Buttercup: On Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities

A. Pasupulati, J. Coit, K. Levitt, S. F. Wu
Department of Computer Science
University of California, Davis
{pasupula, coit, levitt, wu}@cs.ucdavis.edu

S.H. Li, J.C. Kuo, K.P. Fan
ITRI
Hsin-Chu, Taiwan
{shli, jckuo, kpfan}@itri.org.tw

Abstract

Attack polymorphism is a powerful tool for the attackers in the Internet to evade signature-based intrusion detection/prevention systems. In addition, new and faster Internet worms can be coded and launched easily by even high school students anytime against our critical infrastructures, such as DNS or update servers. We believe that polymorphic Internet worms will be developed in the future such that many of our current solutions might have a very small chance to survive. In this paper, we propose a simple solution called "Buttercup" to counter against attacks based on buffer-overflow exploits (such as CodeRed, Nimda, Slammer, and Blaster). We have implemented our idea in SNORT, and included 19 return address ranges of buffer-overflow exploits. With a suite of tests against 55 TCPdump traces, the false positive rate for our best algorithm is as low as 0.01%. This indicates that, potentially, Buttercup can drop 100% worm attack packets on the wire while only 0.01% of the good packets will be sacrificed.

Keywords

Buffer Overflow, Polymorphic Shellcode, Network Intrusion Detection System, Snort.

1. Introduction

Since a signature-based Network Intrusion Detection System (NIDS) identifies an attack instance by exactly matching attack signatures against the incoming and outgoing data packets, when the well-known attacks are modified/transformed differently, the NIDS might fail due to its inability to match them in its signature database. Sometimes, we call these transformed attacks (but all from one single original attack signature, for the purpose of IDS evasion) "***polymorphic attacks***". In this paper, we propose a new solution to accurately identify one particular type of polymorphic attacks, known as **polymorphic shellcode**. Due to the space limitation, solutions for dealing with other types of polymorphic attacks are discussed in [1].

In polymorphic shellcode attacks, the attacker can choose an unknown encryption algorithm to encrypt the attack code and include the decryption code as part of the attack packet. The trick here is to utilize an existing buffer-overflow exploit and to set the "return" memory address on the over-flowed stack, to the entrance point of the decryption code module. The attacker can transform every other bit in the packet payload to avoid being detected by a signature-based NIDS, but a critical constraint exists on the range of the "return" memory address that can be twisted.

Our solution, ***Buttercup***, is simply to identify the ranges of the possible return memory addresses for existing buffer-overflow vulnerabilities, and if a packet contains such addresses, a red/yellow flag might be raised. For the evaluation of false positives, we have modified SNORT, an open-source signature-based NIDS, and selected 19 exploits to test against 55 different TCPdump traffic files. For one of our range matching algorithms, the false positive rate is as low as 0.01%, while the rates for other simpler algorithms are all below 1.13%. Hence, with Buttercup, we can drop all worm packets based on the known buffer-overflow vulnerabilities, while dropping only 0.01% of the good packets in the Internet.

The rest of this paper is organized as follows. Section 2 describes buffer overflows and polymorphic shellcode. Section 3 presents Buttercup, our proposed solution to detect polymorphic shellcode. The simulation results and analysis are presented in Section 4. Section 5 presents some of the related work and we finally conclude and provide suggestions for future work in Section 6.

2. Background

In this section, we briefly describe buffer overflows and polymorphic shellcode.

2.1. Buffer Overflow

On many C implementations, writing past the end of an array causes the execution stack to get corrupted. The stack is used to dynamically allocate the local variables used in functions, to pass parameters to functions, and to return values from functions as shown in Figure 1. It also stores the return addresses for function calls and this is what makes it vulnerable.

A buffer overflow [5,6] is the result of stuffing more data into a buffer than it can handle. The data that does not fit into the allocated buffer space overwrites the bytes after the allocated buffer space in the stack, including the return address. When the stored return address on the stack gets replaced by some arbitrary value due to a buffer overflow, the function returns and tries to read the next instruction from that address. This results in a segmentation violation.

