BLENDED ATTACKS EXPLOITS, VULNERABILITIES AND BUFFER-OVERFLOW TECHNIQUES IN COMPUTER VIRUSES

Eric Chien and Péter Ször
Symantec Corporation 2500 Broadway, Suite 200 Santa Monica, CA 90404, USA
Tel +1 310 453 4600 • Fax +1 310 453 0636 • Email echien@symantec.com, pszor@symantec.com

ABSTRACT

Exploits, vulnerabilities, and buffer-overflow techniques have been used by malicious hackers and virus writers for a long time. However, until recently, these techniques were not common place in computer viruses. The CodeRed worm was a major shock to the antivirus industry since it was the first worm that spread not as a file, but solely in memory by utilizing a buffer overflow in Microsoft IIS. Many antivirus companies were unable to provide protection against CodeRed, while other companies with a wider focus on security were able to provide solutions to the relief of end users.

Usually new techniques are picked up and used by copy cat virus writers. Thus, many other similarly successful worms followed CodeRed, such as Nimda and Badtrans.

In this paper, the authors will not only cover such techniques as buffer overflows and input validation exploits, but also how computer viruses are using them to their advantage.

Finally, the authors will discuss tools, techniques and methods to prevent these blended threats.

DEFINITION OF BLENDED ATTACK

A blended threat is often referred as a 'blended attack'. Some people refer to it as 'combined attacks' or 'mixed techniques'. We are not attempting to make a strong definition for this term, but we wish our readers to understand that we use the term 'blended attack' in the context of computer viruses where the virus exploits some sort of security flaw of a system or application in order to invade new systems. A blended threat exploits one or more vulnerabilities as the main vector of infection and may perform additional network attacks such as a denial of service against other systems.

INTRODUCTION

Security exploits, commonly used by malicious hackers, are being combined with computer viruses resulting in a very complex attack that in some cases goes beyond the general scope of anti-virus software.

In general, a large gap has existed between computer security companies such as intrusion detection and firewall vendors and anti-virus companies. For example, many past popular computer security conferences did not have any papers or presentations dealing with computer viruses. Thus, apparently some computer security people do not consider computer viruses seriously as part of security or ignore the relationship between computer security and computer viruses. When the CodeRed worm appeared there was obvious confusion about which genre of computer security vendors could prevent, detect and stop the worm. Some anti-virus researchers argued that there was nothing that they could do about CodeRed, while others tried to solve the problem with a various sets of security techniques, software and detection tools to support their customers' needs.

Interestingly, such intermediate solutions were often criticized by anti-virus researchers. Instead of realizing the affected customers' need for such tools, some anti-virus researchers suggested that there was nothing to do but to install the security patch.

Obviously, this step is very important in securing the systems. However, the installation of a patch on thousands of systems may not be easy to deliver at large corporations. Furthermore, corporations may have the valid fear that new patches could introduce a new set of problems, compromising system stability.

CodeRed (and blended attacks in general) is a problem that needs to be taken care of by anti-virus vendors as well as by other security product vendors so that multi-layered security solutions can be delivered in a combined effort to deal with blended attacks.

BACKGROUND

The origin of blended attacks begins in November, 1988. 1988 was the year that the Morris worm was introduced. The Morris worm exploited flaws in standard applications of BSD systems:

- It tried to utilize remote shell commands to attack new machines by using rsh from various directories. It demonstrated the possibility of cracking password files. The worm attempted to crack passwords in order to get into new systems. This attack was feasible because the password file was accessible and readable by everyone. Although the password file was encrypted, someone could attempt to encrypt test passwords and then compare them against the encrypted ones. The worm used a small dictionary of passwords that the author of the worm believed to be common or weak. Looking at the list of passwords in the author's dictionary, we have the impression that this was not the most successful of the worm's attacks.
- If the previous step failed, the worm attempted to use a buffer-overflow attack against VAX based systems running a vulnerable version of fingerd. (More details on this attack are available later on.) This resulted in the automatic execution of the worm on a remote VAX system. The worm was able to execute this attack from either VAX or Sun systems, but the attack was only successful against targeted VAX systems. The code was not in place to identify the remote OS version and thus, the same attack was used against the fingerd program of Suns running BSD. This resulted in a core dump (crash) of fingerd on targeted Sun systems.
- The Morris worm also utilized the DEBUG command of the sendmail application. This command was only available in early implementations of sendmail. The DEBUG command made it possible to execute commands on a remote system by sending an SMTP (Simple Mail Transfer Protocol) message. This command was a potential mistake in functionality and was removed in later versions of sendmail. When the DEBUG command was sent to sendmail, someone could execute commands as the recipient of the message.

