Testing Network-based Intrusion Detection Signatures Using Mutant Exploits

Giovanni Vigna
vigna@cs.ucsb.edu

William Robertson
wkr@cs.ucsb.edu

Davide Balzarotti
balzarot@cs.ucsb.edu

Reliable Software Group
University of California, Santa Barbara
Santa Barbara, CA 93106

ABSTRACT

Misuse-based intrusion detection systems rely on models of attacks to identify the manifestation of intrusive behavior. Therefore, 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 (or good signatures) 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. For this reason, it would be beneficial to have testing tools that are able to evaluate the "goodness" of detection signatures. This work describes a technique to test and evaluate misuse detection models in the case of network-based intrusion detection systems. The testing technique is based on a mechanism that generates a large number of variations of an exploit by applying mutant operators to an exploit template. These mutant exploits are then run against a victim host protected by a network-based intrusion detection system. The results of the systems in detecting these variations provide a quantitative basis for the evaluation of the quality of the corresponding detection model.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Network**]: Security and Protection

General Terms

Security

Keywords

Security Testing, Intrusion Detection, Quality Metrics

CCS'04, October 25–29, 2004, Washington, DC, USA.

1. INTRODUCTION

Intrusion detection systems analyze one or more streams of events looking for the manifestations of attacks. For example, network-based intrusion detection systems (NIDSs) analyze network packets, while host-based intrusion detection systems (HIDSs) analyze audit data collected by an operating system about the actions performed by users and applications.

The analysis of the event streams can be performed according to different approaches. A common classification of approaches divides them into misuse detection approaches and anomaly detection approaches. In the first case, the analysis relies on models that represent intrusive behavior. The analysis process tries to identify series of events that conform to these models and, therefore, represent an intrusion. In the second case, the analysis uses models that characterize the normal behavior of a system and aims at identifying events that do not fit the established models, in the assumption that anomalous behavior is often evidence of malicious intent.

Network-based intrusion detection systems based on misuse detection approaches are the most widely-deployed type of intrusion detection systems. For example, Snort [28] and ISS's RealSecure [11], which represent the leading products in the open-source and commercial worlds, respectively, are both network-based misuse detection systems.

One problem with misuse detection systems is that their ability 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 the modeled attack without making mistakes. In technical terms, a perfect model would produce a 100% detection rate with 0 false alarms (also called *false positives*).

Unfortunately, writing good models (or good signatures) is hard and resource-intensive. Attacks that exploit a certain vulnerability may do so in completely different ways. This problem could easily be solved by writing a model for each way in which the vulnerability can be exploited, or, even better, by creating a model which is abstract enough to capture all the different variations of an attack. For example, the work in [10] suggests that good models should consider only those events that if removed from the attack would make the attack unsuccessful. Unfortunately, this is not always possible, and sometimes by creating an abstract signature it is possible to undermine its detection precision (i.e., the model would also flag as intrusive perfectly normal activity). Also, the security expertise of the signature developer may have

a notable impact on the ability of the model to characterize the attack correctly.

If the models used in intrusion detection are known, it is 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 models they use to detect intrusions. Even in the cases when these models are available, it is extremely time consuming to devise testing procedures that analyze the models and identify blind spots.

This work describes a technique to test and evaluate misuse detection models in the case of network-based intrusion detection systems. The testing technique is based on an automated mechanism to generate a large number of variations of an exploit by applying mutant operators to an exploit template. The mutant exploits are then run against a victim system where the vulnerable applications and/or operating systems are installed. The attacks are analyzed by a network-based intrusion detection system. The intrusion alerts produced by the NIDS are then correlated with the execution of the mutant exploits. 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.

Obviously, this technique does not provide a formal evaluation of the "goodness" of an attack model. Nonetheless, we claim that this is a valid way to improve one's confidence in the generality of a detection model. Note that the technique could be easily extended to host-based intrusion detection systems and to systems that use anomaly detection approaches. Nonetheless, hereinafter we will limit the scope of our analysis to network-based misuse detection systems.

