Reverse Engineering of Network Signatures

Darren Mutz Christopher Kruegel[1] William Robertson
Giovanni Vigna Richard A. Kemmerer
Reliable Software Group
University of California, Santa Barbara
Email: {dhm,chris,wkr,vigna,kemm}@cs.ucsb.edu

Abstract

Network-based intrusion detection systems analyze network traffic looking for evidence of attacks. The analysis is usually performed using *signatures*, which are rules that describe what traffic should be considered as malicious. If the signatures are known, it is possible to either craft an attack to avoid detection or to send synthetic traffic that will match the signature to over-stimulate the network sensor causing a denial of service attack. To prevent these attacks, commercial systems usually do not publish their signature sets and their analysis algorithms. This paper describes a reverse engineering process and a reverse engineering tool that are used to analyze the way signatures are matched by network-based intrusion detection systems. The results of the analysis are used to either generate variations of attacks that evade detection or produce non-malicious traffic that over-stimulates the sensor. This shows that security through obscurity does not work. That is, keeping the signatures secret does not necessarily increase the resistance of a system to evasion and over-stimulation attacks.
Keywords: *Evasion, Detection Signatures, Network-based Intrusion Detection, Reverse Engineering.*

I. Introduction

Network-based intrusion detection systems (NIDSs) analyze the contents of network traffic to find evidence that malicious activity is occurring. The analysis is performed using different techniques, which can be classified into anomaly detection techniques and misuse detection techniques.

Anomaly detection techniques rely on models of expected, benign behavior of both the applications (e.g., in terms of the protocol being used) and the network (e.g., in terms of the type and amount of traffic exchanged). These models may be generated manually, derived automatically from application source code, created as the results of protocol analysis, or can be learned by observing the network during a training phase.

Misuse detection techniques take a complementary approach and rely on models of malicious behavior to identify instances of attacks in network traffic. Similar to the previous set of techniques, the models may be written manually, derived from specifications, or learned from sample input.

The most common form of attack model is a manually written *signature*. A signature is a set of rules that, when applied to an input stream, will match every instance of the attack modeled by the signature. Signatures can be applied to input events using different matching models (e.g., stateless or stateful models) and can be expressed in different languages (e.g., regular expressions or predicates).

Although the use of signatures requires continuous updating of the signature set and makes intrusion detection systems somewhat ineffective against novel attacks, all of the most popular NIDSs rely on signatures. This is the case for both open-source systems such as Snort [18], Bro [15], and NetSTAT [23] as well as closed-source systems such as ISS' RealSecure [10], Symantec's ManHunt [22], and NFR [17].

Knowing the set of signatures used by a network-based intrusion detection system gives the attacker two advantages: it allows the attacker (i) to devise ways to evade detection by crafting attacks in a way that will not be matched by a signature and (ii) to perform over-stimulation attacks where synthetic traffic is sent to the intrusion detection system to cause an excessive amount of alerts. This effectively results in a denial of service attack against the NIDS administrator, who quickly becomes desensitized to alerts

[1]Christopher Kruegel is currently with the Technical University of Vienna.

issued by the system. In addition, because developing signatures is a time-consuming, expertise-intensive process, commercial systems often do not disclose their signatures to prevent competitors from copying them.

Developers of closed-source systems often believe that keeping signatures undisclosed is an effective way to protect the system from evasion techniques, over-stimulation attacks, and intellectual property theft. Unfortunately, this sense of security is unjustified. We have developed an approach to evade detection that is based on information obtained from reverse engineering closed-source signatures and detection routines. The reverse engineering process involves the dynamic analysis of the sensor binary when it is stimulated with legitimate and malicious input. The analysis results are then used to guide the selection of appropriate evasion techniques from a set of alternatives.

Being able to generate instances of attacks that evade detection by Snort is not very surprising, since the Snort signatures are readily available. However, in this paper we demonstrate how by using information gathered during the reverse engineering process we were able to generate instances of attacks that evade detection by ISS' RealSecure. Although the results are limited to one commercial system, the comparison with Snort shows that a closed-source approach does not necessarily afford better protection against evasion attacks. In addition, this methodology lays the foundation for an approach that can leverage black-box testing of network-based intrusion detection systems as, for example, described in prior work by the authors [14].

The remainder of this paper is structured as follows. Section II presents related work. Section III discusses our evasion technique and the reverse engineering tool that we developed for this purpose. Section IV presents the vulnerability and a corresponding attack that are used in our case study to demonstrate how Snort (in Section V) and ISS' RealSecure (in Section VI) can be successfully evaded. Finally, Section VII draws conclusions and outlines future work.

