RELIABLY DETERMINING THE OUTCOME OF COMPUTER NETWORK ATTACKS

David J. Chaboya, Richard A. Raines, Rusty O. Baldwin, and Barry E. Mullins
Air Force Institute of Technology
Electrical and Computer Engineering (ENG)
2950 Hobson Way, Bldg 642
Wright-Patterson AFB, OH 45433-7765

ABSTRACT

Organizations frequently rely on the use of Network Intrusion Detection Systems (NIDSs) to identify and prevent intrusions into their computer networks. While NIDSs have proven reasonably successful at detecting attacks, they have fallen short in determining if attacks succeed or fail. This determination is often left to the security analyst or system administrator. Large-scale networks pose a particular challenge for IDS analysts. The process of manually checking systems to determine if an attack is successful becomes burdensome as the size and geographic location of the network increases. Many analysts use network data alone, in particular the server response, to determine the outcome of the attack. Intuitively, the server response is the packet or packets the target computer returns after an attack. However, in the case of buffer overflows, the attacker has the ability to forge or modify this response.

This paper examines two key aspects of network defense: the ability to circumvent detection devices and how network analysts respond to evasion techniques. We examine how social engineering can be used to influence an analyst's decisions and we recommend ways to counter this threat. The intended audience will be responsible for either developing IDS signatures, or analyzing network IDS results. The technical detail is moderate, but does assume some exposure to network traffic analysis, intrusion detection, and exploits in general.

INTRODUCTION

In the early years of phone networks, attackers often used social engineering techniques to gain unauthorized access to computer systems. For example, an intruder could pretend to be a telephone repairman or a company executive out on the job that forgot the password and needed it reset. With just a small amount of preparatory work the intruder would convince the help desk technician that the request was legitimate and needed to be granted immediately. Hacker groups in the late 80's and early 90's such as "The Masters of Deception" found that it was often easier to just ask for information instead of finding and exploiting technical vulnerabilities. Kevin Mitnick, arguably the most famous social engineer, conned employees of some of the largest corporations by exploiting the human tendency to trust [1]. Social engineering involves using human trusts to achieve a desired goal. This goal might be to close an important business deal, or to penetrate a network of criminals as an undercover agent. The goal for the computer attacker is unauthorized access to the target network or computer system. While increased training has reduced the threat of social engineering, it continues to be a problem. For example, in a recent Inspector General audit of the Internal Revenue Service (IRS) 35 of 100 employees were convinced to provide usernames and reset their passwords for someone posing as a help desk technician [2].

There are two main types of social engineering: computer-based and human-based. The previous example highlighted a human-based approach. In a computer-based attack the intruder does not directly interact with another human. Instead, an email might be sent containing a malicious file, or a fake website could be used to harvest usernames and passwords. So what does social engineering have to do with Network Intrusion Detection Systems (NIDSs) and evasion? To answer this question the method in which intrusions are detected must first be examined. We highlight in this paper that in most cases a NIDS consist of both a computer and human component. While the computer component of the NIDS may detect an attack, the human (network security analyst) must decide if the attack is a success or failure.

A review of both academic and commercial literature shows significant research in evading the computer component, but little with respect to the human analyst. We address this gap in research by examining network analyst training and then creating evasion techniques that use computer-based social engineering. The goal is to deceive the analyst into believing that a successful attack has instead failed. The hypothesis is that even if an attack is detected, the actual intrusion will go unnoticed if the analyst can be convinced to not follow-up. We focus the exploits tested and the evasion techniques developed on the Windows operating system because of its predominance throughout industry.

However, the goal of this paper is to not show new avenues of attack. Instead we point out that if network traffic is to be analyzed it must be done correctly. This analysis can be fruitful since our experiments suggest that even in the case of buffer overflows the server response often indicates if the attack succeeds or fails. We then propose several methods for determining if the response can be trusted. These defensive measures allow the NIDS or analyst to immediately determine the attack outcome and prevent "NIDS analyst evasion" from being successful.

The first section below provides background information on the topics covered in this paper. Next, we describe related work in the areas of alert verification and IDS evasion. The next sections address how attack outcomes are currently determined and include the results of some of our recent server response experiments. In the fifth section we introduce various NIDS evasion techniques that focus on the analyst. Next, several methods to determine server response trusts are proposed. Finally, we conclude and present ideas for future work.

