of-Service attacks, new threats been added to the attack landscape. This dissertation presents a new class of Web-based attacks that involve malicious or subverted websites manipulating their visitors towards attacking third parties. We have shown how attackers can set up powerful malicious distributed systems, called Puppetnets, that can be used for distributed DoS, reconnaissance probes, worm propagation and other attacks. We have attempted to quantify the effectiveness of these attacks, demonstrating that the threat of puppetnets is significant.

To defend against traditional and novel cyber-attacks, efficient and scalable defense mechanisms are needed. We have explored the use of honeypots for detection and prevention purposes. This dissertation presents the Network of Affined Honeypots (NoAH). Honeypots, non-production systems that act as decoy systems, are proved to be efficient and without false positives. NoAH is focused on honeypots that listen to unused IP address space and analyze and/or interact with malicious traffic. The architecture of NoAH presents a flexible design for deployment and collaboration of honeypots. At its core, NoAH combines low, medium and high-interaction honeypots to create a robust infrastructure. Low-interaction honeypots act as a filtering mechanism to get rid of uninteresting traffic, while medium and high-interaction honeypots perform the detection and signature generation. Overall, NoAH glues together various network and host components to form a flexible network of honeypots. Although the NoAH core is the main component of NoAH architecture, NoAH is more than a set of honeyfarms. Approaches like tunneling and honey@home extend NoAH far beyond its core and have the potential for wide address space coverage with minimal overhead. The pilot deployment and operation of the NoAH infrastructure has been presented in this thesis. The NoAH sensors receive thousands of attack conversations daily as well as they capture malware binaries. We present the statistics of the NoAH infrastructure in a publicly available Web page. Additionally, this thesis proposes and evaluates an algorithm for validation of generated signatures.

Furthermore, we have explored the design of Honey@home, a lightweight tool that enables users without expertise in honeypot technologies to contribute the fight against cyber-attacks. Honey@home claims unused IP addresses and ports, either dynamically or statically, and forwards all traffic directed to them to a centralized core of honeypots. The core consists of honeyd instances as low-interaction honeypots and Argos systems as high-interaction ones.

As Honey@home can be used by anyone, including attackers, three major challenges need to be addressed: hide the identity of users, hide the identity of honeypots and prevent automatic installations. By using Tor, a well-known and deployed anonymization network used by thousands of users, we can hide the identity of both clients and honeypots. To prevent automatic installations, each client needs to be registered through the official website, where CAPTCHA techniques are used similar to many popular large-scale services. Overall, Honey@home enables the creation of a distributed infrastructure with little effort and aims toward a scalable solution that overcomes the problem of classic honeypots, that is monitoring a small portion of unused IP address space.

8.1 Future Research

The area of honeypots is an active research division. In this dissertation we explored the architecture of a honeypot infrastructure that detects traditional attack vectors, such as worms and denial-of-service attacks, whose source is either botnets or puppetnets. However, new attack vectors should be taken under consideration. Malware propagation takes place in other communication channels, such as Instant Messaging networks and social networking sites. Malware authors take advantage of the fact that users trust the content sent by other users in their friend list. By infecting instant messenger clients or compromising IM accounts, they spread malicious URLs and executables by spamming the friend lists. We have done some first steps for detection of these attack vectors and have performed a preliminary analysis of IM-enabled phishing and malware propagation.

Lately, the number of devices that access the Internet has increased dramatically. Users now crawl the Web by handheld devices, such as mobile phones and PDA's. The nature and volume of mobile networks makes them an attractive platform to launch attacks against. Honeypot technologies could be applied to that domain too, for example decoy services running on mobile phones. Platforms like Java and Android allow the development and deployment of mobile honeypots.

The detectability of honeypots is a concerning issue. Although we have discussed how solutions like Honey@home avoid detection, the rest of sensor types still remain unprotected against discovery attacks. The static configuration of traditional honeypots is a weakness that needs to be addressed. Dynamic configurations and honeypot relocation must be explored to alleviate this problem.