II. Related Work

Evasion techniques have been studied since the very first introduction of intrusion detection systems. In the field of network-based intrusion detection, several techniques have been proposed (see in particular the work of Ptacek et al. [16]) and implemented [8], [21]. A more recent effort has investigated the potential for algorithmic denial of service of NIDSs [4]. Additionally, work has been done on hardening IDSs against evasion attacks using traffic normalization [7] or by eliminating network-level ambiguities [19].

The introduction and vast deployment of open-source NIDSs, most notably of Snort [18], also spawned the creation of tools that leverage the signatures of a system to drive attacks against the detection process. A class of these tools is represented by over-stimulation tools such as Snot [20], Stick [6], IDSWakeup [2], and Mucus [14], which was developed by the authors. The concept of signature-driven traffic generation was extended with Mucus to perform black-box testing of closed-source network-based intrusion detection systems. One of the lessons learned in developing the cross-testing technique was that, due to intellectual property concerns, NIDS developers are very secretive about their signatures, even when presented with the possibility of getting useful feedback about the effectiveness of their detection capabilities [14]. As a consequence, it is almost impossible to obtain signature sets from vendors. These vendors often claim that by making the detection process closed-source, their particular IDS is made more resilient to evasion and over-stimulation attacks. This claim motivated our work and is evaluated in the remainder of the paper. Our findings show that closed-source signatures and algorithms provide only limited protection against evasion or over-stimulation attacks and may provide a false sense of security. To the best of our knowledge, a technique that uses the results of the reverse engineering of closed-source NIDS signatures to drive evasion attacks has never been proposed before.

III. Evasion and Over-Stimulation

Intuitively, we define the *attack space* for an attack as the set of all event sequences (e.g., network traces, system call traces) that contain successful instances of this attack. The *signature space* of a signature is

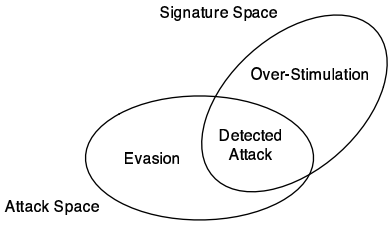

Fig. 1. Evasion and over-stimulation.

defined as the set of all event sequences that match (i.e., fulfill the constraints of) this signature. Evasion and over-stimulation attacks are possible when the attack space and the signature space for a particular attack do not completely overlap (see Figure 1). Event sequences in the attack space that are not in the signature space are called evasion attacks, while sequences that are in the signature space but not in the attack space are called over-stimulation attacks.

Previous work has demonstrated that current IDS implementations are open to a variety of evasion attacks [16], [25]. These evasion attacks involve desynchronizing the view of the IDS from the view of the attacked service with respect to a stream of network events. By doing this, a successful attack against the service may appear benign to the IDS.

The quintessential example of a network-level IDS evasion is the overlapping IP fragment attack [16]. This attack exploits an undefined area of the IP specification with regard to the reassembly of overlapping IP fragments. Since the specification fails to mention whether newer or older overlapping data is to be used, the network stacks for various operating systems behave differently, some preferring newer data, others preferring older data. Historically, NIDS have followed a general policy of preferring either only newer or only older data. Thus, if an attacker knows that a host protected by a NIDS reassembles fragments differently than the NIDS itself, then it is possible to hide the signs of an attack from the NIDS by fragmenting the IP packets carrying the attack in a way that makes the NIDS believe that the traffic is benign.

A common example of a protocol-level IDS evasion is the use of alternate data encodings that are correctly handled by an application but not parsed by an IDS [5]. For instance, the HTTP specification defines a method for encoding characters as part of a URL which are excluded from the set `[0-9a-zA-Z$-_.+!*'(),]` by replacing these characters with a "%" character followed by a two-digit hexadecimal representation of the characters' ISO-Latin code. HTTP servers are expected to correctly decode URL-encoded strings as necessary. However, there have been many instances of IDS implementations that neglected to account for URL-encoded strings. This allowed attackers to evade detection by encoding the attack in a way that prevents the NIDS from successfully matching the malicious URL.

A. Evasion Technique

The task of evading detection by a misuse-based NIDS is possible, as described above, whenever there exists some sequence of events that is both (a) a member of the set of sequences that constitute a successful attack on a target system and (b) falls outside the set of event sequences matched by the signatures in the NIDS. When using a black-box approach (without information about the detection process or precise details about the signatures), finding such a sequence is an iterative process with the following two steps.