BACKGROUND

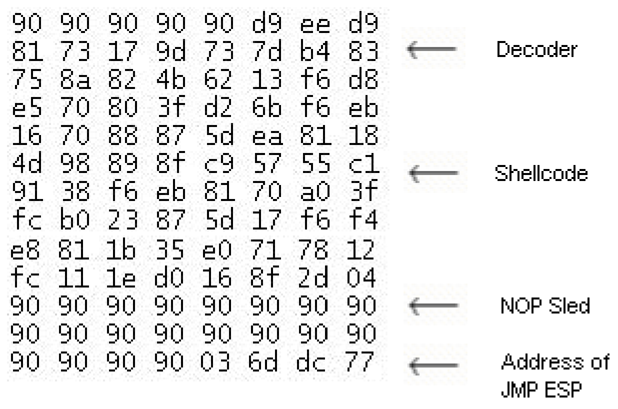
We provide a brief overview of some important terms in computer security in this section. Intrusion detection (ID) is the art and science of finding compromises or the attempt to compromise the integrity of a network or computer system. The term has been broadened to include the detection of other forms of attacks besides intrusions such as scanning, enumeration, and denial-of-service. NIDSs and Host-Based Intrusion Detection Systems (HIDSs) are the two primary technical implementations that exist to detect attacks. A NIDS monitors an entire network for malicious activity, while a HIDS monitors the host it resides on. In addition, Intrusion Prevention Systems (IPSs) not only detect attacks, but also attempt to stop them before any damage is done.

We focus on the NIDS, as it is most vulnerable to the attacks mentioned in this paper and the one that most analysts are responsible for monitoring. The two well-recognized areas of detection are misuse and anomaly-based. Misuse or signature-based detection focuses on known attacks or known characteristics of attacks by matching on a pre-defined byte sequence. Snort [3] is one example of a well-known IDS that makes extensive use of signatures. Anomaly-based detection seeks to establish what is normal and then search for traffic that differs from the baseline. There are many models that can be applied to this form of detection [4].

Alert verification is the process of determining the outcome of an attack [5]. This includes both active measures used after an attack (e.g., running vulnerability scans) and passive measures used before an attack (e.g., collecting network configurations). The server response is defined as the packet or packets the target computer returns after an attack. For instance, a malformed request for a web resource might return a "HTTP/1.1 400 Bad Request" indicating that the attack failed.

Even with the increased focus on good programming practices, buffer overflows continue to be one of the most widely exploited vulnerabilities. In fact, the SANS 2004 Top 10 most exploited Windows vulnerabilities, have buffer overflows in positions one, two, four, and eight [6]. An exploit is the code that an intruder uses to take advantage of a known vulnerability. Many of the early buffer overflow exploits were customized for the UNIX operating system and other variants. A common technique for an attacker was to code the payload to execute a shell upon successful exploitation of the victim. Shellcode now refers to a wide range of payloads that setup backdoor ports, establish reverse-shells to the attacker, modify key operating system files, or add new users. The art of constructing more compact, complex, and functional shellcode has advanced significantly [7].

Buffer overflows typically consist of four main sections: the decoder, NOP sled, shellcode, and return addresses or jump to Instruction Pointer (IP). An abbreviated sample of these sections is shown in Figure 1. In most cases, the decoder is not visibly distinguishable from the rest of the shellcode. In addition, Figure 1 illustrates the overwriting of the saved instruction pointer with the address of a jump ESP instruction. This is just one of the many Windows overflow techniques that allow an attacker to cause control flow to pass to the shellcode. The NOP sled consists of a series of "No-operation" commands (i.e. 90 is the hexadecimal value for the default NOP in the Intel x86 architecture) that pass execution to the next command. This allows the attacker flexibility in guessing the location of the shellcode in memory.



```
90 90 90 90 90 d9 ee d9
81 73 17 9d 73 7d b4 83
75 8a 82 4b 62 13 f6 d8
e5 70 80 3f d2 6b f6 eb
16 70 88 87 5d ea 81 18
4d 98 89 8f c9 57 55 c1
91 38 f6 eb 81 70 a0 3f
fc b0 23 87 5d 17 f6 f4
e8 81 1b 35 e0 71 78 12
fc 11 1e d0 16 8f 2d 04
90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90
90 90 90 90 03 6d dc 77
```

Figure 1. Buffer Overflow Sections

Polymorphic shellcode seeks to evade basic misuse detection by borrowing techniques from computer virus developers [8]. Polymorphism is added through unobtrusive manipulations on spare CPU registers to create unique yet functionally equivalent shellcode. Advances in polymorphic shellcode generation have made even anomaly-based detection methods such as data-mining difficult in locating buffer overflow attacks [9]. In addition, it may be difficult to tell where one section of the shellcode ends and the next begins. Many developers have instead focused on detecting the exploit vector and not the shellcode for this reason. The exploit vector is the method in which the vulnerability will be triggered. For instance, the Printer ISAPI buffer overflow required a URL that included ".printer" and a long Host header field. Of course, not all vulnerabilities have a single exploit vector.