Nevertheless, the worm was not without bugs. Although the worm was not deliberately destructive, it overloaded and slowed down machines so much that it was very noticeable after repeated infections occurred.

Thirteen years later, in July, 2001, CodeRed repeated a very similar set of attacks against vulnerable versions of Microsoft Internet Information Server (IIS) systems. Using a well crafted buffer overflow technique, the worm executed copies of itself (depending upon its version) on Windows 2000 systems running vulnerable versions of Microsoft IIS. The slowdown effect was similar to that of the Morris worm.

Further information on the buffer-overflow attacks is made available in this paper (without any working attack code).

TYPES OF VULNERABILITY

Buffer Overflows

Buffers are data storage areas, which generally hold a pre-defined amount of finite data. A buffer overflow occurs when a program attempts to store data into a buffer, where the data is larger than the size of the buffer.

For example, imagine an empty 33 cl. glass. This is analogous to a buffer. This buffer (empty glass) can store 33 cl. of liquid (data). Now, imagine that I wish to transfer a pint, which is about 47 cl., of beer from my full pint glass into the empty 33 cl. glass. As I begin to fill the glass (buffer) with beer (data), everything is fine until the end when beer begins to spill over the glass and onto the table. This is an example of an overflow. Clearly, such an overflow is bad for beer and unfortunately, even worse if such vulnerabilities exist in computer programs.

When the data exceeds the size of the buffer, the extra data can overflow into adjacent memory locations, corrupting valid data and possibly changing the execution path and instructions. The ability to exploit a buffer overflow allows one to possibly inject arbitrary code into the execution path. This arbitrary code could allow remote system level access, giving unauthorized access to not only malicious hackers, but also to replicating malware.

Buffer overflows are generally broken into multiple categories based both on ease of exploitation and historical discovery of the technique. While there is no formal definition, buffer overflows are, by consensus, broken into three generations. First generation buffer overflows involve overwriting stack memory; second generation overflows involve heaps, function pointers, and off-by-one exploits; and finally, third generation overflows involve format string attacks and vulnerabilities in heap structure management.

For simplicity, the following explanations will assume an Intel CPU architecture, but the concepts can be applied to other processor designs.

First Generation

First generation buffer overflows involve overflowing a buffer that is located on the stack.

Overflowing A Stack Buffer

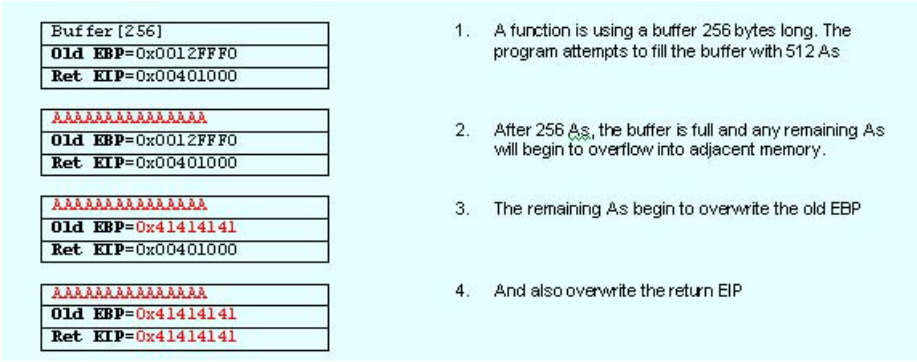
For example, the following program declares a buffer that is 256 bytes long. However, the program attempts to fill it with 512 bytes of the letter ‘A’ (0x41).

```
int i;
void function(void)
{
 char buffer[256]; // create a buffer

 for(i=0;i<512;i++)     // iterate 512 times
    buffer[i]='A';     // copy the letter A
}
```