1) *Modification Step:* Modify the attack using known evasion techniques [24].

2) *Verification Step:* Test the attack on the target system, verifying that the modified exploit successfully compromises the target. Then, check which alerts were generated by the intrusion detection system. When no alerts were generated, the evasion attempt was successful, otherwise the process is repeated from Step 1.

The major shortcoming of this approach is that it can be very time consuming, especially when both the code modification step and the verification step have to be performed manually. Also, without knowledge

of the inner workings of the detection process and the signatures used by the system, it is not clear which changes to the attack are most promising. Additionally, even when both steps (exploit modification and verification) can be automated, it is still possible that the code modification rules are not powerful enough to express an evasion opportunity that is specific enough for a particular system or signature.

When the source code and signatures for a NIDS are available, however, a better approach is to try to understand the detection mechanism of the system and identify potential problems in the input parsing routines or in the attack signatures. In misuse-based IDSs, signatures are typically encoded as a conjunction of constraints on values contained in one or more input events. In general, an alert is issued by the system when all such constraints are satisfied. Thus, for the purposes of evading an attack, only one such constraint needs to be unsatisfied in order for the events associated with the evasion attempt to go undetected. Explicit knowledge of these constraints establishes the power of the modification rules being applied with respect to the set of constraints imposed by the target NIDS. This often permits the attacker to determine whether the modification ruleset is powerful enough to perform the desired evasion. In addition, knowledge of signature constraints can be used to drive the modification in the direction most likely to result in a successful evasion. With this in mind, Step 2 in the two step process from above is replaced with Step 2', which includes an analysis sub-step:

2') *Verification and Analysis Step:* Test the attack on the target system, verifying that the modified exploit successfully compromises the target. Then, check which alerts were generated by the intrusion detection system. When no alerts were generated, the evasion attempt was successful. *If an alert is generated, determine the root cause(s) of the detection. Return to Step 1, applying the findings of Step 2'.*

Analysis of open source NIDS signatures, such as those in Snort's signature set, reveals that signatures often exhibit weaknesses: they are designed to match only one variant of a particular attack, or they may be written to detect circumstantial or collateral evidence of an attack, as opposed to direct evidence. For example, a signature that looks for indirect evidence may be triggered when a packet is sent to a particular port and the packet has a length greater than some value. A more precise signature may additionally look for shellcode in the packet's payload, which more directly indicates an attack. In the case where a signature is written to match a single instance of a vulnerability being exercised – for example, matching a specific string of bytes in an attack's shellcode – evasion is often possible by inserting an instruction that has no effects, or by remapping the registers used by the shellcode. These observations suggest that signature analysis can be a significant benefit to the evasion process.

When the source code of the NIDS and its signatures are available, the attacker is able to observe the precise sequence of checks made by the system on input events prior to issuing or not issuing an alert. However, this is not possible in general for closed-source systems, since the sequence of checks is not deducible from the textual descriptions of signatures that commonly accompany commercial IDSs. Furthermore, even in cases where signatures are available, imprecise definition of the semantics of the signature language may lead to uncertainty with respect to what checks the implementation actually performs on events in its input. The following section proposes a technique for signature analysis under such circumstances.

B. Reverse Engineering

In order to be able to determine the reason why a particular alert is generated by a closed-source IDS, the binary has to be analyzed. This process, often called *reverse engineering*, is defined as the process of analyzing a system to identify the system's components and their interrelationships and creating representations of the system in another form or at a higher level of abstraction. As such, the reverse engineering process is always closely connected with and dependent on the system that is analyzed. It is possible, however, to provide general guidelines and tools that support this process.

First, it is useful to have a mechanism that can quickly generate variations of input events (e.g., network packets with different payloads), which is needed during the modification step. In our experiments with

network-based intrusion detection systems, we used two publicly available tools: hping [9] and an extended version of our IDS testing tool, Mucus [14].

During the analysis step, it is helpful to have (i) a static disassembly of the binary available, and (ii) a dynamic trace of each instruction that is executed by the intrusion detection sensor when an input event is received.

For our experiments, the dynamic traces were gathered using the *ptrace* system call interface. The ptrace system call provides a means by which a process can observe and control the execution of another process and examine and change its process image and registers. One of the ptrace options allows the tracing process to single-step the traced process. That is, after each instruction is executed by the traced process, control is transferred back to the tracing process, which is then able to inspect register and memory values. This allows one to record and analyze each executed instruction along with its operands.