RELATED WORK

In 1998, Ptacek and Newsham introduced the world to the field of IDS evasion [10]. During the same time, Paxson introduced the Bro IDS and several related evasion techniques [11]. Ever since, there has been significant academic and commercial research in IDS evasion. Recent work in mutating network exploits has shown that NIDSs such as Snort and ISS Realsecure still have problems with evasion [12]. We feel that there is little room to advance the state of the art in the areas previously addressed. However, with the exception of alert flooding techniques the analyst's role in detection is untested [13]. Therefore we focus our evasion attacks towards the NIDS analyst.

Each of the attack techniques developed in this paper rely on disguising successful attacks as failures. Previous work in this area focused on HIDSs. For instance, "Mimicry attacks" target HIDS by modifying the exploit characteristics to mimic that of a legitimate application [14,15]. The concept of mimicking a legitimate application can be expanded to describe the general case of hiding attacks by modifying the exploit. This can involve hiding as normal activity, in a spot undetectable to the HIDS, as a less serious attack, and as an entirely new attack [16].

A good discussion of alert verification techniques is provided in [5]. However, this paper omits the category of network based verification (e.g., server response detection). The usefulness of the server response has been discussed in previous works, although the focus has not been on buffer overflows. For instance, Sommer and Paxson discuss the benefit of "request/reply" signatures in improving the quality of alerts [17]. A recent paper parallels our work in describing how response signatures can be used to determine attack outcomes [18]. However, the authors do not address the validity of the response besides saying that it is theoretical possible for a clever intruder to manipulate it. Other methods to determine the success and failure of attacks are based on understanding the layout of the network [17]. The closely related method, commonly used by IDS analysts, of determining the target operating system and matching it to the exploit also gives an indication if the attack will succeed or fail [19].

DETERMINING ATTACK OUTCOME

While NIDSs have proven reasonably successful at detecting attacks, they have fallen short in determining if attacks succeed or fail. Often analysts and system administrators must review network traffic, analyze system and firewall logs, or run vulnerability scans. With the exception of a few active techniques [5], almost all verification methods require human involvement. The process of determining the attack outcome can either be immediate or delayed. The immediate methods include: attacker makes it obvious, server response, network layout or system knowledge, and active vulnerability scanning. The delayed methods include: manual checks of logs or patches, backdoor traffic, and anomaly detection.

The first immediate technique is by far the most common method that analysts use to determine if the attack succeeded or failed. The assumption is that if an attack is successful then the intruder will immediately take action against the target system. The problem with this assumption is that it has no technical foundation. The server response is a proven method to determine success or failure. In the next section we extend this method to common buffer overflows. However, we will also show a possibly overlooked vulnerability in server response analysis. While the use of passive or active network mapping is beneficial it often only provides relevance to an alert. For example, an Apache web attack against an Internet Information Services (IIS) web server will obviously fail. However, if an attacker uses an IIS exploit against an IIS computer then more information is needed. Active verification using vulnerability scans is a way to obtain this information. For example, the target system can be scanned after attack to determine if the vulnerability is present. A weakness is that there is potentially a small window where an attacker could modify the target system to appear patched.

There are also many methods to determine that an intrusion occurred several hours, days, or even weeks after an attack. Analysts or system administrators often manually check logs or patches if they are suspicious that an attack may have succeeded. Unfortunately, this approach is vulnerable to system tampering and can be very time consuming particularly in large networks. Backdoor detection methods can be very effective in catching comprises even though the original attack goes unseen. Finally, anomaly-based detection methods are particularly effective in recognizing unauthorized network traffic or suspicious user activities.

SERVER RESPONSE ANALYSIS

For large organizations the manual verification of attack outcomes is burdensome. In addition, the distributed architecture presents certain problems for active verification methods. For example, if there are multiple paths into a network then the verification system may not have the same visibility as the intruder. In addition, extra costs would be involved to add this capability to regional sites. As a complement to active verification, server response review allows determining the outcome immediately after the attack. In this section we show that server response analysis also applies to buffer overflow vulnerabilities.

Windows 2000 Server and XP Professional were selected for target operating systems and configured with VMware [20], virtual machine software that allows for multiple operating systems on a single physical computer. Windows 2000 Service Pack (SP) 0 through Windows XP SP 1 were tested to insure consistency of results. A testing and development framework, the Metasploit Framework [21], was selected as it is open source and has a wide range of robust Windows buffer overflow exploits. The eight exploits chosen, shown in Table 1, represent some of the most serious and commonly exploited vulnerabilities in the past few years. For example, the WebDav vulnerability was used in 0-day attacks and the LSASS and RPC DCOM vulnerabilities were exploited by the Sasser and MSBlaster worms respectively. In addition, there is a particular focus on IIS attacks as HTTP requests are often the most likely to be allowed through perimeter firewalls. Additional details on the vulnerabilities and exploits listed in Table 1 are available on the Microsoft Security website [22] and the Metasploit website.