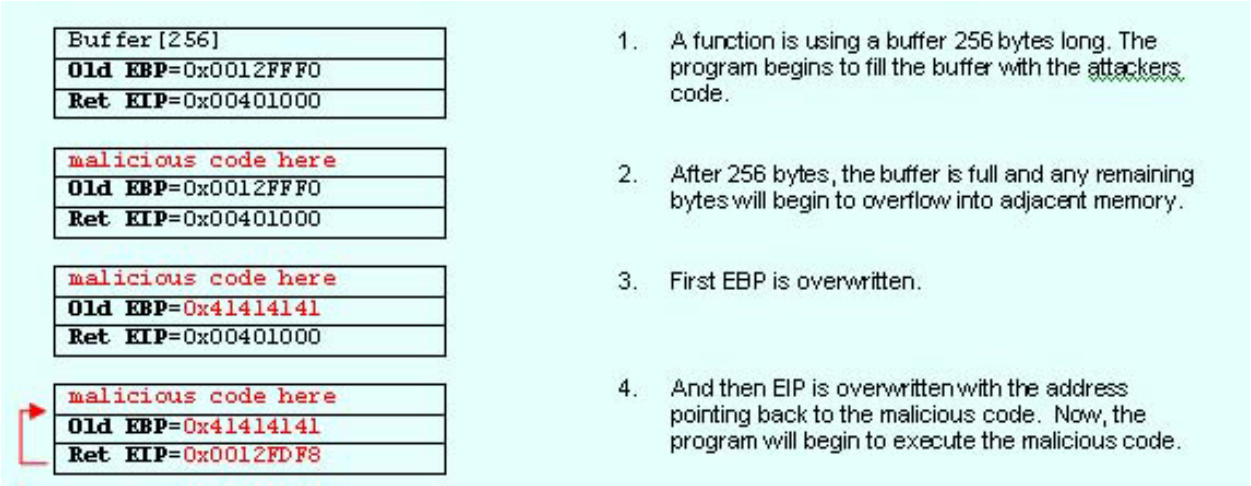
The diagram opposite (top of page) illustrates how the EIP (where to execute next) is modified due to the program overflowing the small 256 byte buffer. When data exceeds the size of the buffer, the data overwrites adjacent areas of data stored on the stack including critical values such as the instruction pointer (EIP), which define the execution path. By overwriting the return EIP, an attacker can change what the program should execute next.

Exploiting A Stack Buffer

Overflow Instead of filling the buffer full of As, a classic exploit will fill the buffer with it’s own



malicious code. Also, instead of overwriting the return EIP (where the program will execute next) with random bytes, the exploit will overwrite EIP with the address to the buffer which is now filled with malicious code. This causes the execution path to change and causes the program to execute injected malicious code.



While the above example demonstrates a classic stack-based (first generation) buffer overflow, there are variations. The exploit utilized by CodeRed was a first generation buffer overflow that is more complex and is described below.

Causes of Stack-based Overflow Vulnerabilities

Stack-based buffer overflows are caused by programs that do not verify the length of data being copied into a buffer. This is often caused by using common functions that do not limit the amount of data that is copied from one location to another.

For example, strcpy is a C programming language function that copies a string from one buffer to another. However, the function does not verify if the receiving buffer is large enough and thus, an overflow may occur. Many such functions have safer counterparts, such as strncpy which takes an additional count parameter specifying the number of bytes that should be copied. On BSD systems, even safer versions such as strlcpy are available.

Of course, if the count is larger than the receiving buffer, an overflow can still occur. Programmers often make counting errors and utilize counts that are off by one byte. This can result in a second generation overflow called off-by-ones.

Second Generation

Off-By-One Overflows

Programmers who attempt to use relatively safe functions such as strncpy do not necessarily make their programs much more secure from overflows. Errors in counting the size of the buffer can occur usually resulting in a single byte overflow known as an off-by-one.

Consider the following program where the programmer has mistakenly utilized 'less than or equal to' rather than simply 'less than'.

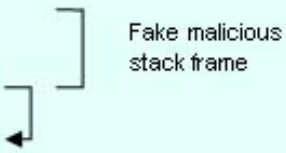
```
#include <stdio.h>

int i;
void vuln(char *foobar)
{
     char buffer[512];

     for (i=0;i<=512;i++)
          buffer[i]=foobar[i];
}

void main(int argc, char *argv[])
{
     if (argc==2)
          vuln(argv[1]);
}
```

In this case, the stack will appear as:

Address	Value
0x0012FD74	Buffer[512]
…	…
…	…
0x0012FF74	**Old EBP**=0x0012FF80
	Ret EIP=0x00401048

While one is unable to overwrite EIP because the overflow is only one-byte, one can overwrite the least-significant byte of EBP because it is a little-endian system. For example, if the overflow byte is set to 0x00, the Old EBP will be set to 0x0012FF00.

Address	Value
0x0012FD74	…
…	…
0x0012FF00	Fake Local Variables
	Fake EBP
	Fake return EIP
…	
…	Malicious Code
0x0012FF74	Old EBP=0x0012FF**00**
	Old EIP=0x00401048



Depending on the code, the fake EBP may be used in a variety of manners. Often, a mov esp,

ebp will follow causing stack frame pointer to now be in the location of buffer. Thus, buffer can be constructed to hold local variables, an EBP, and more importantly an EIP that redirects to the malicious code.

The local variables of the fake stack frame will be processed and the program will continue execution at the fake return EIP, which has been set to injected code located in the stack. Thus, a single byte overflow may allow arbitrary code execution.