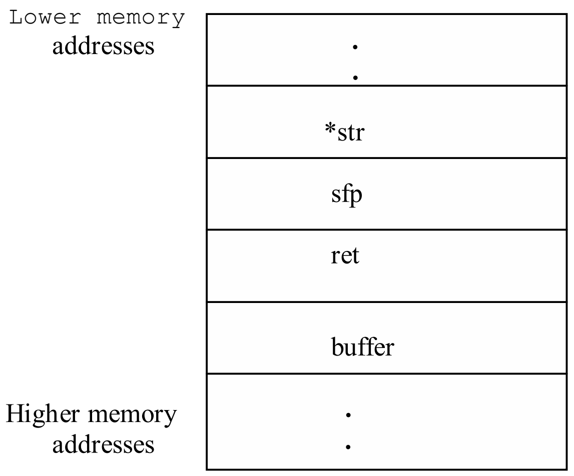

Figure 1: Structure of a stack

By sending a string that overflows a buffer such that it fills the return address on the stack, with an address where arbitrary code is placed, one could use the buffer overflow vulnerability to execute one's own code. Most of the time, it involves spawning a root shell and then executing commands on it. The hexadecimal representation, of the commands in machine language, which are used to spawn a shell, is sent as a part of the string that is used to overflow the buffer. This string is thus called the ***shell code***.

Importance of return address

When a buffer overflow vulnerability is discovered, the most important requirement for an exploit to work is to get the return address right. A buffer overflow exploit involves loading shellcode onto the buffer and overwriting the return address variable of the stack frame so it points back into the buffer. Hence, the address placed in the return address variable would be a value within the address space allocated for the process i.e., the shellcode is executed off the stack. If the shellcode occupies a portion of memory other than the memory space of the program we are trying to exploit, a segmentation violation occurs.

The problem faced when trying to overflow the buffer of another program is to figure out at what address the buffer (and thus the exploit code) will be. One possible solution is to pad the front of the buffer overflow with NOP instructions that perform NULL operations. If the return address points anywhere in the string of NOPs, they will just get executed until they reach the shell code. Assuming the stack starts at 0xFF, that S stands for shell code, and that N stands for a NOP instruction, the new stack would look like this:

```
bottom of EEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF  top of
 memory   123456789AB  CDEF  0123  4567  89AB  CDEF  memory
           buffer       sfp   ret    a     b     c

 <-----   [NNNNSSSSSSS][0xE2][0xE2][0xE2][0xE2][0xE2]
            ^                     |
            |_____________________|
```

Figure 2: Structure (horizontal representation) of a buffer overflow exploit.

For remote overflow exploits, the intruder can compile and analyze the service program on his machine to get a rough idea of the correct address, the exact value of which depends on the environment variables that the user has set. Hence, in order to increase the chances of pointing to the exploit code, a string of NOPs are placed, as in local exploits.

2.2. Polymorphic shellcode

Polymorphic shellcode [7] is basically a functionally equivalent form of a buffer overflow exploit with a different signature on the network. As it hits the target machine, it reassembles, having eluded the IDS [8].

A well-known tool that generates polymorphic shellcode is a polymorphic buffer-overflow engine called **ADMutate** [9]. An attacker feeds the ADMutate a buffer overflow exploit to generate hundreds or thousands of functionally equivalent exploits [10]. This is accomplished by using simple encryption techniques, along with the substitution of functionally equivalent machine-language instructions. This confuses many IDS tools (including Snort) that search for the familiar NOP sled or the known machine-language exploit included in buffer overflows, as ADMutate dynamically modifies these elements.

A buffer overflow attack script consists of three parts, a set of NOPs, the shellcode, and the return address in the form [NNNN][SSSS][RRRR]. In polymorphic shellcode, the NOPs are replaced by a random mix of no-effect instructions and the shellcode is encrypted differently each time, thus making signature-based detection by an NIDS, that looks for NOPs or certain strings within the shellcode, impossible. Having generated encoded shellcode and substituted NOPs, ADMutate then places the decoder in the code. The shellcode is then of the form [NNNN][DDDD][SSSS][RRRR], where "D" represents the decoder. It is not possible to detect the decoder either since techniques such as multiple code paths, non-operational pad instructions, out-of-order decoder generation and randomly generated instructions make it look different each time.