Bibliography

[1] Address allocation for private internets. RFC 1918 *http://www.faqs.org/rfcs/rfc1918.html*.

[2] Advanced Honey Pot Identification and Exploitation. http://phrack.ru/63/p63-0x09.txt.

[3] Amun honeypot. http://amunhoney.sourceforge.net/.

[4] Beagle worm. http://www.symantec.com/security_response/writeup.jsp?docid=2004-011815-3332-99.

[5] Capture-HPC. https://projects.honeynet.org/capture-hpc.

[6] Capture the flag dataset. http://cctf.shmoo.com/.

[7] Cipe - crypto ip encapsulation. *http://sites.inka.de/ W1011/devel/cipe.html*.

[8] Cross-site scripting worms and viruses. http://net-security.org/dl/articles/WHXSSThreats.pdf.

[9] Dshield web honeypot project. http://sites.google.com/site/webhoneypotsite/.

[10] Generic routing encapsulation. RFC 2784 *http://www.faqs.org/rfcs/rfc2784.html*.

[11] Google hack honeypot. *http://ghh.sourceforge.net/*.

[12] Hihat, high-interaction honeypot analysis toolkit. http://hihat.sourceforge.net/.

[13] Honeybee. *http://www.thomas-apel.de/honeybee*.

[14] HoneyClient. http://www.honeyclient.org.

[15] Honeynet project.
http://www.honeynet.org.

[16] Honeypoint family of products. http://microsolved.com/?page_id=69.

[17] Honeypot-php. http://pablotron.org/software/honeypot-php/.

[18] Honeyspot: The wireless honeypot. http://honeynet.org.es/papers/honeyspot-the-wireless-honeypot.

[19] Honeytrap. http://honeytrap.mwcollect.org/.

[20] Labrea. *http://labrea.sourceforge.net/labrea-info.html*.

[21] Leurre.com honeypot project. *http://www.leurrecom.org/* .

[22] LMbench Performance Analysis Tool. http://www.bitmover.com/lmbench.

[23] Microsoft Security Bulletin MS04-028. http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx.

[24] Microsoft Security Bulletin MS06-001. http://www.microsoft.com/technet/security/bulletin/MS06-001.mspx.

[25] Mitre corporation. vulnerability type distributions in cve, may 2007. http://cwe.mitre.org/documents/vuln-trends/index.html.

[26] Multipot. http://labs.idefense.com/files/labs/releases/previews/multipot/index.html.

[27] MyDoom worm. http://www.symantec.com/security_response/writeup.jsp?docid=2004-012612-5422-99.

[28] Nepenthes. http://nepenthes.mwcollect.org/.

[29] Nmap. http://insecure.org/nmap/.

[30] Openvpn - an open source ssl vpn solution. *http://www.openvpn.net/*.

[31] Orkut. http://www.orkut.com.

[32] p0f. http://lcamtuf.coredump.cx/p0f.shtml.

[33] Project honey pot. http://projecthoneypot.org.

[34] Qemu data handling multiple command execution and denial of service vulnerabilities. *http://www.frsirt.com/english/advisories/2007/1597*.

[35] Quattor system administrator toolsuite. *http://quattor.web.cern.ch/quattor/*.

[36] Safari carpet bomb. http://www.dhanjani.com/blog/2008/05/safari-carpet-b.html.

[37] Samba. Available online at http://www.samba.org.

[38] Sebek homepage. *http://www.honeynet.org/tools/sebek/*.

[39] Shelia. http://www.cs.vu.nl/~herbertb/misc/shelia/.

[40] Technical explanation of the myspace worm, feb. 2006. http://web.archive.org/web/20060208182348/namb.la/popular/tech.html.

[41] The Protocol Informatics Project. http://www.4tphi.net/~awalters/PI/PI.html.

[42] Twitter targeted by xss worms. http://www.securityfocus.com/brief/945.