Ethereal [23] was used to capture the response since, in a lab environment, it has the same traffic collection capabilities of most NIDS. For a response characteristic to be of significant value it must be consistent (reduces false positives or false negatives) and distinguishable (determines success from failure). It is important to note that we define a "successful" attack as one where attacker code is executed. Unless the shellcode causes obvious network activity (e.g., reverse shell) there is no way to determine (based on the response) if the executed code actually did anything useful. Often shellcode will not be completely universal or reliable resulting in failures. Next, we used a minimum of three repetitions for both patched and unpatched computers in each configuration to obtain the appropriate responses (there was not significant variance observed). In addition, the VMware target host was reset to its default setting (i.e. using the snapshot feature) after each repetition to insure the accuracy of the tests. When possible, we also used publicly available exploits to improve the variability in the tests.

The results of our experiments indicate that unpatched servers do not return a distinguishable response. We can partially attribute this to the control the attacker has when exiting the shellcode. On the contrary, patched servers had very consistent responses as expected. The exploit tested, corresponding Microsoft bulletin, patched and unpatched server responses, and size of the default response is shown in Table 1. The size of the error message can be used in determining the legitimacy of the response as seen later in the paper.

Table 1: Experimental Server Response Results

Exploit	MS Bulletin	Patched Server Response	Unpatched Response	Size (bytes)
Apache Chunked	N/A	HTTP/1.1 400 Bad Request	None	542
IIS_WebDAV	03-07	HTTP/1.1 400 Bad Request	None	235
IIS_Nsiislog	03-19/03-22	HTTP/1.1 400 Bad Request	None or 500 Server Error	111
IIS_Printer	01-23	None	None	N/A
IIS_Fp30Reg	03-51	HTTP/1.1 500 Server Error	None	258/261
LSASS	04-11	WinXP: DCERPC Fault Win2K: LSA-DS Response	None	XP:92 2K:108
RPC DCOM	03-26	RemoteActivation Response	None	92

The results demonstrate that the server response can give clear evidence of the configuration of the target system. The consistency of response (especially for HTTP overflows) makes it ideal for both misuse and anomaly-based detection applications. For example, specific signatures could be created that match the exploit attempt with the associated expected response (i.e., using Snort's flowbits plug-in) [18]. If this response is not seen, then an alert for a successful attack could be generated.

A somewhat surprising result from the above tests was that the exploit vector did not impact the patched server response. However, past experience and the fact that the operating system on the LSASS exploit did influence the response led us to investigate the impact of the exploit vector on additional samples. While not as rigorous as those listed in Table 1, all of the tests (SQL_Hello, Workstation Service (Wksrvc) WINS, NetDDE) reinforced the validity of determining outcome via patched server response. The Wksrvc (MS-03-49) test confirmed that there are limitations to matching on responses. In particular, the two different exploit vectors (*NetrValidateName2* and *NetrAddAlternateComputerName*) resulted in two different patched responses on a Win2k server. In addition, the response on the Win2K and WinXP systems differed. While there is little issue with creating accurate signatures, the limitations of signature detection and the varying OS and exploit vector result in a potential scaling problem. A more sophisticated post-processing engine using protocol analysis may be a solution for those vulnerabilities with no obvious difference between patched and unpatched responses.

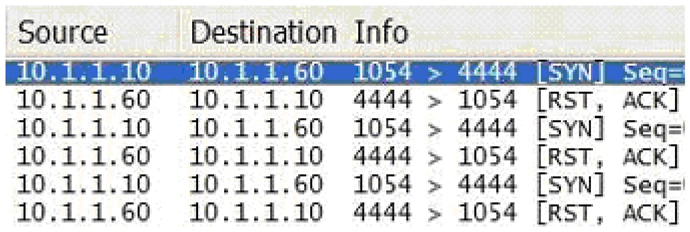
When implementing response checking methods in real world environments one must keep in mind the possibility of custom error messages. For instance, the patched response for the WebDAV exploit might be a 200 OK initially before redirecting to the error page. In addition, it may be another error entirely. However, unless system administrators start getting really creative there should still be clear distinguishable differences that can be programmed in the NIDS. The effort required to determine a reliable and valid response characteristics should not be trivialized. It would be a serious error to write a signature based solely on tests with one public exploit. Instead, developers must be careful to recognize the difference between a response caused by specific exploit characteristics and the responses that may be caused by the various ways to exploit the vulnerability.