The only part of the script that remains constant through each instance of a buffer overflow attack is the return address. In fact, even the return address is modified by modulating its least significant bit, but when this is done, sometimes, the address may no longer be valid when it hits the target. Hence, we intend to use this part of a buffer overflow attack script in enabling an IDS to detect polymorphic shellcode.

3. ButterCup: an IDS architecture against Attack Polymorphism

As we saw above, one solution to the problem of determining the return address to exploit a buffer overflow vulnerability is, to pad the front of the shellcode with NOP instructions. If the return address points anywhere within the NOPs, they will just get executed till the exploit code is reached. Using this method, the exploit might work for a certain range of values since the return address could point anywhere within the string of NOPs.

Hence, for every buffer overflow vulnerability, the return address is overwritten with a value, which can only lie within a certain range of values (the process' address space). By determining the address range for a particular buffer overflow exploit and looking for values that lie within this range, in incoming packets, we hope to detect the exploit.

Determination of address range values

Determining a lower limit and an upper limit within which the return address can fall can reduce the range of values, which need to be checked, further. The lower limit would be the address at which the buffer starts since the string we send to overflow starts at the start of the buffer.

Let's take a look at the example we saw above (Figure 2). In this example, since the buffer starts at address 0xE1, the lower limit of our address range would be 0xE1. Since the string in the example can be changed by increasing or decreasing the number of NOPs, we try to determine a suitable range that would help us detect the attacks even if the number of NOPs is changed.

In addition to having the form [NNNN][SSSS][RRRR], the attack script can also be of the form [RRRR] [NNNN][SSSS], especially in cases where the buffer is small. In this case, the buffer and the return address field are filled up with the address where the shellcode is to be found. The attack in this case looks like this:

```
bottom of EEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF   top of
memory    0123456789AB  CDEF  0123  4567  89AB  CDEF   memory
            buffer        sfp   ret   a     b     c
 <------  [0xF80xF80xF8][0xF8][0xF8][NNNN][SSSS][SSSS]
                                  |         ^
                                  |_________|
```

Figure 3: Structure (horizontal representation) of a different form of a buffer overflow exploit.

In the case where the attack is of the form [NNNN][SSSS][RRRR], the upper limit would be the (address of the return address field - length of the shellcode). In the case where the attack is of the form [RRRR] [NNNN][SSSS], the upper limit would be (bottom of stack – length of the shellcode).

We have thus determined that there is definitely an upper limit and a lower limit within which the return address of the shellcode of a buffer overflow exploit should fall. The only task left now is determining the address range. This range of values can aid in the detection of a particular buffer overflow exploit.

In some buffer overflow exploits targeting Windows systems (such as Code Red and Slammer worms), the return address does not point directly into the stack. It points to a CALL or JMP instruction in a DLL, which in turn points back into the stack. Buttercup can still be used to detect the address range of the CALL or JMP instruction in the particular DLL.

3.1. Implementation of proposed solution

We implemented the solution by including a new keyword in Snort-2.0.0 called "range". We call this implementation of our solution in Snort, **Buttercup**. In Buttercup, a new detection plugin file named sp_range_check, was included in Snort version 2.0.1, which takes 32 bits at a time from the payload of the incoming packet, starting from the first byte, and compares it against the two values provided as the values for the "range" keyword. If it lies within the range, then the buffer overflow alert corresponding to those return address values is generated. Else, the 32 bits starting from the next byte are compared with the two values.

The range values are obtained by getting the return address used for a particular buffer overflow exploit and initially, the lower limit is taken to be a value –200 from the return address value and the upper limit is a value +200 from the return address value. In this way, the entire packet is analyzed.

We also obtained a range for the Microsoft Windows RPC Buffer Overflow vulnerability, which was exploited by the very recent ***Blaster*** worm that caused a lot of damage worldwide. We obtained this range by studying some exploit codes for this vulnerability. The lower range and higher range values were found to be 0x77d73713 and 0x77f92b63 respectively. However, a rule for detecting this attack wasn't added to our rules file before we performed all the tests, since this vulnerability was exploited only recently.