The ptrace interface is usually used to implement debuggers (such as `gdb`) or system call tracing. For our experiments, we implemented an instruction tracing tool, called *itrace*, to gather dynamic traces. Itrace uses the single-step functionality of ptrace to execute single instructions of the process that is under analysis. After each step, the instruction that has been executed is parsed and disassembled. Additionally, an analysis of the instruction's operands is performed. For each register operand, the current value of the corresponding register is shown. For each memory access, the corresponding memory addresses are calculated and the values at these addresses are extracted from the running process image. Because itrace 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 variable length CISC operations with addressing modes that can take up to three register and immediate value components.

Itrace also has the capability to identify function call and return instructions. This information is used to build a control flow graph of the application that graphically shows the way functions call each other at run-time. Having dynamic data from an actual program execution available is particularly beneficial when the code contains indirect function calls or indirect jumps. A control transfer instruction, such as a call or a jump, is called *indirect* if the target of this instruction is not determined by a constant address or offset. Instead, the target is obtained from a register or memory address at run-time. Therefore, it is often not possible to statically find the targets of indirect control transfer instructions and only an incomplete call graph can be built. When using the data from a dynamic trace, however, it is at least possible to include a subset of the valid call targets (i.e., all targets that are called during the program's execution) in the control flow graph.

IV. Case Study

The case study in this paper demonstrates our evasion technique on both an open-source and a closed-source NIDS, and makes use of a remotely exploitable vulnerability in the Apache web server as a pedagogical example. The vulnerability appears in Apache 1.3 through 1.3.24 and Apache 2.0 through 2.0.36. It is caused by the way in which chunk-encoded HTTP requests are handled.

HTTP chunk encoding is specified in the HTTP/1.1 protocol as a specific form of transfer 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`" has to be specified in the transfer encoding header field; then a sequence of chunks can be transmitted. 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. Both the chunk length field and the chunk data block are terminated by a carriage-return ("\r") character followed by a line-feed character ("\n"). A simple example of a chunk-encoded request is shown below.

```
Transfer-Encoding: chunked

6\r\n          \  first chunk
AAAAAA\r\n     /
4\r\n          \  second chunk
BBBB\r\n       /
0                   final chunk
```

Apache is vulnerable to an integer overflow when the size of a chunk exceeds 0x7fffffff (which causes the most significant bit to become 1). Vulnerable versions of Apache treat the chunk size as a signed 32-bit integer and fail to include the proper size checks. Thus, an attacker can craft the request so that the overflow is triggered, and arbitrary code (which can be included in other header fields of the HTTP request) can be executed.

This particular vulnerability was selected for a number of reasons. First, Apache is the most widely deployed web server and thus a prominent target for attackers. A vulnerability in a program with Apache's installation base raises a lot of interest both in the black-hat community and among security system vendors. This has the important result that several exploits that can be readily used for our case study are currently circulating on the Internet. In addition, these exploits cannot be ignored by intrusion detection system vendors and maintainers, and it is in their interest to provide "good" signatures for them.

Another reason for choosing the chunk encoding exploit is that it is a complex attack that exploits an input validation error in Apache. The attacker has to first connect to the web server using a TCP three-way handshake and then supply specially crafted input, including shellcode, that triggers the vulnerability. This gives the IDS multiple chances to detect the attack. However, to be able to write a good signature that correctly models the vulnerability, the system is required to correctly reassemble the TCP stream, parse the HTTP protocol, and detect the oversized chunk length.

The following two sections describe the steps that were taken to evade detection by an open-source IDS (Snort) and a closed-source IDS (ISS' RealSecure) when exploiting the Apache vulnerability described above.

V. Evading the Snort NIDS

The Snort 2.1.1 ruleset contains two signatures for detecting the Apache chunk overflow attack. The first signature looks for traffic directed to a web server that contains a binary sequence known to appear in the shellcode of a known chunked encoding exploit:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
  (msg:"WEB-MISC apache chunked encoding memory corruption exploit
  attempt"; flow:established,to_server;
  content:"|C0 50 52 89 E1 50 51 52 50 B8 3B 00 00 00 CD 80|";
  reference:bugtraq,5033; reference:cve,CAN-2002-0392;
  classtype:web-application-activity; sid:1808; rev:3;)
```

This signature is readily recognizable as being weak in the sense that it matches a particular *attack*, as opposed to detecting a general class of activity related to the *vulnerability*.

The second signature matches packets destined for a web server whose payloads contain padding characters that are known to occur in packets generated by another known exploit of this vulnerability:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
  (msg:"WEB-MISC Apache Chunked-Encoding worm attempt";
  flow:to_server,established;
  content:"CCCCCCC\: AAAAAAAAAAAAAAAAAAA"; nocase;
  classtype:web-application-attack; reference:bugtraq,4474;
  reference:cve,CAN-2002-0079;reference:bugtraq,5033;
  reference:cve,CAN-2002-0392; sid:1809; rev:2;)
```