[43] Upnp forum.
http://www.upnp.org.

[44] User-mode linux. *http://user-mode-linux.sourceforge.net/*.

[45] Using honeyclients to Detect New Attacks. http://www.synacklabs.net/honeyclient/Wang-Honeyclient-ToorCon2005.pdf.

[46] Vmware homepage. *http://www.vmware.com/* .

[47] Vpns and public key infrastructure. *http://www.onlamp.com/pub/a/security/2004/09/23/vpns_and_pki.html*.

[48] The web application security consortium / distributed open proxy honeypots. http://projects.webappsec.org/Distributed-Open-Proxy-Honeypots.

[49] The xen virtual machine monitor. *http://www.cl.cam.ac.uk/Research/SRG/netos/xen/performance.html*.

[50] Xss worm strikes gaiaonline. http://blogs.securiteam.com/index.php/archives/786.

[51] Cert Advisory CA-2003-04: MS-SQL Server Worm.
http://www.cert.org/advisories/CA-2003-04.html, January 2003.

[52] The Honeynet Project, 2003. http://www.honeynet.org/.

[53] The Spread of the Sapphire/Slammer Worm. http://www.silicondefense.com/research/worms/slammer.php, February 2003.

[54] Mozilla Port Blocking. http://www.mozilla.org/projects/netlib/PortBanning.html, December 2004.

[55] PuppetNet Project Web Site. http://s3g.i2r.a-star.edu.sg/proj/puppetnets, September 2005.

[56] ABC Electronic. ABCE Database. http://www.abce.org.uk/cgi-bin/gen5?runprog=abce/abce&noc=y, 2006.

[57] P. Akritidis and E. P. Markatos. Efficient content-based detection of zero-day worms. In *Proceedings of the IEEE International Conference on Communications (ICC)*, May 2005.

[58] W. Alcorn. The cross-site scripting virus. http://www.bindshell.net/papers/xssv/xssv.html. Published: 27th September, 2005. Last Edited: 16th October 2005.

[59] Alexa Internet Inc. Global top 500. http://www.alexa.com/site/ds/top_500, 2006.

[60] K. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the* 14^{th} *Usenix Security Symposium*, Aug. 2005.

[61] K. G. Anagnostakis, E. P. Markatos, S. Antonatos, and M. Polychronakis. E^2xB: A domain-specific string matching algorithm for intrusion detection. In *Proceedings of the* 18^{th} *IFIP International Information Security Conference (IFIP/SEC)*, May 2003.

[62] S. Andersen and V. Abella. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 2: Network Protection Technologies. Microsoft TechNet, http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2netwk.mspx, November 2004.

[63] Anonymous. About the Alexa Toolbar and traffic monitoring service: How accurate is Alexa? http://www.mediacollege.com/internet/utilities/alexa/, 2004.

[64] S. Antonatos, M. Polychronakis, P. Akritidis, K. G. Anagnostakis, and E. P. Markatos. Piranha: Fast and memory-efficient pattern matching for intrusion detection. In *Proceedings of the* 20^{th} *IFIP International Information Security Conference (IFIP/SEC)*, June 2005.

[65] E. Athanasopoulos and S. Antonatos. Enhanced captchas: Using animation to tell humans and computers apart. In *Proceedings of the 10th IFIP Open Conference on Communications and Multimedia Security*, October 2006.

[66] M. Bailey, E. Cooke, F. Jahanian, J. Nazario, and D. Watson. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. In *Proceedings of The 12th Annual Network and Distributed System Security Symposium (NDSS)*, Feb. 2005.

[67] M. Bailey, E. Cooke, F. Jahanian, N. Provos, K. Rosaen, and D. Watson. Data Reduction for the Scalable Automated Analysis of Distributed Darknet Traffic. In *Proceedings of the Internet Measurement Conference (IMC) 2005*, 2005.