3.2. Steps proposed to reduce false positives

In order to reduce the number of false positives further, 2 other keywords, 'rangeoffset' and 'rangedepth' were introduced. The value provided with the 'rangeoffset' indicates the starting point in the packet payload from where the 32-bit values are checked. The 'rangedepth' sets the maximum search depth for the range check function to search from the beginning of its search region. The 'rangeoffset' and 'rangedepth' options are used as modifiers to rules using the 'range' option keyword.

By carefully studying the buffer overflow exploit code, we can determine the part of the shellcode in which the return address is placed and thus provide values for the above two option keywords. We also used the ‘dsize’ option keyword, already implemented in Snort, in order to flag alerts only for those packets that have payloads whose length falls within a given range in addition to containing the particular return address values. Using these three additional keywords, the number of false positives was brought down considerably. An example of a Snort rule containing the ‘dsize’, ‘rangeoffset’ and ‘rangedepth’ keywords, in addition to the ‘range’ keyword, is as follows:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS any (msg:"MSSQL2000 remote
UDP exploit"; range:"|42ae1000-42b0caa4|"; dsize:475<>550; rangeoffset:97;
rangedepth:20; )
```

The above rule is used for detecting attacks that exploit the MS SQL 2000 buffer overflow vulnerability. We detect these attacks by looking for values lying between 42ae1000 and 42b0caa4, only in packets whose size falls in the range 475-550. Also, we look for these values starting from the 97th character in the packet payload and only within 20 characters from the starting point. As we can see from this example, this greatly reduces the amount of processing that the IDS needs to do since it looks for the address values only in certain packets and only in certain portions of those packets instead of searching all the packet payloads from start to finish.

However, due to the complex nature of the shellcodes, we could not get values for the ‘dsize’, ‘rangeoffset’ and ‘rangedepth’ keywords for all of the exploits. We hope that through deeper evaluation, these values can be obtained for all the exploits.

[11] is a complete version of the paper, which includes all the rules we have in included in the rules file for detecting polymorphic shellcode.

4. Simulation and Analysis

In this section, we describe the various tests that were performed on Buttercup in order to compare its performance with the original version of Snort. In order to determine the performance of our IDS architecture against polymorphic shellcode, various parameters, such as ‘range’ and ‘dsize’ values, were changed in our implementation and the performance of Snort observed in terms of processing time and percentage of alerts generated.

Simulation

For our simulation, 55 real TCPdump files of network traffic were obtained from the MIT Lincoln Laboratory IDS evaluation Data Sets. These TCPdump files were provided as input to Buttercup, which included the ‘range’, ‘dsize’, ‘rangeoffset’ and ‘rangedepth’ keywords and 19 new rules. Buttercup was then tested for false positives on each of these files.

Table 1 depicts the results obtained in the form of the percentage of alerts generated i.e. (no. of alerts / no. of packets) when several TCPdump files were taken as input by Buttercup. In order to observe how the number of alerts would change when the range values were changed, we present the percentage of alerts for range values of +-50, +-100, +-200, +-250, +-300, +-400 and +-500 in table 3 below.

TCPdump files	*RANGE*						
	+-50	*+-100*	*+-200*	*+-250*	*+-300*	*+-400*	*+-500*
inside.tcpdump-00	0.3488	0.6213	0.6664	0.6746	0.6927	0.7165	0.7973
outside.tcpdump-00	0.3967	0.6727	0.7276	0.7356	0.7541	0.7797	0.8174
sampledata01-dump	0.1928	0.2066	0.2203	0.2203	0.2203	0.2203	0.2617
tcpdins-00	0.1617	0.2846	0.3113	0.3176	0.3296	0.3421	0.3684