Again, it can be seen that the content being matched is not direct evidence of a vulnerability being exploited, but rather a string that is associated with a known (single) instance of an attack that exploits the vulnerability. As we will see, this specificity can be exploited to evade detection.

As mentioned above, two exploits for the Apache chunked encoding vulnerability were readily available from sources on the Internet. The first exploit considered for the reverse engineering effort uses shellcode that contains precisely the binary sequence specified in the first signature, so it was discarded in favor of the second exploit.

Having chosen the exploit, the evasion effort focused on refinement of the exploit with respect to the second signature. Recall that this signature triggers on payloads containing a fixed string. In order to evade detection, the string produced by the attack was modified so that one space instead of two spaces occurred after the colon (i.e., "`CCCCCCC\:␣AAA...`" was replaced with "`CCCCCCC\:␣␣AAA...`"). The attack still succeeds since this modification has no effect on how Apache interprets the string. However, the signature does not fire on the modified string.

Surprisingly, this testing revealed an additional alert from one of Snort's preprocessors (`http_inspect_server`):

```
[**] [119:16:1] (http_inspect) OVERSIZE CHUNK ENCODING [**]
```

This unexpected alert had to be evaded as well in order for the evasion attack to be considered successful.

Examining the exploit code in conjunction with the `http_inspect_server` preprocessor code revealed the cause of the alert. Since the chunked encoding exploit relies on overflowing a signed integer representing the chunk length, the length appears as a very large unsigned integer (`0xffffff6e`). The `http_inspect _server` preprocessor can be parameterized with a maximum chunk encoding that defaults to 500000, far below the value for the chunk length required for the exploit to be successful.

With this knowledge, an attack on Snort's parser was devised. Upon inspecting the preprocessor's routine for parsing the chunk length, we noted that the code assumes the integer encoding is immediately followed by a carriage-return/line-feed sequence ("`\r\n`"). Violating this assumption results in a variable being set to `0`, which prevents the accumulated value of the chunk length from being compared against the limit. The attack was therefore modified to send a length sequence containing a tab character: "`ffffff6e\t\r\n`", and the padding string generated by the exploit was reduced by a single character to account for the modification. The modified attack was tested successfully against Apache, and no Snort alerts were produced.

VI. Evading the RealSecure NIDS

RealSecure, which is developed by Internet Security Systems (ISS), was chosen as the closed-source system for our evaluation because, as of the time of writing, it is commonly considered to be the most widely deployed commercial intrusion detection system. In addition, ISS established X-Force, a respected security team that performs in-depth security research, including penetration testing on common server applications and systems. Their real-world expertise is claimed to be a driving influence on the security of ISS' products and services.

First, we downloaded the evaluation version of the RealSecure network sensor from ISS' web site. The sensor is Version 7.0 (the latest available at the time of our experiments), and it is shipped as several binary

packages for RedHat Linux 7.3. In addition to the sensor, a custom Linux 2.4.18 kernel is provided, which replaces the standard kernel of the RedHat 7.3 installation. We also downloaded SiteProtector for Windows 2000, a centralized management console that remotely controls RealSecure sensors and provides a central point to collect and display alerts. The Windows 2000 host running SiteProtector and the RedHat 7.3 host running the RealSecure sensor were both deployed on the same private network that was previously used to analyze Snort.

Before the reverse engineering effort was undertaken, we hoped to extract the bulk of RealSecure's signatures. We assumed that there was a signature file, which is loaded when the sensor is started, and that it would only be necessary to determine the signature format. At the very least, we expected that there would be data structures in the memory image of the running sensor process that could be mapped to simple checks (e.g., such as checking the destination port in a TCP packet against the value of the HTTP port). This would then allow us to reconstruct most signatures.