The mutation process is deterministic and guided by a seed value, which makes the mutations reproducible. 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.

To address this issue, the technique relies 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 used to determine if the exploit was successful. However, in some cases it is not possible to generate evidence of the effectiveness of an attack as part of its execution, and, for those cases, an external oracle that reports on the outcome of specific attacks has to be developed.

We have developed a tool based on our testing technique and used it to evaluate two popular network-based intrusion detection systems, namely Snort 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 rest of paper is structured as follows. Section 2 presents some related work. Section 3 presents the set of mutation operators. Section 4 describes the mutant exploit generation process. Then, Section 5 presents the results of the application of our technique to the network-based misuse detection systems under test. Finally, Section 6 concludes and outlines future work.

2. RELATED WORK

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

A class of intrusion detection evaluation efforts have 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 are launched against hosts in the testbed [14, 5, 4, 8]. 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 [15].

Unfortunately, testing and comparing intrusion detection systems is difficult because different systems have different operational environments and may employ a variety of techniques for producing alerts corresponding to attacks [23, 26]. 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 paper we are concerned with the black-box evaluation of the signatures of network-based intrusion detection systems. This is a complementary approach with respect to our previous research on using IDS stimulators (e.g., Mucus [17], Snot [30], Stick [7], and IDSwakeup [2]) to perform cross-testing of network-based signatures. In particular, in [17] we 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). These test cases were then analyzed using a different network-based intrusion detection system. 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. 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 testing technique discussed in this paper attempts to overcome this limitation by relying on exploits to generate test cases instead of using intrusion detection signatures. The test cases are obtained by applying evasion techniques to an exploit pattern. These techniques may operate at the network level [1, 24], at the application level [27], or at the exploit level [16]. One advantage is that many exploits are publicly available and can be converted into exploit patterns with limited effort.

A similar approach was followed by the Thor tool [9], which was developed to perform testing of intrusion detection systems using variations of attacks. However, Thor's implementation is very limited and it includes only one evasion technique, namely IP fragmentation. This technique has the advantage that it can be independently applied to an exploit without having to modify the exploit source code because it operates at the network level. Unfortunately, this also means

that a system that performs defragmentation correctly is able to detect all instances of these variations.

Our approach supports multiple evasion techniques and allows the developer of the test exploit to compose these techniques to achieve a wide range of mutations. Compared to Thor, our system also supports a more sophisticated verification mechanism to test the effectiveness of the modified attacks. Note that our approach neither claims to completely cover the space of possible variations of an attack, nor states that it guarantees that the variations of attacks are successful. Nevertheless, it provides an effective framework for the composition of evasion techniques to test the quality of intrusion detection signatures.

The testing technique we propose is similar to the fault injection approach. Software fault injection [6, 34, 12] is a testing methodology that aims at evaluating the dependability of a software system by analyzing its behavior in the 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 [3]. This approach is supported by "fuzzer" tools, such as *Sharefuzz* [29] and *Spike* [31]. Fuzzers are 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, the Spike tool provides an API that allows one to easily model an arbitrary network protocol and then generate traffic that contains different ranges of values for each field of the messages used in the protocol.

Our approach differs from traditional fault injection 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 software fault injection 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 simply 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 "mutation testing." Mutation testing [25] 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.

3. MUTATION MECHANISMS

Exploit mutation is a general term comprising a broad range of techniques to modify and obfuscate an attack against a vulnerable service. Mutation techniques can operate at several layers, the most significant of which are the network layer, the application layer, and the exploit layer. The following sections discuss the set of techniques that were utilized in the creation of our exploit mutation engine.

3.1 Network Layer Mutations

Network layer mutations are a class of techniques that operate at the network or transport layers of the classic OSI networking model. These mutations 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 [24], they remain effective in evading current NIDS implementations.

3.1.1 IPv6

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 [21]. 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.

3.1.2 IP Packet Splitting

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.