tcpdwk1mon-98	0.7181	0.9904	1.1237	1.1336	1.2203	1.2704	1.3104
tcpdwk1tue-98	0.6796	0.9422	1.0721	1.0804	1.1566	1.2009	1.2415
tcpdinswk1wed-99	0.2678	0.4734	0.5210	0.5284	0.5431	0.5634	0.6031
tcpdinswk2mon-99	0.2670	0.4439	0.4927	0.4996	0.5172	0.5393	0.5749

Table 1: Percentage of alerts generated by Buttercup for various address ranges and TCPdump files

Table 2 again depicts the change in the percentage of alerts, but this time, comparison is made between the cases where the rules have just the 'range' keyword alone, the rules have the 'dsize' keyword, with symbol '<>', in addition to the 'range' keyword, the rules have the 'range', 'dsize', 'rangeoffset' and 'rangedepth' keywords and the symbol '>' is used with the 'dsize' keyword.

TCPdump files	*Snort versions*				
	BC-range	*BC-range-dsize<>*	*BC-range-dsize<> -RO-RD*	*BC-range-dsize>*	*BC-range-dsize>-RO-RD*
Inside.tcpdump-00	0.6664	0.0144	0.0138	0.5293	0.2468
Outside.tcpdump-00	0.7276	0.0245	0.0249	0.5987	0.2833
sampledata01-dump	0.2203	0	0	0.2203	0.0275
tcpdins-00	0.3113	0.0057	0.0051	0.2408	0.1039
tcpdwk1mon-98	1.1237	0.0077	0.0093	0.9642	0.3240
tcpdwk1tue-98	1.0721	0.0075	0.0092	0.9275	0.3229
tcpdinswk1wed-99	0.5210	0.0106	0.0099	0.4083	0.1902
tcpdinswk2mon-99	0.4927	0.0107	0.0098	0.3845	0.1710

Table 2: Percentage of alerts generated for various versions of Snort for a range value of +-200.

where
BC-range – Buttercup with only 'range' keyword and range of +-200.
BC-range-dsize<> – Buttercup with range of +-200 and 'dsize' <> values (values derived from size of shellcode).
BC-range-dsize<>-RO-RD – Buttercup with 'range' of +-200 and 'dsize' <> values (values derived from size of shellcode) and 'rangeoffset' and 'rangedepth' keywords included.
BC-range-dsize> – Buttercup with 'range' of +-200 and 'dsize' > value (size of shellcode obtained from buffer overflow exploits).
BC-range-dsize>-RO-RD - Buttercup with range of +-200 and 'dsize' > value (size of shellcode obtained from buffer overflow exploits) and 'rangeoffset' and 'rangedepth' keywords included.

Since, in the above two cases, we only want to concentrate on how many alerts Buttercup generates due to the buffer overflow rules we have added, we only include our rules file my.rules in the configuration file, snort.conf.

Finally, Table 3 depicts the change in the processing times of original Snort and Buttercup. In this case, since we are concerned about how our modified Snort compares with the unmodified Snort, we include all the rules files in the configuration file, snort.conf.

TCPdump files	*Snort versions*				
	No. of Packets	*Snort-2.0.0*	*BC-range*	*BC-range-dsize<>*	*BC-range-dsize<>-RO-RD*
phase-1-dump-00	40	0.311	0.301	0.308	0.314
phase-2-dump-00	158	0.12466	0.21532	0.12144	0.13130
phase-3-dump-00	225	0.10749	0.19784	0.12670	0.42505
phase-4-dump-00	520	0.54868	0.36267	0.33663	1.63335
phase-5-dump-2-00	954	0.30983	0.36433	0.35798	0.48870
sampledata01-dump	14523	1.33127	5.76940	3.51988	3.43390
tcpdwk3mon-98	793256	73.11217	215.2422	219.2653	230.4325
tcpdwk3tue-98	393566	37.42337	135.6459	125.3525	149.3899

Table 3: Processing times (in seconds) of different versions of Snort
where
Snort-2.0.0 - original snort-2.0.0 with all rules files included in snort.conf.
BC-range – Buttercup with all rules files included in snort.conf and only 'range' keyword with a range of +-200.
BC-range-dsize<> – Buttercup with all rules files included in snort.conf and 'range' keyword with a range of +-200 and 'dsize' (<> values) keyword.

BC-range-dsize<>-RO-RD – Buttercup with all rules files included in snort.conf and 'range' keyword with a range of +-200 and 'dsize' (<> values), 'rangeoffset' and 'rangedepth' keywords.

Figure 4 is a graphical representation of the results presented in Table 1. Figure 5 is a bar graph representing the results presented in Table 2.

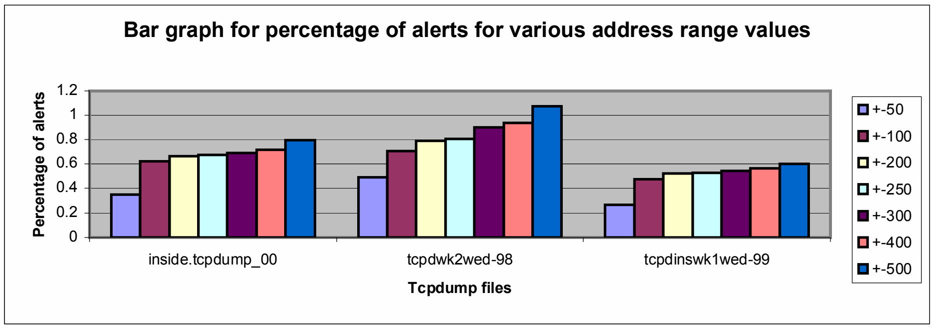

Figure 4: Bar graph showing percentage of alerts for various address range values for 3 TCPdump files.

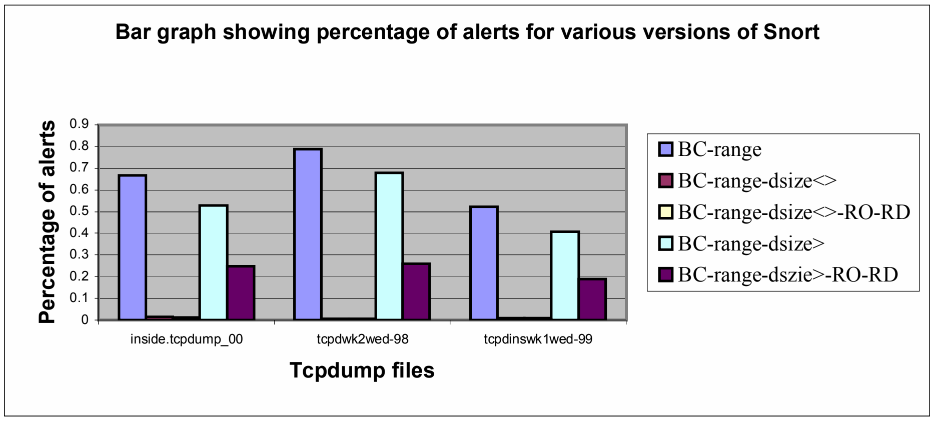