Heap Overflows

A common misconception by programmers is that by dynamically allocating memory (utilizing heaps) they can avoid using the stack and thus, reduce the likelihood of exploitation. While stack overflows may be the low-hanging fruit, utilizing heaps instead does not eliminate the possibility of exploitation.

The Heap

A heap is memory that has been dynamically allocated. This memory is logically separate from the memory allocated for the stack and code. Heaps are dynamically created (e.g., new, malloc) and removed (e.g., delete,free).

Heaps are generally used because the size of memory needed by the program is not known ahead of time or larger than the stack.

The memory where heaps are located generally do not contain return addresses such as the stack. Thus, without the ability to overwrite saved return addresses, redirecting execution is potentially more difficult. However, that does not mean by utilizing heaps, one is secure from buffer overflows and exploitation.

Vulnerable Code

Here is a sample program with a heap overflow. The program dynamically allocates memory for two buffers. One buffer is filled with ‘A’s. The other one is taken in from the command line. If one types too many characters on the command line an overflow will occur.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(int argc, char **argv)
{
     char *buffer = (char *) malloc(16);
     char *input = (char *) malloc(16);

     strcpy(buffer,”AAAAAAAAAAAAAAA”);

     // Use a non-bounds checked function
     strcpy(input,argv[1]);
     printf(“%s”,buffer);
}
```

With a normal amount of input, memory will appear as follows:

Address	Variable	Value
00300350	Input	BBBBBBBBBBBBBBB
00300060	?????	????????????????
00300370	Buffer	AAAAAAAAAAAAAAAA

However, if one inputs a large amount of data to overflow the heap, one can potentially overwrite the adjacent heap.

Address	Variable	Value
00300350	Input	BBBBBBBBBBBBBBB
00300360	??????	BBBBBBBBBBBBBBB
00300370	Buffer	BBBBBBBBAAAAAAAA

Exploiting the Overflow

In a stack overflow, one could overflow a buffer and change EIP. This allowed one to change EIP to point exploit code usually in the stack.

By overflowing a heap, one does not typically affect EIP. However, overflowing heaps can potentially overwrite data or modify pointers to data or functions. For example, on a locked down system one may not be able to write to C:\AUTOEXEC.BAT. However, if a program with system rights had a heap buffer overflow, instead of writing to some temporary file, someone could change the pointer to the temporary file name to instead point to the string C:\AUTOEXEC.BAT and thus, induce the program with system rights to write to C:\AUTOEXEC.BAT. This results in user-rights elevation.

In addition, heap buffer overflows also generally can result in a denial of service allowing one to crash an application.

Consider the following vulnerable program, which writes characters to C:\harmless.txt:

```
#include <stdio.h>
#include <tdlib.h>
#include <string.h>

void main(int argc, char **argv)
{
     int i=0,ch;
     FILE *f;
     static char buffer[16], *szFilename;
     szFilename = “C:\\harmless.txt”;

     ch = getchar();
     while (ch != EOF)
     {
          buffer[i] = ch;
          ch = getchar();
          i++;
     }
     f = fopen(szFilename, “w+b”);
     fputs(buffer, f);
     fclose(f);
}
```

Memory will appear like:

Address	Variable	Value
0x00300ECB	argv[1]	
...	...	...
0x00407034	*szFilename	C:\harmless.txt
...	...	...
0x00407680	Buffer	
0x00407690	szFilename	0x00407034

Notice that buffer is close to szFilename. If one can overflow buffer, one can overwrite the szFilename pointer and change it from 0x00407034 to another address. For example, changing it to 0x00300ECB which is argv[1] allows one to change the filename to any arbitrary filename passed in on the command line.

For example, if buffer is equal to: XXXXXXXXXXXXXXXX00300ECB and argv[1] is C:\AUTOEXEC.BAT, memory appears as:

Address	Variable	Value
0x00300ECB	argv[1]	C:\AUTOEXEC.BAT
...	...	...
0x00407034	*szFilename	C:\harmless.txt
...	...	...
0x00407680	Buffer	XXXXXXXXXXXXXXX
0x00407690	szFilename	0x00300ECB

Notice that szFilename has changed and now points to argv[1], which is C:\AUTOEXEC.BAT. So, while heap overflows may be more difficult to exploit than the average stack overflow, unfortunately utilizing heaps does not guarantee one is invulnerable to overflow attacks.

Function Pointers

Another second generation overflow involves function pointers. A function pointer occurs mainly when callbacks occur. If in memory, a function pointer follows a buffer, there is the possibility to overwrite the function pointer if the buffer is unchecked. Here is a simple example of such code.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int CallBack(const char *szTemp)
{
     printf("CallBack(%s)\n", szTemp);
     return 0;
}

void main(int argc, char **argv)
{
     static char buffer[16];
     static int (*funcptr)(const char *szTemp);

     funcptr = (int (*)(const char *szTemp))CallBack;
     strcpy(buffer, argv[1]); // unchecked buffer
     (int)(*funcptr)(argv[2]);
}
```