3.2 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.

3.2.1 Protocol Rounds

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 implemen-

tation by conducting a benign initial round of the protocol before launching the actual attack.

3.2.2 FTP Evasion Techniques

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's SideStep [27] to evade NIDSs. 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.

3.2.3 HTTP Evasion Techniques

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 [22]. 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 that will be served by a web server might be discarded by the NIDS. 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.

3.2.4 SSL NULL Record Evasion Technique

The Secure Sockets Layer (SSL) is a protocol developed by Netscape to provide a private, authenticated, and reliable communications channel for networked applications [19]. The specification defines a set of messages (e.g., `client-hello`, `server-hello`, `client-master-key`, `server-verify`); these messages are encapsulated in objects known as "records." SSL records are composed of both a header and 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. Thus, a malicious client could modify a valid session handshake, described in Figure 1, as shown in Figure 2.

```
client-hello      C -> S: challenge,cipher_specs
server-hello      S -> C: connection-id,server_certificate,
                          cipher_specs
client-master-key C -> S: {master_key}server_public_key
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 1: Valid SSLv2 session negotiation example.

This handshake is illegal according to the 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

```
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 2: SSLv2 session negotiation NULL record evasion.

of real-world implementations, because they will discard the monitored traffic as an invalid SSL handshake.

3.3 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.

3.3.1 Polymorphic Shellcode

The ADMmutate polymorphic shellcode engine is used to generate self-decrypting exploit payloads that will defeat most popular NIDS shellcode detectors [13]. Features of this engine include XOR-encoded payloads with 16-bit sliding keys, randomized NOP generation, support for banned characters, to{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.

3.3.2 Alternate Encodings

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.

4. MUTANT GENERATION

In order to test the effectiveness of NIDSs against mutant exploits, we developed a framework for automatically generating large numbers of exploit variations. The framework, written in the Python scripting language, consists of a set of exploit templates, a set of mutation mechanisms, and a mutation engine. An architectural overview of the framework is depicted in Figure 3.

Exploit templates are realized as Python classes that implement a generic exploit interface. This interface, utilized by the mutation engine, allows each exploit to perform the necessary setup and teardown for each exploit attempt (e.g., restart a target service, remove artifacts from previous exploitation, etc.), perform the actual exploit, and determine if the attack was successful. Each attempt requires a target