NIDS ANALYST EVASION

While it is desirable for the attacker to evade attack detection, the ultimate goal is to conceal the intrusion. Furthermore, the intruder may be unsure if the attack will be detected by the NIDS, despite evasion attempts. It is then logical to assume that the intruder will, in addition, wish to attack the last link in the chain, the analyst. We argue that if the intruder can convince the analyst that a successful attack has failed then the intrusion will go undetected. Then after sufficient time has passed the intruder would either connect back to the victim, presumably using a different Internet Protocol (IP) address, or wait for the shellcode to initiate an outbound connection. In this case, it is almost as if the attack had never been detected in the first place.

Three social engineering techniques are developed in this section. The first method attempts to deceive the analyst into believing an intruder is unable to connect to a backdoor port. The second uses decoy shellcode that makes the attack appear to target another operating system. The third technique requires that the attacker manipulate or forge the server response on unpatched systems. Each of these methods either attacks training, common practices, or trusts that analysts use.

The first method is a simple attempt to deceive the analyst into believing an intruder is unable to connect to his/her backdoor port. IDS analysts are trained to look for signs an exploit failed such as the inability to connect to a backdoor the port. Figure 2 illustrates how this would appear to an analyst.



```
Source     Destination Info
10.1.1.10  10.1.1.60   1054 > 4444 [SYN] Seq=
10.1.1.60  10.1.1.10   4444 > 1054 [RST, ACK]
10.1.1.10  10.1.1.60   1054 > 4444 [SYN] Seq=
10.1.1.60  10.1.1.10   4444 > 1054 [RST, ACK]
10.1.1.10  10.1.1.60   1054 > 4444 [SYN] Seq=
10.1.1.60  10.1.1.10   4444 > 1054 [RST, ACK]
```

Figure 2. Fake Backdoor Evasion

In the example, the intruder attempts to initiate a connection to port 4444 (a known backdoor port associated with several exploits). The target server responds with a RST ACK indicating that the port is not open. An attacker would implement this second technique by using a payload that does not setup a backdoor (i.e., instead adding a new user) and then by sending SYN packets to a backdoor port associated with the exploit.

The second evasion method involves using decoy shellcode that makes the attack appear to target another operating system. IDS analysts are trained to identify the operating system the exploit targets by reviewing the shellcode to look for common indicators [19]. Shellcode that executes the UNIX command shell (e.g. /bin/sh) is frequently seen on the internet. In addition, code that sets up a backdoor on Linux x86 systems is also common in shellcode. The below command sets up a temporary backdoor (on port 1524) in inetd.conf, the internet daemon on Linux:

```
/bin/sh –c echo 'ingreslock stream tcp nowait root /bin/bash bash -i'> /tmp/.inetd.conf; /usr/sbin/inetd /tmp/.inetd.conf
```

If commands similar to the ones above are present in shellcode, it is typically assumed that the shellcode is developed for a UNIX environment. This is used to develop Windows payloads that include non-functional UNIX-based shellcode at the end. To implement this technique in the Metasploit Framework, the real Windows payload is encoded using the *msfencode* command line tool. The tool is modified to insure that the bad characters for the exploit are not present and the final decoy shellcode is created. Finally, the default encoding behavior of the framework is adapted to use an encoder that returns the payload unmodified. This method plays to the "dumb" attacker concept and encourages the analyst to believe that the intruder has no clue about what the target is and how to construct an exploit.

The final evasion technique is the most complex and involves modifying or forging the server response. While it is true that the server response can be trusted in association with simple exploits (i.e., password guessing or directory traversals), buffer overflows and other attacks that give the intruder either administrator or root access are exceptions. In order to forge a realistic packet, a server socket handle or equivalent is required for the connection in question. This handle can either be created through the use of raw sockets or can be located if still available.

The first case is when the attacker creates the forged packet. To do this the IP and TCP headers must be manually constructed and the false response data inserted. While the IP header is easily constructed, creation of a reliable TCP header is not straightforward. The attacker must capture the initial sequence numbers and then calculate the checksum and acknowledgment sequence number. While difficult, this is possible since the attacker is part of the session and usually has prior knowledge of the size of the attack. The final part of sending a raw packet is generating and including the application layer data. For instance, an IIS buffer overflow response might be a HTTP/1.1 400 Bad Request. This method is limited because raw socket forging or *rawsock* requires Windows 2000 or greater due to the required use of IP_HDRINCL. In addition, another limitation is the size to create the packet is at least 350 bytes including the Application Programming Interface (API) calls required (*socket*, *setsockopt*, and *sendto*). However, in the cases where a socket is no longer available or requires too much size to locate, *rawsock* would be the obvious choice.