[68] M. Bailey, E. Cooke, F. Jahanian, D. Watson, and J. Nazario. The Blaster Worm: Then and Now. *IEEE Security & Privacy*, 3(4):26–31, July/August 2005.

[69] M. Bailey, E. Cooke, D. Watson, F. Jahanian, and N. Provos. A hybrid honeypot architecture for scalable network monitoring. In *CSE-TR-499-04*.

[70] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Visualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP '03)*, Oct. 2003.

[71] B. L. Barrett. Home of the webalizer. http://www.mrunix.net/webalizer, August 2005.

[72] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[73] V. Berk, G. Bakos, and R. Morris. Designing a framework for active worm detection on global networks. In *Proceedings of the IEEE International Workshop on Information Assurance*, March 2003.

[74] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). *RFC 1738*, Dec. 1994.

[75] J. Bethencourt, J. Franklin, and M. Vernon. Mapping Internet Sensors With Probe Response Attacks. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, pages 193–208, August 2005.

[76] K. Biba. Integrity considerations for secure computer systems. In *MITRE Technical Report TR-3153*, 1977.

[77] A. Bortz, D. Boneh, and P. Nandy. Exposing private information by timing web applications. In *Proceedings of the* 16^{th} *International World Wide Web Conference*, May 2007.

[78] F. Bouget and T. Holz. A pointillist approach for comparing honeypots. In *Proceedings of DIMVA 2005*, 2005.

[79] CERT. Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests. `http://www.cert.org/advisories/CA-2000-02.html`, February 2000.

[80] CERT. Advisory CA-2001-19: 'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. `http://www.cert.org/advisories/CA-2001-19.html`, July 2001.

[81] CERT. Vulnerability Note VU#476267: Standard HTML form implementation contains vulnerability allowing malicious user to access SMTP, NNTP, POP3, and other services via crafted HTML page. `http://www.kb.cert.org/vuls/id/476267`, August 2001.

[82] R. Chinchani and E. V. D. Berg. A fast static analysis approach to detect exploit code inside network flows. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2005.

[83] N. Chou, R. Ledesma, Y. Teraguchi, and J. Mitchell. Client-side defense against web-based identity theft. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04)*, February 2004.

[84] J. Claessens, B. Preneel, and J. Vandewalle. A tangled world wide web of security issues. *First Monday*, 7(3), March 2002.

[85] F. Cohen. Computer Viruses: Theory and Practice. *Computers & Security*, 6:22–35, February 1987.

[86] E. Cooke, F. Jahanian, and D. McPherson. The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In *Proceedings of the 1st USENIX Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI 2005)*, July 2005.

[87] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–147, 2005.

[88] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the* 7^{th} *USENIX Security Conference*, pages 63–78, jan 1998.

[89] J. Crandall, F. Chong, and S. Wu. Minos: Architectural Support for Protecting Control Data. In *Transactions on Architecture and Code Optimization (TACO). Volume 3, Issue 4*, Dec. 2006.

[90] T. T. Cymru. The team cymru darknet project. June 2004.

[91] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levin, and H. Owen. Honeystat: Local worm detection using honeypots. In *Proceedings of the Recent Advance in Intrusion Detection (RAID) Conference 2004*, September 2004.

[92] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *Proceedings of the* 13^{th} *Usenix Security Symposium*, Aug. 2004.

[93] M. Dornseif, T. Holz, and C. Klein. NoSEBrEaK - Attacking Honeynets. In *Proceedings of the 2004 Workshop on Information Assurance and Security*, June 2004.

[94] M. Dornseif, T. Holz, and C. Klein. Nosebreak - attacking honeynets. In *Proceedings of the 5th IEEE Information Assurance Workshop*, June 2004.

[95] D. Endler. The Evolution of Cross-Site Scripting Attacks. `http://www.cgisecurity.com/lib/XSS.pdf`.