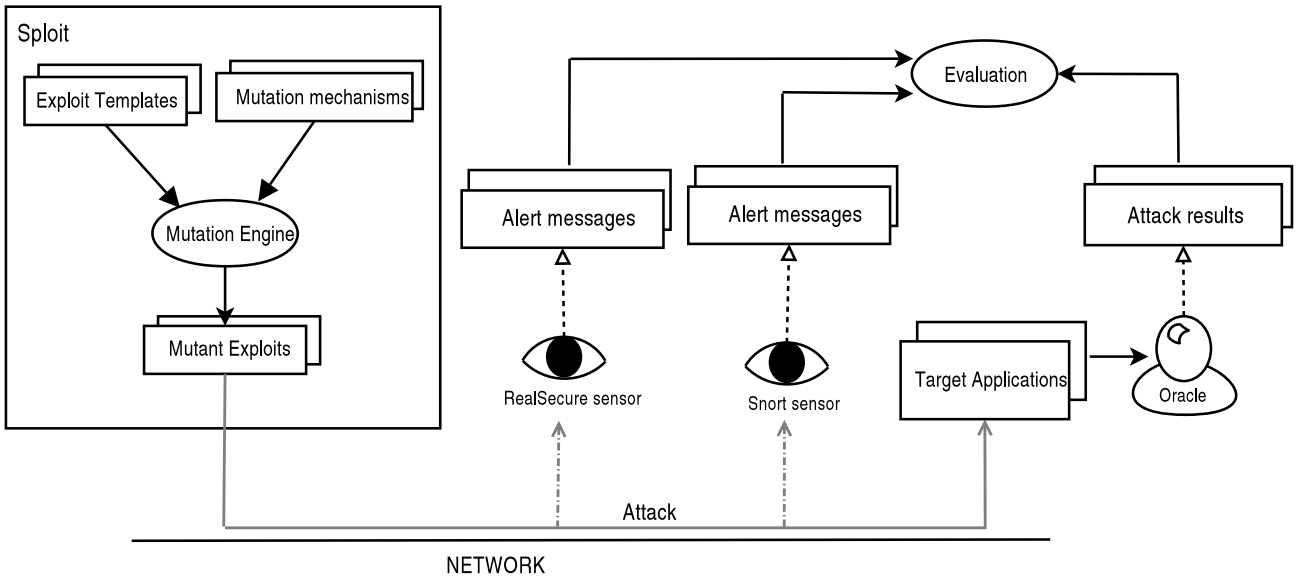

Figure 3: Exploit mutation framework.

and a random seed value provided by the engine, the use of which is explained below.

The success of an exploit attempt is determined through one of several methods, depending on the suitability of a method in relation to a given exploit. One commonly used means 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 create a file in a known location. A subsequent check over an auxiliary channel for the existence of the file is then made after an exploit attempt to determine if the exploit was successful.

Each exploit template implements a baseline exploit for a specific vulnerability and utilizes a subset of the mutation operators provided by the framework to generate mutant attacks. These mutation mechanisms implement a generic interface composed of a single method, "`mutate`," that takes an exploit or a component of an exploit, applies a transformation to it, and returns the result. Mutation methods are further parameterized by a random seed, provided by the mutation engine, which influences the behavior of the mutator in some specific and repeatable way. For instance, the SSL session mutation mechanism varies the location and number of SSL `NULL` records inserted into a session, as described in Section 3.2.4, based on the provided seed.

Each operator, regardless of the seed value, must be constrained to generating mutations that preserve the semantics and effectiveness of the baseline exploit. By applying a composition of mutation operators to an exploit template with a single seed, a random – yet repeatable – mutant exploit can be generated from the exploit template.

The final component of the framework is the mutation engine, which is responsible for generating mutant exploits by instantiating an exploit template with a set of random seeds, executing the resulting mutants against a chosen target, and determining the results of the execution of each exploit.

5. EVALUATION

In order to test the detection capabilities of commonly deployed NIDSs as well as the effectiveness of the exploit mutation framework, an evaluation of the testing tool against a set of vulnerable services monitored by two popular network-based intrusion detection systems was conducted. The goal of the evaluation was to determine whether the described exploit mutation framework is capable of automatically generating exploits that can evade today's most advanced NIDSs. As a byproduct, the results of the execution of the mutant exploit provides an insight on the ability of the tested NIDSs to detect variations of attacks.

The evaluation took place on an isolated testbed composed of an attacking host running RedHat Linux 9.0, a sensor host running Snort and ISS's RealSecure on RedHat Linux 7.3, and a set of target hosts with various operating systems, including OpenBSD 3.1, RedHat Linux 6.2, RedHat Linux 7.3, RedHat Linux 9.0, and Windows 2000 Server. A variety of vulnerable services were installed on these target hosts.

5.1 Exploits

The exploits used in this evaluation were selected for 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 one recently performed by the Neohapsis OSEC Project [18]. 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-ftpd 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_<machine-name>` 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 popular open-source web server that features full compliance with the HTTP/1.1 specification [32]. Apache versions below 1.3.24 and 2.0.38 are vulnerable to an overflow in the handling of the chunked-encoding transfer mechanism, as described in CVE-2002-0392. HTTP chunk encoding is described in the HTTP/1.1 specification as a specific form of encoding for HTTP requests and replies.