The second case requires that the attacker locate the socket handle associated with the exploit connection. Two compact methods to achieve this involve locating the peer or source port (*findsock*) or sending and recognizing a hardcoded tag (*findtag*). In the findsock option, the attacker uses the *getpeername* call to "determine the endpoint associated with a given socket" [7]. If the source port matches that of the attacker then a response packet can be forged by placing the required data on the stack and calling *send*. When optimized and hardcoded to a specific service pack this process requires only 40 bytes. In the findtag option, socket descriptors are enumerated by using *ioctlsocket* to determine the amount of data pending in the network's input buffer that can be read from the socket. If data is pending, *recv* is called to compare the hard-coded tag with the one the attacker sent. While findtag does require an additional packet, it also works through Network Address Translation (NAT) devices, unlike findsock. The benefit of using findtag or findsock is in the minimal case the required shellcode size is small. All that is required is the code to find the socket handle (40 bytes), the added size of the error message (shown in Table 1), and a secondary backdoor payload to execute.

A limitation of both the findsock and findtag methods is that in many cases (i.e., Internet Information Server (IIS)) the socket owner is not the process that is exploited. Code that attempts to find or use valid socket descriptors, then, will loop until the thread is externally closed. For these attacks to be successful, the findsock or findtag code must be injected into the correct process and then executed in the context of that process [24]. This requires, at the minimum, the following API calls: *OpenProcess*, *VirtualAllocEx*, *WriteProcessMemory*, and *CreateRemoteThread*. In addition, the correct process must be located through either enumerating open processes (*EnumProcesses*, *EnumProcessModules*, *GetModule*) or by stepping through a snapshot of current processes (*CreateToolhelp32Snapshot*, *Process32First*, *Process32Next*). The minimum size required for process injection is at least 255 bytes.

In certain overflows it is also possible to use the default constructs of the target application to forge the response. This is the ideal case for the attacker since it does not require the extra size of including the error message. For example, in several Internet Server Application Program Interface (ISAPI) overflows it is possible to instead locate the connection ID and use ISAPI functions to forge a message. An attacker using the forging technique would exploit the target system, fake a failed response to trick the analyst, and then execute a delayed backdoor. Figure 3 shows how this attack might appear to an analyst. Packets 1-3 are the TCP handshake. Packet 4 contains the buffer overflow. It then appears that the target issues a bad request error in packet 5 and then resets the connection. In fact, packet 5 is forged by the intruder.

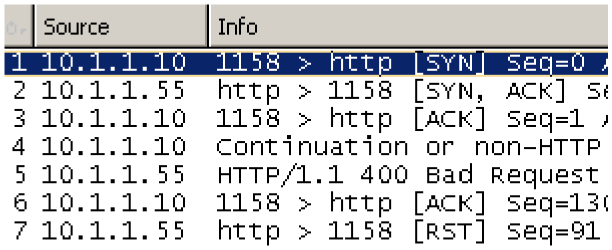

```
  Source      Info
1 10.1.1.10   1158 > http [SYN] Seq=0
2 10.1.1.55   http > 1158 [SYN, ACK] S
3 10.1.1.10   1158 > http [ACK] Seq=1
4 10.1.1.10   Continuation or non-HTTP
5 10.1.1.55   HTTP/1.1 400 Bad Request
6 10.1.1.10   1158 > http [ACK] Seq=13
7 10.1.1.55   http > 1158 [RST] Seq=91
```

Figure 3. Forged Server Response

DETERMINING RESPONSE TRUST

While network traffic analysis saves time and has been shown to confirm attack outcomes it is not as straightforward as once thought. Analysts that blindly trust the response might incorrectly characterize a successful intrusion. Even worse, is the case when the NIDS is programmed to disregard attacks based solely on the response, resulting in complete evasion. Therefore, a method is needed to determine when the server response can be trusted. We provide a brief overview of three such methods in this section. Note that these methods are not intended to be a 100% solution, as an attacker using advanced polymorphic techniques can force the analyst to resort to other means (e.g., active verification) of confirming the outcome. Instead, we focus on the majority of attacks that will not likely be completely polymorphic in nature. Also, all three methods are likely to be processor intensive so post-processing or offline analysis would be ideal.

One solution stems from current methods analysts use to determine the function of the shellcode. The analyst first determines the encoding technique and then decodes the shellcode. The shellcode is then reverse-engineered to determine its functionality. While possible, it is difficult and time consuming. Nevertheless, this method is useful when only the decoder can be located due to a polymorphic NOP sled. One approach is to use emulation and heuristics to speed up the determination of the shellcode function [25]. A similar method is commonly used in the anti-virus community to analyze potentially malicious code without executing it. However, the NIDS has several disadvantages primarily the fact that it does not reside on the attacked host and must be capable of multi-OS and instruction set emulation. In addition, attacker shellcode can be designed specifically to resist automatic analysis through emulation [26].