Figure 5: Bar graph showing percentage of alerts for various versions of Snort for 3 TCPdump files.

Performance

In Table 2, we see that the number of alerts generated by Buttercup is far greater than those generated by Snort. This is most probably due to a large number of false positives generated by Buttercup since the version of Buttercup used here does not contain the 'dsize', 'rangeoffset' and 'rangedepth' keywords.

From Table 3, we observe that as the address range values are increased from +-50 to +-500, there is a corresponding rise in the percentage of alerts generated. From Figure 4, which shows the bar graph comparing the percentage of alerts for the various address range values, we can see that the percentage of alerts for the various range values doesn't change too much. For some variations in range values, such as between the +-200 and +-250, the percentage of alerts is almost constant. Since increasing the range values doesn't increase the number of alerts too much, we can conclude that this solution of detecting buffer overflow attacks is feasible.

Table 4 compares the percentage of alerts generated for different versions of Buttercup for various TCPdump files. It can be observed from Figure 5, which shows the bar graph more clearly depicting the percentage of alerts for the various versions of Snort, that the percentage of alerts is the greatest when only the 'range' keyword is used, is lesser when the 'dsize' (with symbol '<>') keyword is included and is the least when the 'rangeoffset' and 'rangedepth' keywords are also included. Hence, by studying a buffer overflow exploit carefully and determining the size of the shellcode and the part of the shellcode that contains the return addresses, the number of false positives can be brought down considerable, thereby, enabling a more accurate detection of buffer overflow attacks.

However, the drawback of narrowing the payload in which to look for address ranges is that some of the buffer overflow attacks may not be detected if there is a miscalculation in the 'rangeoffset' and 'rangedepth' values or if the shellcode is modified considerably. The same behavior repeats for the cases where symbol '>' is used with the 'dsize' keyword, but the percentage of alerts is far greater than those where symbol '<>' is used. Hence, calculating the range in which the size of the shellcode falls helps us determine buffer overflow attacks more accurately than just looking for packets that are larger than a given size. It must be pointed out here that there is definitely a range for the size of the shellcode, since there aren't too many ways of modifying the size of a particular shellcode other than varying the number of NOPs included.

Table 5 compares the processing times of four different versions of Snort for different TCPdump files and also lists the number of packets in each file. It can be observed that the TCPdump files used aren't the same as the ones used in the above three cases. This is because these are the smaller TCPdump files, which didn't generate too many alerts and hence, are unsuitable for determining the performance of Snort in the first three cases. These files are, however, useful in this case, since the larger files cannot be used for determining the processing times because all the rules files are included in the snort.conf (since the performance is compared to the unmodified Snort with all its rules files included). Since all the rules files are included, when the large TCPdump files are used, too many alerts are generated and Snort halts.

It can be observed that the processing time increases sharply when the 'range' keyword is included as compared to the unmodified version. However, when the 'dsize' keyword is included, the processing time decreases since only packets whose payload size falls within a specific payload are searched for the address ranges. This considerably brings down the processing time. We would expect the processing time to decrease further when the ‘rangeoffset’ and ‘rangedepth’ keywords are added since the payload in which to look for address ranges is further narrowed down, but this doesn’t happen. In fact, the processing time increases slightly. It can be concluded that this happens due to the extra processing involved with the inclusion of two new keywords.