Unfortunately, these assumptions turned out to be wrong. RealSecure uses a shared library (called `iss-pam1.so`), which is dynamically loaded when the sensor process is launched. This library encodes the signatures and their corresponding checks directly as executable code. To be more precise, there is no location in the executable file or in memory that contains a parametric description of which checks have to be performed by a general purpose detection routine. Instead, the library code contains explicit machine code instructions for each check that needs to be performed on behalf of every signature. These instructions mostly operate with immediate values (i.e., hard-coded values that are part of the instruction) that are compared against values in the input data. This makes the automatic extraction of signatures comparable to the difficulty of the program understanding problem. However, it is feasible to analyze the program trace for a single attack and determine the input processing and the checks that are performed before a particular alarm is raised.

Fortunately, there was no obvious attempt to obfuscate the library code or to harden the binary against reverse engineering. Even though the signature library is stripped (i.e., symbol information used for debugging and relocation information used for linking are removed) it can be easily disassembled. Additionally, when printing strings of three or more consecutive printable characters in the library file (using the `strings` utility) many content strings that were likely to be part of attack signatures can be seen. Also, ISS ships RealSecure with a list that contains each signature's name, identification number, and a brief description of its purpose.

The idea for the analysis was to use *itrace* to record a dynamic trace of each instruction that was executed by the sensor beginning from the point in time when a packet, containing the malicious payload, was received to the instant in which the attack was actually detected by the sensor. To constrain the program trace to contain only relevant code that is executed when input data is matched against signatures, we attached *itrace* to the sensor process after all startup routines had finished and the sensor had entered a polling loop to wait for a network packet to arrive.

We decided to start as simply as possible, by sending a single, zero-length UDP packet with a destination port of 161 to the RealSecure sensor. According to the signature list, such packets are considered to be SNMP probes that raise a single alert. No other packets were transmitted over the network while the trace was in progress. However, even for one UDP packet, the resulting trace contained several million lines. At this point, we observed that whenever a signature is triggered, an immediate value that is equivalent to this signature's identifier (according to RealSecure's signature list) is pushed on the stack. This observation allowed us to locate the point in the trace where a signature is detected and allowed us to focus on the code region that is executed immediately before, in the assumption that the relevant checks happen there.

This assumption proved to be correct, and for the UDP packet sent to port 161 a series of instructions that compare the packet destination to immediate values of prominent destination ports (e.g., HTTP, SMTP, or TELNET ports) was executed before the signature identifier for the SNMP attack was pushed. By following the path in the library code for other destination ports, signatures for attacks against the

corresponding services were located.

Before starting with the more complex Apache chunk encoding attack, which involves multiple packets, a simpler UDP-based NTP (network time protocol) daemon overflow attack was analyzed. A signature for this attack is included in the RealSecure ruleset and a readily available exploit from the Internet was correctly identified. Upon analysis of the library code and the traces, it turned out that ISS' signature is triggered by all UDP packets that are sent to the NTP port and that exceed a certain length. That is, no analysis of the protocol or the payload takes place. This finding was easily verified by crafting a zero-filled packet of the appropriate length and observing that RealSecure generated an alert in response. Thus, the NTP daemon buffer overflow signature is a typical example of an inaccurate signature that is vulnerable to over-stimulation. Because the packet payload and protocol details are ignored, it is easy for an attacker to trigger this alert at will and over-stimulate the system.

After our experience with the two previous test cases, the Apache exploit was analyzed. As an initial step, we executed the unmodified exploit against the victim host running Apache. RealSecure reported the following three alerts:

1) HTTP_Apache_Chunked_BO: This signature checks for an HTTP packet containing the string "Transfer-Encoding: chunked" and evaluates each chunk to see if its specified size is greater than a certain Max_Chunk_Size.
2) HTTP_Field_With_Binary: This signature detects HTTP requests for fields with binary (non-ASCII) data.
3) HTTP_Fields_With_Binary: This signature detects HTTP requests for three or more fields of any size that contain binary (non-ASCII) data.

The alerts show that the system correctly detected the attack and, in addition, reported two additional warnings that refer to the shellcode that is included in more than twenty additional HTTP header fields. When the modified exploit that evaded detection by Snort was launched, RealSecure still reported all three alerts. This provides at least some evidence that ISS does not simply adopt the publicly available Snort rule for the Apache attack.

The HTTP_Field_With_Binary alert is triggered because the shellcode and return addresses of the exploit are encapsulated in header lines and contain many non-ASCII characters. When analyzing the trace for this alert, we noticed an additional check that is performed on the length of the header line. It turned out that the alert is only raised when binary data is present **and** the number of characters in the header line exceeds a threshold of 100. While this additional test was probably introduced to reduce false positives, it provides an easy way to evade detection by keeping the length of the exploit code below this threshold. An alternative avenue of evasion would be to encode instructions or replace them with other semantically equivalent ones such that only ASCII characters are used [1]. In this experiment, the simpler approach was chosen and the shellcode was shortened appropriately. Given the knowledge of the detection process and the additional length check, it was straightforward to apply a successful evasion technique for this alert.