The next method requires that public shellcode be stored in a database and matched against the attacker's payload. If the payload matches then the response can be trusted (i.e., assuming forging payloads are not made public). While this method does require extra work to maintain the database, once it is created, the upkeep should be minimal. This method is particularly effective against static public exploits (i.e., they are usually compiled and executed with no changes). It is important that the payload comparison algorithm check for the existence of other payloads to prevent an attacker inserting a known shellcode in addition to the real payload.

The first step in using the payload comparison method is to construct a database of known shellcode. To do this we analyzed the public exploits on "Securityfocus.com" and "Securiteam.com" to extract just the payload component. While this captures the majority of the relevant shellcode (for our tests), it is assumed that many more would need to be added in an operational environment. The next examples show the three most likely situations encountered when attempting to extract the relevant payload. First, the easiest case is when the shellcode is almost entirely standalone as in the "oc192-dcom" exploit. This exploit uses a bindshell payload where the only option is the port the shell will bind to. In addition, all bad characters have already been removed from the payload so with the exception of a few of bytes of port information there should be an exact match. The "wbr_c" WebDAV exploit is an example of a payload with nulls and a static XOR key. Slightly more work is involved because each byte of the shellcode must be XORed against the key of 0x95. However, once that work is accomplished it becomes a similar problem as in the first case. The worst case scenario is when the key is calculated during runtime. For instance, the "Webdav-reloaded" exploit determines the key based on a fairly simple *for* loop that checks for nulls, carriage returns, and line feeds.

The next step, and one required in all the methods of determining trust, is to program the NIDS to locate the shellcode. In the common case of public exploits, this can be done by locating the decoder or by following the NOP sled until the shellcode is encountered. Then the end of the shellcode is determined through simple heuristics. Finally, a differential analysis is performed between the intruder's shellcode and those in the database (starting with shellcode associated with the particular exploit). If a match is not found then the system could either default to generating an alert or proceed to the next method of determining trust. If a match is found then the variability between the two payloads is analyzed. Some flexibility is required to account for differences due to attacker modifiable options like ports, IP address, and user name/password.

There is an increasing trend towards small, dynamically encoded payloads which makes payload comparison obsolete. In this case, we recommend using the size of the payload to determine if forging is possible. For example, the latest release of the Metasploit Framework has an average encoded Windows payload size of 246 bytes. Based on payload size, only 1 of the 19 payloads would be large enough to forge responses for the exploits tested. In "payload size analysis" we use knowledge of the attacks developed earlier in the paper to estimate the minimum forging size. Factors considered are the size of the backdoor, error message size (e.g., shown in Table 1), process injection code size, and forging and/or socket location code size. It is important to note that this is the most risky approach of the three as it requires that the IDS developer or analyst be aware of the most optimum methods of forging. For example, we earlier addressed the fact that the intruder does not always have to include the error message in the shellcode. While more research is required to make any definitive statements about size required to forge responses and install a reliable backdoor we feel in most cases at least 350 bytes are required.

The size of the payload is determined by using the NOP sled and any padding to separate the payload from the rest of the exploit. Also, if the NOP sled is polymorphic then the decoder could be located and the size of the payload determined through reverse-engineering. An intruder could split a larger payload either within or between packets in an attempt to defeat the payload size analysis method. However, this attack is defeated by inspecting all packets and reassembling the payload before analysis. Whichever method is used it should be apparent that it takes a more methodical approach to determine the outcome instead of using network characteristics to "guess" at the outcome of the attack.

CONCLUSION

The results of this study highlight the dangers of improper traffic analysis and why the network security analyst can also be vulnerable to evasion attacks. Since the manual evaluation of NIDS alerts is time consuming, error prone, and requires expert knowledge more efficient methods are needed to determine attack outcomes. Hopefully, we have shown that in many cases this process can be automated. However, the presented methods are certainly not a silver bullet approach and often the NIDS will be required to resort to a more active method of alert verification. It is no surprise to us why many people are turned off by the magnitude of effort involved in resolving IDS alerts and why some unfortunately choose to instead just block attacks and forget about what might be getting through. We expect that organizations will wish to block traffic that can be identified, with high confidence, as malicious. However, not all malicious traffic can be blocked and some may wish to not drop attacks but instead only alert on them for whatever reason. It is in those cases that we feel there is significant room for improving how IDS alerts are handled. As a follow-on to this research we expect to implement the response checking methods and further investigate response detection and forging on additional platforms such as Linux.