[96] E. W. Felten, D. Balfanz, D. Dean, and D. S. Wallach. Web Spoofing: An Internet Con Game. In *Proceedings of the 20th National Information Systems Security Conference*, pages 95–103, October 1997.

[97] E. W. Felten and M. A. Schneider. Timing attacks on Web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS'00)*, pages 25–32, New York, NY, USA, 2000. ACM Press.

[98] P. Ferrie and F. Perriot. Mostly harmless. *Virus Bulletin*, pages 5–8, August 2004.

[99] J. J. Garrett. Ajax: A New Approach to Web Applications. `http://www.adaptivepath.com/publications/essays/archi-ves/000385.php`, February 2005.

[100] P. Gladychev, A. Patel, and D. O'Mahony. Cracking RC5 with Java applets. *Concurrency: Practice and Experience*, 10(11-13):1165–1171, 1998.

[101] J. Grossman and T. Niedzialkowski. Hacking intranet websites from the outside - javascript malware just got a lot more dangerous. Blackhat USA, August 2006.

[102] M. Healan. Referer spam. `http://www.spywareinfo.com/articles/referer_spam/`, Sept. 2003.

[103] A. Ho, , M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection using Demand Emulation. In *Proceedings of the EuroSys*, pages 39–58, Apr. 2006.

[104] W. Inc. Webtrends web analytics and web statistics. `http://www.webtrends.com`, 2006.

[105] S. Ioannidis and S. M. Bellovin. Building a Secure Browser. In *Proceedings of the Annual USENIX Technical Conference, Freenix Track*, June 2001.

[106] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from Web privacy attacks. In *Proceedings of the WWW Conference*, 2006.

[107] R. James, Z. Diego, and D. Yann. Building and deploying billy goat, a worm detection system. In *Proceedings of the 18th Annual FIRST Conference*, 2006.

[108] X. Jiang and D. Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. In *Proceedings of the* 13^{th} *USENIX Security Sumposium*, pages 15–28, August 2004.

[109] T. Jim, N. Swamy, and M. Hicks. Defeating scripting attacks with browser-enforced embedded policies. In *Proceedings of the 16th International World Wide Web Conference (WWW'07)*, May 2007.

[110] G. Keizer. Dutch botnet bigger than expected. `http://informationweek.com/story/showArticle.jhtml?articleID=172303265`, October 2005.

[111] J. O. Kephart. A Biologically Inspired Immune System for Computers. In *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 130–139. MIT Press, 1994.

[112] J. O. Kephart and S. R. White. Directed-graph epidemiological models of computer viruses. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 343–359, May 1991.

[113] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, pages 271–286, 2004.

[114] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000.

[115] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *ACM Transactions on Computer Systems 18(3)*, pages 263–297, Aug. 2000.

[116] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. SETI@home – Massively Distributed Computing for SETI. *Computing in Science & Enginering*, 3(1):78–83, 2001.

[117] C. Kreibich and J. Crowcroft. Honeycomb – creating intrusion detection signatures using honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, Nov. 2003.

[118] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2005.

[119] C. Kruegel and G. Vigna. Anomaly detection of Web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 251–261, New York, NY, USA, 2003. ACM Press.

[120] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis. Puppetnets: Misusing web browsers as a distributed attack infrastructure (extended version). Technical Report, http://s3g.i2r.a-star.edu.sg/proj/puppetnets, August 2006.

[121] C. Leita, K. Mermoud, and M. Dacier. Scriptgen: an automated script generation tool for honeyd. In *21st Annual Computer Security Applications Conference*, December 2005.

[122] J. Li, T. Ehrenkranz, G. Kuenning, and P. Reiher. Simulation and analysis on the resiliency and efficiency of malnets. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation (PADS'05)*, pages 262–269, Washington, DC, USA, 2005. IEEE Computer Society.

[123] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of the IEEE Symposium on Security & Privacy*, pages 32–47, 2006.

[124] J. D. C. Little. A Proof of the Queueing Formula $L = \lambda W$. *Operations Research*, (9):383–387, 1961.