In general, transfer encoding values are used to indicate 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 4.

```
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 4: 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

Exploit	Baseline attack	Mutated Attack	Evasion Techniques
WU-ftpd Remote Format String Stack Overwrite	Detected	*Evaded*	Telnet control sequences Shellcode mutation IP packet splitting
WU-imapd Remote Buffer Overflow	Detected	*Evaded*	Prepend zeros to numbers Shellcode mutation
IIS Escaped Characters Double Decoding	Detected	Detected	
Microsoft DCOM-RPC	Detected	Detected	
IIS Extended Unicode Directory Traversal	Detected	*Evaded*	URL encoding
NSIISlog.DLL Remote Buffer Overflow	Detected	Detected	
IIS 5.0 .printer ISAPI Extension Buffer Overflow	Detected	Detected	
WS-FTP Server `STAT` Buffer Overflow	Detected	*Evaded*	Telnet control sequences IP packet splitting
OpenSSL SSLv2 Client Master Key Overflow	Detected	*Evaded*	SSL `NULL` record insertion
Apache HTTP Chunked Encoding Overflow	Detected	*Evaded*	HTTP protocol evasion

Table 1: Evaluation Results for Snort.

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 implementation of the Secure Sockets Layer (SSLv2/v3) and Transport Layer Security (TLSv1) protocols [33]. OpenSSL versions below 0.9.6e and 0.9.7beta3 are vulnerable to an overflow in the handling of SSLv2 client master keys, as described in CAN-2002-0656. Client master keys are generated by the client during the handshake procedure of the SSL protocol, and are used to derive the session keys that encrypt data transmitted over the secured connection (a typical SSL handshake was shown in Figure 1). 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.

5.2 Intrusion Detection Systems

For our test we focused on two different NIDSs: Snort and ISS's RealSecure. Snort [28] is a lightweight NIDS released under the GNU GPL license and available both for Windows and Linux. Its attack detection process relies on a large and publicly available set of signatures and a set of preprocessors that handle functions such as protocol decoding and normalization, packet reassembly, and portscan detection. For this evaluation, Snort v2.1.2 was deployed on our testbed with all preprocessors enabled with their default settings and all available signatures loaded.

ISS's RealSecure [11] is one of the most popular commercial NIDSs. The attack detection process utilizes a sophisticated set of precompiled closed-source rules. For this evaluation, a RealSecure 10/100 network sensor v7.0 and SiteProtector management console were deployed on our testbed with all signatures enabled.

These two systems were chosen based on several factors. First, Snort and RealSecure are generally regarded as the premier NIDSs in the open-source and closed-source worlds, respectively, and as such enjoy wide deployment across the Internet. Also, in many previous evaluations both systems have been shown to possess excellent attack detection capabilities and to be able to correctly manage many different types of evasion, obfuscation, and anti-IDS techniques.

5.3 Experiments

The evaluation was conducted by using our mutation engine to generate a number of mutant exploits for each baseline exploit. These mutant exploits were then launched against the target hosts in an otherwise silent network. The success of each mutant exploit was then determined using an oracle, and unsuccessful attempts were removed from the evaluation. Finally, the alerts generated by the deployed NIDS sensors were gathered and correlated with the corresponding exploit attempt.

The experiment focused on providing some useful indication about the average quality of the signatures shipped with the NIDSs under test. It was not our intention to perform a complete evaluation of all possible characteristics of these NIDSs. For this reason, we did not take into consideration properties such as the number of false positive alerts, the behavior of the systems when run with background traffic, or their correlation and reporting capabilities.

5.4 Results

Tables 1 and 2 present the evaluation results for Snort and RealSecure, respectively. For each attack, we report several values. The first represents the ability of the intrusion detection system to correctly detect the baseline attack when the exploit was not subjected to any mutation technique. The second reports whether the IDS was able to detect all of the mutations of the same attack attempted during the experiment. In the last column we summarize the key techniques that enabled the mutated exploits to evade detection, when applicable.

The total number of possible mutants that the engine can generate 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 the way in which each technique is configured. For instance, an application-level transformation that consists of modifying the number of space characters between the HTTP method (e.g., `GET` or `POST`) and the requested URL can generate a large number of mutants, one for each number of space characters selected. When composed with other techniques, this operator may lead to an unmanageable number of mutant exploits. There-

Exploit	Baseline attack	Mutated Attack	Evasion Techniques
WU-ftpd Remote Format String Stack Overwrite	Detected	*Evaded*	Telnet control sequences Shellcode mutation
WU-imapd Remote Buffer Overflow	Detected	*Evaded*	CR character between the tag and the command
IIS Escaped Characters Double Decoding	Detected	*Evaded*	CR character before the request command
Microsoft DCOM-RPC	Detected	Detected	
IIS Extended Unicode Directory Traversal	Detected	*Evaded*	CR character before the request command
NSIISlog.DLL Remote Buffer Overflow	Detected	*Evaded*	CR character before the request command
IIS 5.0 .printer ISAPI Extension Buffer Overflow	Detected	*Evaded*	CR character before the request command
WS-FTP Server `STAT` Buffer Overflow	Detected	*Evaded*	Telnet control sequence
OpenSSL SSLv2 Client Master Key Overflow	Detected	*Evaded*	SSL `NULL` record insertion
Apache HTTP Chunked Encoding Overflow	Detected	*Evaded*	HTTP protocol evasion

Table 2: Evaluation Results for IIS RealSecure.

fore, it is necessary to configure the mutant operators to produce mutants within a reasonable range of variability.

For our tests, we configured the engine to generate from a minimum of about a hundred mutations (e.g., in the case of the FTP-related attacks) to a maximum of several hundred thousand (e.g., in the case of the HTTP-related attacks). 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 lower than the number of possible mutations of an exploit.

As Table 1 shows, Snort correctly detected all instances of the baseline attacks. The exploit mutation engine, however, was able to automatically generate mutated exploits that evaded Snort's detection engine for 6 of the 10 attacks. Table 2 shows that RealSecure, similarly, was able to detect all instances of the baseline attacks. In this case, however, the exploit mutation engine was able to generate mutant exploits that evaded detection by RealSecure in 9 out of 10 cases. Even though it is tempting to make relative comparisons between the two 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 both sensors proved to be surprisingly vulnerable to the generated mutant exploits.

It is worth noting that most of these techniques are well-known and thus one would expect that the mutants should be correctly detected by both Snort and RealSecure. The results demonstrate, however, that NIDSs remain vulnerable to variations based on these classic mutation techniques. Consider as an example the evasion technique that relies on inserting telnet control sequences in an FTP command stream; this approach has been used by the SideStep IDS evasion tool since 2000. Both of the tested 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 NIDSs. 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.

In the case of the IMAP attacks, the evasion 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 prefix-quoted with a byte count between curly brackets, followed by a `CR-LF`. An example of a legal IMAP login is shown in Figure 5.