ACKNOWLEDGMENTS

We would like to thank Sean OHara, Matt Miller and H.D. Moore for their many suggestions and ideas during the development of this research. Finally, we would like to thank the reviewers whose suggestions were incorporated into the paper/presentation and the Air Force Research Laboratory's Anti-Tamper Software Protection Initiative Technology Office for supporting this work.

REFERENCES

[1] K. Mitnick, W. Simon, S. Wozniak, The Art of Deception: Controlling the Human Element of Security, Wiley Publishing Inc., Indiana, Oct 2002, pp. 41-55.

[2] IRS Audit Report 200420035, http://www.lawprofessorblogs.com/taxprof/linkdocs/2005-5545-1.pdf, Mar 2005, pp. 1-13.

[3] Snort Documentation, http://www.snort.org/docs, 2004.

[4] P. Ning, S. Jajodia, "Intrusion Detection Techniques," In H. Bidgoli (Ed.), The Internet Encyclopedia. John Wiley & Sons, Dec 2003, pp. 2-6.

[5] C. Kruegel, W. Robertson, "Alert Verification: Determining the Success of Intrusion Attempts," 1st Workshop on the Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), Jul 2004, pp. 1-14.

[6] SANS Top 20 Internet Security Vulnerabilities, http://www.sans.org/top20, 2004.

[7] M. Miller, "Understanding Windows Shellcode", White Paper http://www.hick.org/code/skape/papers/win32 -shellcode.pdf, Dec 2003.

[8] K2, ADMmutate Documentation, http://www.ktwo .ca/readme.html, 2001.

[9] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk, Polymorphic Shellcode Engine Using Spectrum Analysis, Phrack Issue 0x3d, Aug 2003.

[10] T. Ptacek, T. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," Secure Networks, Inc., Jan. 1998, pp. 11-14.

[11] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," 7th Annual USENIX Security Symposium, Jan 1998, pp. 12-15.

[12] G. Vigna, W. Robertson, D. Balzarotti, "Testing Network-based Intrusion Detection Signatures Using Mutant Exploits," 11th ACM Conference on Computer Security and Communications Security, Oct 2004, pp. 1-10.

[13] D. Mutz, G. Vigna, R. Kemmerer, "An Experience Developing an IDS Stimulator for the Black-Box Testing of Network Intrusion Detection Systems," 19th Annual IEEE Computer Security Applications Conference, Dec 2003, pp. 2-7.

[14] D. Wagner, D. Dean, "Intrusion Detection via Static Analysis," 2001 IEEE Symposium on Security & Privacy, May 2001, pp. 9-11.

[15] D. Wagner, P. Soto, "Mimicry Attacks on Host-Based Intrusion Detection Systems," 9th ACM Conference on Computer Security, Nov 2002, pp.1-4.

[16] K. Tan, J. McHugh, K. Killourhy, "Hiding Intrusions: From the Abnormal to the Normal and Beyond," 5th International Workshop on Information Hiding, Volume 2578 of Lecture Notes in Computer Science (LNCS), Springer Verlag, Oct 2002, pp. 10-16.

[17] R. Sommer, V. Paxon, "Enhancing Byte-Level Network Intrusion Detection Signatures with Context," 10th ACM Conference on Computer and Communications Security, Oct 2003, pp. 5-6.

[18] J. Zhou, A. Carlson, M. Bishop, "Verify Results of Network Intrusion Alerts Using Lightweight Protocol Analysis", 2006 Annual Computer Security Applications Conference (ACSAC), Dec 2006, pp. 1-10.

[19] S. Northcutt, M. Cooper, M. Fearnow, K. Frederick, Intrusion Signatures and Analysis, New Riders Publishing, Indiana, Jan 2001, pp. 269-297.

[20] VMware Documentation, http://www.vmware.com/s- upport/pubs, 2004.

[21] H.D. Moore, Metasploit Framework Documentation,
http://www.metasploit.com, 2004.

[22] Microsoft Security Bulletin Website, http://www.
microsoft.com/security/bulletins/default.mspx, 2004.

[23] Ethereal Documentation, http://www.ethereal.com, 2004.

[24] R. Kuster, "Three Ways to Inject Your Code into Another Process", http://www.codeproject.com/threads/ winspy, Jul 2003.

[25] M. Polychronakis, K. Anagnostakis, E. Markatos, "Network-Level Polymorphic Shellcode Detection Using Emulation", 3rd Conference on the Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), Jul 2006, pp. 11-20.

[26] A. Czarnowski, "Code Emulation in Network Intrusion Detection/Prevention Systems", Virus Bulletin 2005 Article, http://www.virusbtn.com/virusbulletin/archive/2005/08/vb200508-code-emulation.dkb, Aug 2005