Also, it should be noted that this behavior is true for most of the TCPdump files, but, as can be observed from Table 5, for some of them, the results are different. This is due to the fact that due to the complexity of some of the exploit codes, the ‘rangeoffset’ and ‘rangedepth’ values for all the rules could not be determined. Hence, some of the rules have just the ‘range’ and ‘dsize’ keywords, thereby leading to the inconsistency in the results of the TCPdump files. A final observation is that as the size of the TCPdump files increases, the processing time increases significantly.

5. Related Work

Several solutions have been proposed to deal with the problem of buffer overflow attacks. Some of them try to detect buffer overflow vulnerabilities and prevent them from being exploited while others try to detect an exploit before it causes any damage to the target system.

StackGuard [12] is a simple compiler technique, that is implemented as a small patch to gcc that places a “canary” word next to the return address when the function is called and before the function returns, detects the change of the return address by checking to see if the canary word has been changed. This method virtually eliminates buffer overflow vulnerabilities with only modest performance penalties.

Return Address Defender (RAD) [13] is a simple compiler patch that automatically creates a safe area to store a copy of return addresses and automatically adds protection code into applications that it compiles to defend programs against buffer overflow attacks. RAD does not modify the source code or the layout of stack frames, so the binary code it generates is compatible with exisiting libraries and other object files.

A tool for the automatic detection of buffer overrun vulnerabilities in object code is presented in [14]. It searches the file systems for critical programs and tests them individually against buffer overruns. For the vulnerable ones, the exploit is built and executed against the program. The system administrator is then notified of the vulnerability.

[15] describes a technique for finding potential buffer overrun vulnerabilities through the use of static analysis. The technique is to formulate detection of buffer overruns as an integer range analysis problem. The advantages of this static analysis are that security bugs can be eliminated before code is deployed and new remotely exploitable vulnerabilities can be found and the disadvantage is it generates a large number of false positives and a small number of false negatives.

An approach that accurately detects buffer overflow code in the request's payload by concentrating on the *sledge* (e.g. string of NOPs) of the attack has been presented in [16]. This method works by performing abstract execution of the payload to identify sequences of executable code with virtually no false positives.

Finally, several solutions to detect polymorphic solution using an IDS have been proposed in [17]. These solutions include shellcode payload decryption, signatures to detect the decrypter engine, decrypter engine emulation and NOP section detection. These pros and cons of the various methods are pointed out and it is concluded that NOP section detection is the best technique.

All these papers, except the last one, do not deal with polymorphic shellcode detection using a signature-based NIDS, which is what Buttercup proposes. Most of them are solutions that need to be implemented in the compiler or detecting vulnerabilities in programs themselves and preventing them from being exploited.

6. Conclusion