```
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 5: 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 the HTTP attacks, the RealSecure analyzer is deceived by some non-standard characters in the request. In this case, it is sufficient to insert a `CR` before the command. With Snort, the known techniques of encoding malicious URLs still seem to be effective, as shown by its inability to detect variations of the directory traversal attack.

All of these techniques are 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 techniques 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.

Also of note is the relative effectiveness of our automated approach as opposed to manual efforts such as the recent IDS evaluation by NSS (4^{th} edition) [20]. 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.

6. CONCLUSIONS AND FUTURE WORK

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 exist one or more "blind spots" in the attack model.

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 paper presented a technique for the black-box testing of network-based signatures and a tool based on the technique. The tool takes exploit templates and generates exploit mutants. These mutants 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 ten common exploits and used the test cases against two 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 the tool does not guarantee complete coverage of the possible mutation space, the tool is useful in gaining assurance about the quality of the signatures of an intrusion detection system.

Future work will focus on extending the framework for the description of the mutation process to include mutations that are conditionally applied according to the results of previous mutant operators, i.e., to support the concurrent application of co-dependent mutation techniques in an automated way. Another possible future direction (suggested by one of the reviewers) is to use our mutation approach to evaluate the amount of false positives generated by a signature. This would allow one to evaluate another important aspect of signature quality.