[125] G. Maone. Firefox add-ons: Noscript. `https://addons.mozilla.org/firefox/722/`, May 2006.

[126] E. P. Markatos, S. Antonatos, M. Polychronakis, and K. G. Anagnostakis. ExB: Exclusion-based signature matching for intrusion detection. In *Proceedings of the IASTED International Conference on Communications and Computer Networks (CCN)*, pages 146–152, November 2002.

[127] S. McCanne, C. Leres, and V. Jacobson. Libpcap, 2006. `http://www.tcpdump.org/`.

[128] D. Moniz and H. Moore. Six degrees of xssploitation. Blackhat USA, August 2006.

[129] D. Moore, C. Shanning, and K. Claffy. Code-Red: a case study on the spread and victims of an Internet worm. In *Proceedings of the* 2^{nd} *Internet Measurement Workshop (IMW)*, pages 273–284, November 2002.

[130] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial-of-Service Activity. In *Proceedings of the* 10^{th} *USENIX Security Symposium*, pages 9–22, August 2001.

[131] Mozilla.org. End User Guide: Automatic Proxy Configuration (PAC). `http://www.mozilla.org/catalog/end-user/customizing/enduserPAC.html`, August 2004.

[132] C. Nachenberg. Computer virus-antivirus coevolution. *Commun. ACM*, 40(1):46–51, 1997.

[133] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of the IEEE Security & Privacy Symposium*, pages 226–241, May 2005.

[134] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the* 12^{th} *Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[135] M. Overton. Worm charming: taking smb lure to the next level. *Virus Bulletin Conference*, 2003.

[136] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of Internet background radiation. In *Proceedings of the* 4^{th} *ACOM SIGCOMM conference on Internet measurement*, pages 27–40, 2004.

[137] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[138] V. Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *ACM Computer Communication Review*, 31(3):38–47, 2001.

[139] Philippine Honeynet Project. Philippine Internet Security Monitor - First Quarter of 2006. `http://www.philippinehoneynet.org/docs/PISM20061Q.pdf`

.

[140] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, July 2006.

[141] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proceedings of ACM SIGOPS Eurosys 2006*, April 2006.

[142] N. Provos. A virtual honeypot framework. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 1–14, Aug. 2004.

[143] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Nov 2006.

[144] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[145] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of USENIX LISA*, November 1999. (software available from *http://www.snort.org/*).

[146] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the 14th Usenix Security Symposium*, 2005.

[147] A. D. Rubin and D. E. G. Jr. A Survey of Web Security. *IEEE Computer*, 31(9):34–41, 1998.

[148] J. Ruderman. The Same Origin Policy. `http://www.mozilla.org/projects/security/components/same-origin.html`, August 2001.

[149] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*, 2002.

[150] B. Schneier. Attack trends 2004 and 2005. *ACM Queue*, 3(5), June 2005.

[151] C. Shannon and D. Moore. The Spread of the Witty Worm. *IEEE Security & Privacy*, 2(4):46–50, July/August 2004.

[152] Y. Shinoda, K. Ikai, and M. Itoh. Vulnerabilities of Passive Internet Threat Monitors. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, pages 209–224, August 2005.

[153] S. Sidiroglou, G. Giovanidis, and A. Keromytis. A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. In *Proceedings of the 8th Information Security Conference (ISC)*, pages 1–15, Sept. 2005.

[154] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI)*, Dec. 2004.

[155] S. Sinha, M. Bailey, and F. Jahanian. Shedding Light on the Configuration of Dark Addresses. In *Proceedings of the* 14^{th} *Network and Distributed System Security Sympoisum (NDSS)*, Feb. 2007.

[156] A. Smith and J. Fox. RandomNet. In *http://www.citi.umich.edu/u/provos/honeyd/ch01-results/3/*, March 2003.