The HTTP_Fields_With_Binary alert is raised because the exploit uses more than two header fields to store the exploit data. For this alert, the signature description, which states that three or more fields have to contain binary data to trigger the alarm, is sufficient to allow successful evasion by reducing the number of header lines with binary data to two.

The last alert, HTTP_Apache_Chunked_BO, is directly related to the attack. When we reverse engineered the code that triggered this signature, we found that the RealSecure sensor parsed the HTTP request and extracted the chunk size values from the request. This chunk size is then checked to ensure that it is small enough to not cause an overflow. The signature is higher quality than the comparable Snort rules that check for particular string values that appear in payloads of wide-spread exploits but that are unrelated to the actual vulnerability[1]. This signature also highlights the importance of reverse engineering the intrusion

[1]However, recall that Snort's http_inspect_server preprocessor extracts the chunk size values from the request and compares them against a pre-set limit.

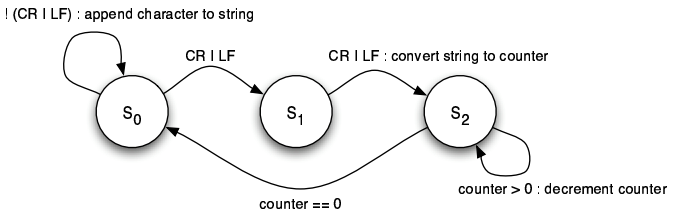

Fig. 2. Simple parsing state machine reverse engineered from RealSecure.

detection sensor. A black-box approach could have invested significant effort in modifying and disguising the shellcode of the attack, thus focusing on an area that is not checked by the signature.

Given the quality of the signature, which relies on protocol knowledge to accurately model the vulnerability, our focus shifted to techniques to desynchronize the view of Apache and RealSecure with respect to the HTTP request. This requires an understanding of the parsing routines that are used by RealSecure to extract the chunk sizes from the HTTP request. The analysis of the traces and the corresponding library code revealed that a simple state machine (shown in Figure 2) is used.

This state machine correctly implements the HTTP specification for chunk-encoded requests. In state S_0, the characters of the chunk size field are read. Because the size field is expected to be terminated by both a carriage-return (CR) and a line-feed (LF) character, there are two transitions from state S_0 to state S_1 and from state S_1 to state S_2. These transitions are actually taken for both CR and LF characters. When state S_2 is reached, the size field string is converted into the corresponding number and a counter is initialized. This counter is decremented for every character that is subsequently read in state S_2, thus discarding the following chunk data. When the counter reaches zero, the transition back to state S_0 is taken and the next chunk is processed.

Apache, however, is more lenient in accepting input that does not completely adhere to the standards. This is necessary for a web server program that has to deal with peculiarities and implementation differences of a large variety of web clients. In particular, Apache accepts chunk size fields that are terminated with a single line-feed character, omitting the required carriage-return.

This difference in input handling was exploited to launch a successful evasion attack. Consider the following request:

```
5\n              // first chunk length
XXXXX\n          // chunk data (5 characters)
ffffff6e\r\n     // second chunk length (overflow)
```

By omitting the carriage-return character after the chunk size field, RealSecure's parsing routine was forced to remain in state S_1, waiting for a second end-of-line character, while reading characters from the chunk data. Because RealSecure's implementors did not expect to receive input characters other than carriage-return or line-feed in state S_1, the input is silently dropped. When the second line-feed character is encountered at the end of the chunk data, only the string "5" is passed to the number conversion routine and the counter is set to five. Thus, the next five characters ("fffff") of the second chunk length in the third line are interpreted as chunk data and discarded in state S_2. Then, the state machine returns to the initial state S_0. The remaining characters of the second length field ("f6e") are interpreted as the length of the following chunk, passing the check for overflowing values. Apache, however, interprets the request as if all lines were correctly terminated with both carriage-return and line-feed characters. Therefore, an attacker can successfully exploit the vulnerability in Apache and evade detection by RealSecure.