Acknowledgments

This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484 and by the National Science Foundation under grants CCR-0209065 and CCR-0238492. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation, the Army Research Office, or the U.S. Government.

7. REFERENCES

[1] Anzen. nidsbench: A Network Intrusion Detection System Test Suite. `http://packetstorm.widexs.nl/-UNIX/IDS/nidsbench/`, 1999.

[2] S. Aubert. Idswakeup. `http://www.hsc.fr/-ressources/outils/idswakeup/`, 2000.

[3] W. Du and A. P. Mathur. Vulnerability Testing of Software System Using Fault Injection. Technical Report, COAST, Purdue University, West Lafayette, IN, US, April 1998.

[4] 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.

[5] R. Durst, T. Champion, B. Witten, E. Miller, and L. Spagnuolo. Testing and Evaluating Computer Intrusion Detection Systems. *CACM*, 42(7):53–61, July 1999.

[6] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins, and D. Powell. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, 1990.

[7] C. Giovanni. Fun with Packets: Designing a Stick. `http://www.eurocompton.net/stick/`, 2002.

[8] 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.

[9] R. Marty. Thor: A Tool to Test Intrusion Detection Systems by Variations of Attacks. ETH Zurich Diploma Thesis, March 2002.

[10] 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.

[11] ISS. Realsecure 7.0. `http://www.iss.net/`, 2004.

[12] D. Pradhan J. Clark. Fault Injection: A Method For Validating Computer-System Dependability. *IEEE Computer*, 28(6):47–56, 1995.

[13] K2. ADMmutate. `http://www.ktwo.ca/security.html`, 2004.

[14] 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.

[15] 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.

[16] Metasploit Project. Metasploit. `http://www.metasploit.com/`, 2004.

[17] 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.

[18] Neohapsis OSEC Project. Neohapsis OSEC. `http://osec.neohapsis.com/`, 2004.

[19] Netscape Communications Corporation. SSL 2.0 Protocol Specification. `http://wp.netscape.com/eng/security/SSL_2.html`, 1995.

[20] Network Security Services Group. NSS IDS Evaluation (4^{th} Edition). `http://www.nss.co.uk/ips`, 2004.

[21] Network Working Group. Internet Protocol, Version 6 (IPv6) Specification. `http://www.faqs.org/rfcs/rfc2460.html`, 1998.

[22] Network Working Group. Hypertext Transfer Protocol – HTTP/1.1. `http://www.w3.org/Protocols/-rfc2616/rfc2616.html`, 1999.

[23] Next Generation Software Security Ltd. NGSS Evaluation. `http://www.nextgenss.com/`, 2004.

[24] T.H. Ptacek and T.N. Newsham. Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection. Technical Report, Secure Networks, January 1998.

[25] 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.

[26] M. Ranum. Experience Benchmarking Intrusion Detection Systems. NFR Security White Paper, December 2001.

[27] R. Graham. SideStep. `http://www.robertgraham.com/tmp/sidestep.html`, 2004.

[28] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX LISA '99 Conference*, November 1999.

[29] D. Aitel. Sharefuzz. `http://www.atstake.com/research/tools/-vulnerability_scanning/`, 2004.

[30] Sniph. Snot. http://www.sec33.com/sniph/, 2001.

[31] D. Aitel. Spike. `http://www.immunitysec.com/-resources-freesoftware.shtml`, 2004.

[32] The Apache HTTP Server Project. Apache HTTP Server. `http://httpd.apache.org/`, 2004.

[33] The OpenSSL Project. OpenSSL. `http://www.openssl.org/`, 2004.

[34] J. Voas, G. McGraw, L. Kassab, and L. Voas. A 'Crystal Ball' for Software Liability. *IEEE Computer*, 30(6):29–36, 1997.