[157] F. Smith, J. Aikat, J. Kapur, and K. Jeffay. Variability in TCP round-trip times. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet measurement*, 2003.

[158] S. Stamm, Z. Ramzan, and M. Jakobson. Drive-by pharming. In *Technical Report TR641, Indiana University Department of Computer Science*, Dec. 2006.

[159] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The Top Speed of Flash Worms. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 33–42, October 2004.

[160] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The top speed of flash worms. In *Proc. ACM WORM*, Oct. 2004.

[161] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The top speed of flash worms. In *Proceedings of the Second ACM Workshop on Rapid Malcode (WORM)*, pages 33–42, 2004.

[162] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, Aug. 2002.

[163] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, pages 149–167, August 2002.

[164] Stunnix. Stunnix javascript obfuscator - obfuscate javascript source code. `http://www.stunnix.com/prod/jo/overview.shtml`, 2006.

[165] Symantec. Internet Threat Report: Trends for January 05-June 05. Volume VIII. Available from www.symantec.com, September 2005.

[166] C. Taylor. Attack of the world wide worms. *TIME*, 2003.

[167] TechWeb.com. Lycos strikes back at spammers with dos screensaver. `http://www.techweb.com/wire/security/54201269`, 2004.

[168] The Honeynet Project. Know your enemy: Tracking botnets. `http://www.honeynet.org/papers/bots/`, March 2005.

[169] J. Topf. HTML Form Protocol Attack. `http://www.remote.org/jochen/sec/hfpa/`, August 2001.

[170] N. Vanderavero, X. Brouckaert, O. Bonaventure, and B. Charlier. The honeytank : a scalable approach to collect malicious internet traffic. In *Proceedings of the International Infrastructure Survivability Workshop (IISW'04)*, Dec. 2004.

[171] VNExpress Electronic Newspaper. Website of largest Vietnamese hacker group attacked by DDoS. `http://vnexpress.net/Vietnam/Vi-tinh/2005/12/3B9E4A6D/`, December 2005.

[172] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS '07)*, February 2007.

[173] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP)*, pages 148–162, 2005.

[174] D. Wang. HOWTO: ISAPI Filter which rejects requests from SF_NOTIFY_PREPROC_HEADERS based on HTTP Referer. `http://blogs.msdn.com/david.wang`, July 2005.

[175] J. Wang, I. Hamadeh, G. Kesidis, and D. J. Miller. Polymorphic Worm Detection and Defense: System Design, Experimental Methodology, and Data Resources. In *Proceedings of the* 1^{st} *Workshop on Large-Scale Attack Defence (LSAD)*, pages 169–176, September 2006.

[176] K. Wang and S. J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the* 7^{th} *International Symposium on Recent Advanced in Intrusion Detection (RAID)*, pages 201–222, September 2004.

[177] Y. Wang and C. Wang. Modeling timing parameters for virus propagation on the internet. In *Proceeding of the 1st Workshop Of Rapid Malcode (WORM'03)*, Oct 2003.

[178] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. Kin. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS '06)*, February 2006.

[179] N. Weaver. Warhol worms: The potential for very fast internet plagues. *Technical Report*, 2002.

[180] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 29–44, August 2004.

[181] A. T. Williams and J. Heiser. Protect your PCs and Servers From the Bothet Threat. Gartner Research, ID Number: G00124737, December 2004.

[182] V. Yegneswaran, P. Barford, and D. Plonka. On the design and utility of internet sinks for network abuse monitoring. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.

[183] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, August 2005.

[184] zone-h. Digital attacks archive. `http://www.zone-h.org/en/defacements/`, 2006.

[185] C. C. Zou, W. Gong, and D. Towsley. Code Red Worm Propagation Modeling and Analysis. In *Proceedings of the* 9^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 138–147, November 2002.

[186] O. Zurutuza. *Data Mining Approaches for Analysis of Worm Activity Toward Automatic Signature Generation.* PhD thesis, Mondragon University, November 2007.