Here is what memory looks like: Address Variable Value 00401005 CallBack()

Address	Variable	Value
00401005	CallBack()	
004013B0	system()	...
...	...	...
004255D8	buffer	????????
004255DC	funcptr	00401005

So, for the exploit, one passes in the string ABCDEFGHIJKLMNOP004013B0 as argv[1] and the program will call system() instead of CallBack(). In this case, usage of a NULL (0x00) byte renders this example inert. Avoiding a NULL byte is left to the reader as to avoid exact exploit code.

Address	Variable	Value
00401005	CallBack()	
004013B0	system()	...
...	...	...
004255D8	buffer	ABCDEFGHIJKLMNOP
004255EE	funcptr	004013B0

This demonstrates another non-stack overflow that allows the attacker to run any command by spawning system() with arbitrary arguments.

Third Generation

Format String Attacks

Format string vulnerabilities occur due to sloppy coding by software engineers. A variety of C language functions allow printing of characters to files, buffers, and the screen. These functions not only place values on the screen, but can format them as well.

The following listing are common ANSI format functions:

- printf – print formatted output to the standard output stream
- wprintf – wide-character version of printf
- fprintf – print formatted data to a stream (usually a file)
- fwprintf – wide-character version of fprintf
- sprintf – write formatted data to a string
- swprintf – wide-character version of sprintf
- vprintf – write formatted output using a pointer to a list of arguments
- vwprintf – wide-character version of vprintf
- vfprintf – write formatted output to a stream using a pointer to a list of arguments
- vwfprintf – wide-character version of vfprintf

The formatting ability of these functions allows programmers to control how their output is written. For example, a program could print the same value in both decimal and hexadecimal.

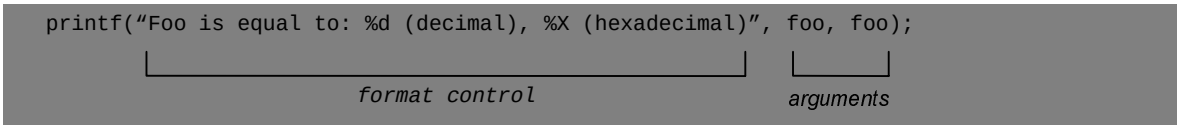
```
#include <stdio.h>
void main(void)
{
     int foo = 1234;
     printf("Foo is equal to: %d (decimal), %X (hexadecimal)", foo,
foo);
}
```

The above program would display:

```
Foo is equal to: 1234 (decimal), 4D2 (hexadecimal)
```

The percent sign '%' is an escape character signifying the next character(s) represent the form the value should be displayed. Percent-d (%d) for example, means display the value in decimal format and %X means display the value in hexadecimal with uppercase letters. These are known as format specifiers. See Appendix B, for the full list of ANSI C format specifiers.

The format function family specification requires the format control and then an optional number of arguments to be formatted.



However, sloppy programmers often do not follow this specification. The below program will print out Hello World! to the screen, but does not strictly follow the specification. The commented line demonstrates the proper way the program should be written.

```
#include <stdio.h>
void main(void)
{
     char buffer[13]="Hello World!";
     printf(buffer);    // using argument as format control!
     // printf("%s",buffer); this is the proper way
}
```

This type of sloppy programming allows one to potentially control the stack and inject and execute arbitrary code. The following program takes in one command line parameter and writes the parameter back to the screen. Notice the printf statement is used incorrectly by using the argument directly instead of a format control.

```
int vuln(char buffer[256])
{
     int nReturn=0;
     printf(buffer); // print out command line
     // printf("%s",buffer); // correct-way
     return(nReturn);
}

void main(int argc,char *argv[])
{
     char buffer[256]=""; // allocate buffer
     if (argc == 2)
     {
          strncpy(buffer,argv[1],255); // copy command line
```