For the previous two case studies as well as the Apache exploit, the use of itrace was essential in quickly identifying the sections of code responsible for RealSecure's attack detection. First, the automated nature of the execution trace generation allowed analysis of the code to be performed in a time-efficient manner with various stimulations at a great level of detail. To the best of our knowledge, no other tool available

today has this capability. Additionally, by correlating each run with the changes in the input event stream, itrace enabled us to quickly pinpoint the location and nature of the tests that RealSecure performed on the input data. This ability made it relatively simple to enumerate most or all of the checks performed on input data for a given attack, iteratively revealing the nature of a given signature in a short period of time. Having successfully reverse engineered RealSecure's detection algorithms, it was then trivial to deduce methods of evading the system.

VII. Conclusions and Future Work

We developed a technique to reverse engineer closed-source signatures using correlated analysis between an IDS stimulator and an execution tracer. The results of the reverse engineering process are used as a basis to select and configure evasion mechanisms that are suitable for a specific attack or a specific system. The experiments show that by leveraging the results of the analysis it is possible to build modified versions of attacks that will evade detection by a closed-source, commercial tool.

Our reverse engineering approach supports a better understanding of how the match for a signature is performed and what the reasons for a missed detection are. Therefore, by using this approach it is possible to perform more focused and precise testing of closed-source systems.

Future work will focus on creating a more general methodology that will allow one to derive over-stimulation or denial of service attacks in addition to evasion attacks. Preliminary analysis indicates that by using dynamic analysis, it is possible to correlate the amount of resources used by a network sensor to the traffic generated by an IDS stimulator. Using this information, it may be possible to devise attacks that would force a sensor to use a large amount of resources.

Section VI identified the potential problem that binary and algorithmic obfuscation of binary signatures poses for our approach. Previous work has explored the related problems of obfuscation ([13]) and deobfuscation ([3], [11], [12]). Future work will examine the extent to which signature obfuscation arises in commercial sensors and explore effective means for extracting semantic meaning to assist reverse engineering in the face of obfuscation.

Finally, we plan to automate the comparison of trace sets generated by itrace in order to automatically derive the tests performed by a closed-source sensor. By composing the IDS stimulation device with itrace in a framework that allows feedback from itrace to drive the stimulation process, we believe that the process of reverse engineering closed-source signatures can be performed almost completely without user intervention.

VIII. Acknowledgments

This research was supported by the Army Research Laboratory and the Army Research Office, under agreement DAAD19-01-1-0484. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

References

[1] C. Anley. Creating Arbitrary Shellcode In Unicode Expanded Strings – The "Venetian" exploit. `http://www.ngssoftware.com`, January 2002.

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

[3] C. Cifuentes and K. Gough. Decompilation of Binary Programs. *Software Practice & Experience*, 25(7):811–829, July 1995.

[4] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[5] H. Debar and B. Morin. Evaluation of the diagnostic capabilities of commercial intrusion detection systems. In *Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID)*, October 2002.

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

[7] M. Handley, C. Kreibich, and V. Paxson. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Usenix Security Symposium*, 2001.

[8] Horizon. Defeating Sniffers and Intrusion Detection Systems. *Phrack Magazine*, 8(54), December 1998.

[9] hping. Network Packet Analyzer and Reassembler. `http://www.hping.org/`, 2004.

[10] ISS. RealSecure. http://www.iss.net/, 2004.
[11] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.
[12] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, December 2004.
[13] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 290–299, Washington, DC, October 2003.
[14] 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 Annual Computer Security Applications Conference (ACSAC '03)*, pages 374–383, Las Vegas, Nevada, December 2003.
[15] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.
[16] T.H. Ptacek and T.N. Newsham. Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, January 1998.
[17] M.J. Ranum, K. Landfield, M. Stolarchuck, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a Generalized Tool for Network Monitoring. In *Proceedings of the 11th Large Systems Administration Conference (LISA '97)*. USENIX, October 1997.
[18] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th Large Systems Administration Conference (LISA '99)*, November 1999.
[19] U. Shankar and V. Paxson. Active Mapping: Resisting NIDS Evasion Without Altering Traffic. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 2003.
[20] Sniph. Snot. http://www.sec33.com/sniph, 2001.
[21] D. Song. nidsbench: A network intrusion detection system test suite. http://packetstorm.widexs.nl/UNIX/IDS/nidsbench/, 1999.
[22] Symantec. ManHunt. http://enterprisesecurity.symantec.com/, March 2004.
[23] G. Vigna and R.A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.
[24] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, October 2004.
[25] D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the* 9^{th} *ACM Conference on Computer and Communications Security*, pages 255–264, Washington DC, USA, November 2002.