In this paper, we focus on the weakness of signature-based Network Intrusion Detection Systems in detecting polymorphic attacks. When a regular attack, for which an NIDS already has a signature available in its signature database, is modified or transformed, the IDS might fail to detect it.

We present a new solution here called "Buttercup" to counter against any attacks based on buffer-overflow vulnerabilities. We have implemented our idea in SNORT, and included 19 return address ranges of buffer-overflow vulnerabilities. We introduce three new keywords in SNORT namely 'range', 'rangeoffset' and 'rangedepth' and used a keyword already existing in Snort named 'dsize' to detect packets with return address values potentially lying within specific ranges.

For evaluation, with a suite of tests against 55 TCPdump traces, the false positive rate for our best algorithm is as low as 0.01%. This indicates that, potentially, Buttercup can drop 100% worm and other attack packets on the wire while only 0.01% of the good packets will be sacrificed. We believe that our solution is simple and practical as normally, a vulnerability is known long before the worms based on it are developed and launched.

Currently, Buttercup needs an accurate input of the return address ranges to be effective. For high-speed Internet worms, we are currently developing solutions such that Buttercup can intelligently discover address ranges for existing buffer overflow vulnerabilities, which haven't been exploited yet.

The future work for this project includes false negative analysis i.e. we need to determine whether this technique detects real-time traffic containing an attack. We also need to look into how this technique can be applied to detect other kinds of blended attacks [18], which include off-by-one overflows, heap overflows and attacks exploiting format string vulnerabilities. We believe that ButterCup can be used to detect each of the above exploits by developing suitable address ranges through careful analysis of the attack techniques used.

Acknowledgment

This research is sponsored by NSF under grant nos. 0113388, 0220147, and ITRI.

References

[1] "Network-Based Detection of Polymorphic Attacks", MS thesis, Computer Science Department, UC Davis.
[2] Martin Roesch, "Snort-Lightweight Intrusion Detection for Networks".
[3] Martin Roesch, "Snort Users Manual", Snort Release: 1.9.x.
[4] Aleph One. Smashing the Stack for fun and profit. *Phrack Magazine, 49(14), 1996.*
[5] "Buffer Overflows Demystified", http://www.enderunix.org/docs/eng/bof-eng.txt
[6] Lefty, "Buffer Overruns, what's the real story?", http://destroy.net/machines/security/stack.nfo.txt
[7] K. Timm, "IDS Evasion Techniques and Tactics", http://online.securityfocus.com/infocus/1577
[8] E. Messmer, "Put to the test", http://www.nwfusion.com/news/2002/0415idsevad.html
[9] "ADMuate Readme", http://www.ktwo.ca/readme.html
[10] E. Skoudis, "Sneaking Past IDS", http://www.infosecuritymag.com/2002/jul/sneaking.shtml
[11] http://wwwcsif.cs.ucdavis.edu/~pasupula/ Buttercup-paper.doc
[12] Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. *In Proceedings of the 7th USENIX Security Conference, January 1998.*
[13] T. Chiueh, F. Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. *International Conference on Distributed Computing Systems (ICDCS), Phoenix, Arizona, USA, April 2001.*
[14] D. Bruschi, E. Rosti, R. Banfi. A tool for pro-active defense against the buffer overrun attack. *In Proceedings of ESORICS 98, 5th European Symposium on Research in Computer Security.*
[15] D. Wagner, J. S. Foster, E. A. Brewer, and A Aiken. A first step towards automated detection of buffer overrun vulnerabilities. *In Proceedings of the Network and distributed system security symposium, February 2000.*
[16] T. Toth, C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. *RAID 2002, LNCS 2516, pp.274-291, 2002.*
[17] "Polymorphic Shellcodes vs. Application IDSs", NGSEC White Paper, http://www.ngsec.com
[18] E. Chien, P. Szor. Blended Attacks Exploits, Vulnerabilities and Buffer-Overflow Techniques in Computer Viruses. *Virus Bulletin Conference 2002.*