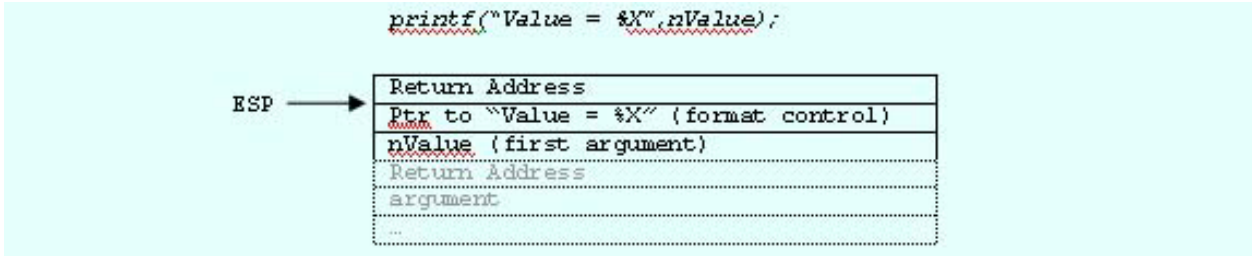
```
        }
        vuln(buffer); // pass buffer to bad function
    }
```

This program will copy the first parameter on the command line into a buffer. Then, the buffer will be passed to the vulnerable function. However, since the buffer is being used as the format control, instead of feeding in a simple string, one can attempt to feed in a format specifier.

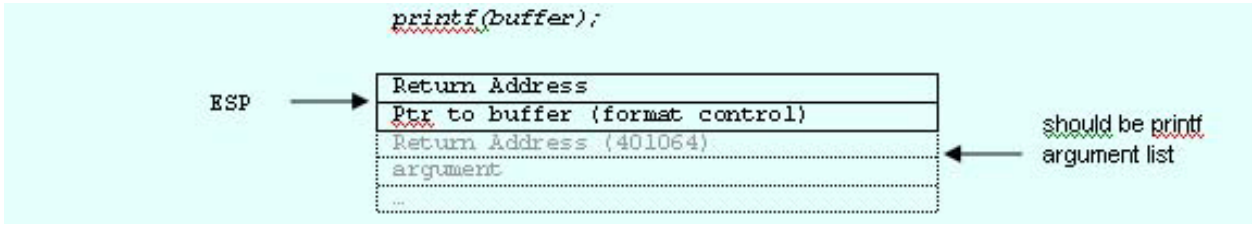
For example, running this program with the argument “%X” will return some value on the stack instead of “%X”:

```
C:\>myprog.exe %X
401064
```

The program interprets the input as the format specifier and returns the value on the stack that should be the passed-in argument. To understand why this is the case, one needs to examine the stack just after printf is called. Normally, if one uses a format control and the proper number of arguments the stack will look similar to the following:



However, by using printf incorrectly, the stack will appear differently since one does not push on the argument list and only the format control.



In this case, the program will believe the stack space after the format control is the first argument. Thus, by feeding in “%X” to our sample program, printf displays the return address of the previous function call instead of the expected missing argument. In this example we display arbitrary memory, but the key to exploitation from the point of view of a malicious hacker or virus writer would be the ability to write to memory in order to inject arbitrary code.

The format function families allow for the “%n” format specifier. This format specifier will store (write to memory) the total number of bytes written. For example,

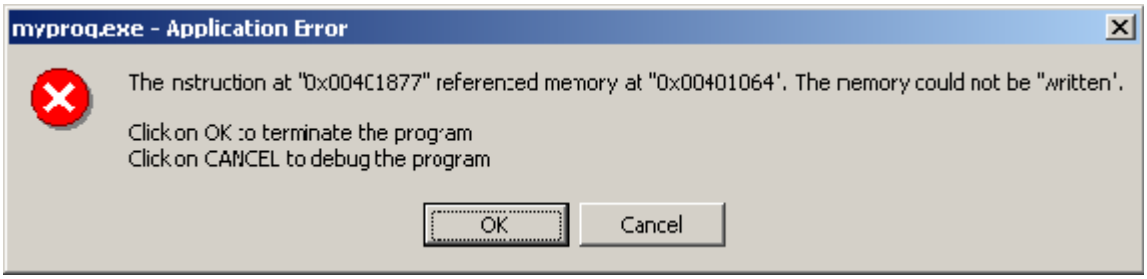
```
printf(“foobar%n”,&nBytesWritten);
```

will store ‘6’ in nBytesWritten since foobar consists of six characters. Consider the following:

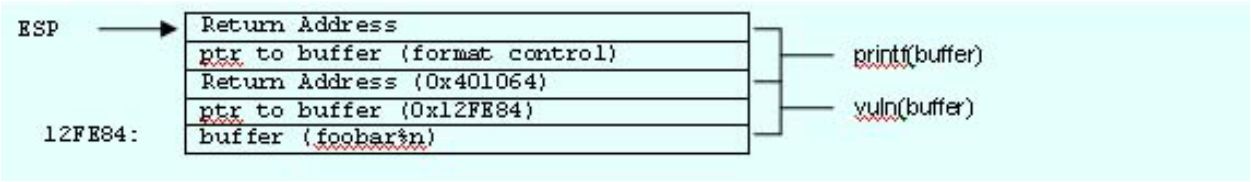
```
C:\>myprog.exe foobar%n
```

When executed instead of displaying the value on the stack after the format control, the program

will attempt to write to that location. So, instead of displaying 401064 as demonstrated above, the program will attempt to write the number 6 (the total characters in foobar) to the address 401064 and result in an application error message:

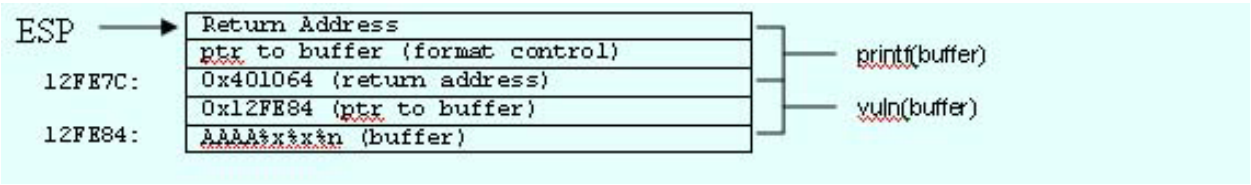


However, this demonstrates one can write to memory. With this ability, one would actually wish to overwrite a return pointer (as in buffer overflows) redirecting the execution path to injected code. Examining the stack more fully, one determines in the example program, the stack appears as follows:



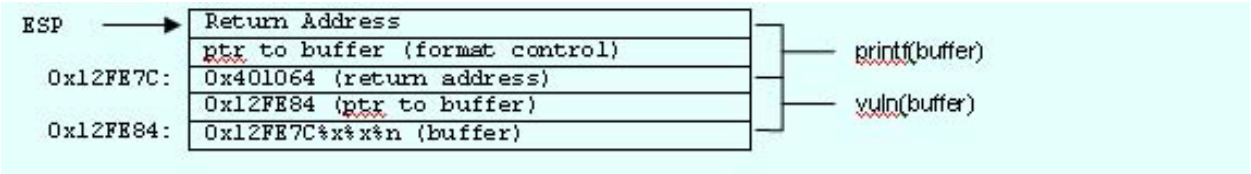
Knowing how the stack appears, consider the following exploit string:

```
C:>myprog.exe AAAA%x%x%n
```



The first format specifier, '%x', is considered the Return Address (0x401064), the next format specifier, '%x', is 0x12FE84. Finally, %n will attempt to write to the address specified by the next DWORD on the stack, which is 0x41414141 (AAAA). This allows an attacker to write to arbitrary memory addresses.

Instead of writing to address 0x41414141, an exploit would attempt to write to a memory location that contains a saved return address (such as in buffer overflows). In this example, 0x12FE7C is where a return address is stored. By overwriting the address in 0x12FE7C, one can redirect the execution path. So, instead of using a string of As in the exploit string, one would replace them with the value 0x12FE7C.

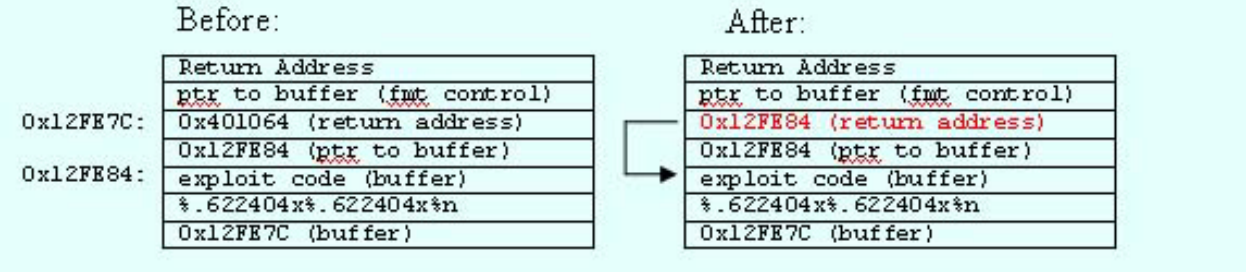


The return address should be overwritten by the address that contains the exploit code. In this case, that would be at 0x12FE84 which is where the input buffer is located. Fortunately, the

format specifiers can include how many bytes to write using the syntax %.<bytestowrite>x. Consider the following exploit string:

```
<exploitcode>%x%x%x%x%x%x%x%x%x%.622404x%.622400x%n\x7C\xFEx12
```

This will cause printf to write the value 0x12FE84 (622404+622400=0x12FE84) to 0x12FE7C if the exploit code is two bytes long. This overwrites a saved return address causing the execution path to proceed to 0x12FE84, which is where an attacker would place their exploit code.



Thus, by not following the exact specification, programmers can allow hackers to overwrite values in memory and execute arbitrary code. While the improper usage of the format function family is widespread, finding vulnerable programs is also relatively easy.

Therefore, most popular applications have been tested by security researchers for such vulnerabilities. Nevertheless, new applications are developed constantly and unfortunately developers continue to use the format functions improperly leaving them vulnerable.

Heap Management

Heap management implementations vary widely. For example, the GNU C malloc is different than the System V routine. However, malloc implementations all generally store management information within the heap area itself. This information includes such data as the size of memory blocks and is usually stored right before or after the actual data.

Thus, by overflowing heap data, one can modify values within a management information structure (or control block). Depending on the memory management functions actions (e.g., malloc and free) and specific implementation, one cause the memory management function to write arbitrary data at arbitrary memory addresses when it utilizes the overwritten control block.

Input Validation

Input validation exploits take advantage of programs that do not properly validate user supplied data. For example, a web page form that asks for an email address and other personal information should validate the email address is in the proper form and in addition does not contain special escape or reserved characters.

However, many applications such as web servers and email clients do not properly validate input; this allows hackers to inject specially crafted input that causes the application to perform in an unexpected manner.

While there are many types of input validation vulnerabilities, URL canonicalization and MIME header parsing are specifically discussed here due to their widespread usage in recent blended attacks.

URL Encoding and Canonicalization

Canonicalization is when a resource can be represented in more than one manner. For example, C:\test\foo.txt and C:\test\bar\..\foo.txt are different full pathnames that represent the same file. Canonicalization of URLs occurs in a similar manner where http://doman.tld/user/foo.gif and http://domain.tld/user/bar/../foo.gif would represent the same image file.

A URL canonicalization vulnerability results when a security decision is based on the URL and all of the URL representations are not taken into account. For example, a web server may allow access only to the /user and sub-directories by examining the URL for the string /user immediately after the domain name. For example, the URL http://domain.tld/user/../../autoexec.bat would pass the security check, but actually provide access to the root directory.

After more widespread exposure of URL canonicalization issues due to such an issue in Microsoft Internet Information Server, many applications added security checks for the double-dot string '..' in the URL. However, canonicalization attacks were still possible due to encoding.

For example, Microsoft IIS supports UTF-8 encoding such that %2F represents a forward slash '/'. UTF-8 translates US-ASCII characters (7 bits) to a single octet (8 bits) and other characters to multi-octets. Translation occurs as follows:

0-7 bits	0xxxxxxx			
8-11 bits	110xxxxx	10xxxxxx		
12-16 bits	1110xxxx	10xxxxxx	10xxxxxx	
17-21 bits	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

For example, a slash (0x2F, 0101111) would be 00101111 in UTF-8. A character with the hexadecimal representation of 0x10A (binary 100001010) has 9-bits and thus, a UTF-8 representation of 11000100 10001010.

While standard encoding did not defeat aforementioned input validation security checks, interestingly, Microsoft IIS still provides decoding, if one encodes a 7-bit character using the 8-11 bit rule format.

For example, a slash (0x2F, 101111) in 8-11 bit UTF-8 encoding would be 11000000 10101111 (hexadecimal 0xC0 0xAF). Thus, instead of using the URL, http://domain.tld/user/../../autoexec.bat one could substitute the slash with the improper UTF-8 representation, http://domain.tld/user/..%co%af../autoexec.bat. The input validation allowed this URL since it did not recognize the UTF-8 encoded forward slash, which gave access outside the web root directory. Microsoft fixed this vulnerability with security bulletin MS00-78.

In addition, Microsoft IIS performs UTF-8 decoding on two separate occasions. This allows characters to be double-encoded. For example, a backslash (0x5C) can be represented as %5C. However, one can also encode the percent-sign (0x25), itself. Thus, %5C can be encoded as %255c. On the first decoding pass, %255c is decoded to %5c and on the second decoding pass %5C is then decoded to a backslash.

Thus, a URL such as http://domain.tld/user/..%5c../autoexec.bat will not pass the input validation check, but http://domain.tld/user/..%255c../autoexec.bat would pass, allowing access outside the web root directory. Microsoft fixed this vulnerability with security bulletin MS01-26.

The inability of webservers to properly provide input validation can then lead to attacks. For

example, in IIS, one can utilize the encoding vulnerabilities to break out of the web root directory and execute cmd.exe from the Windows system directory, allowing remote execution. Win32/Nimda utilized such an attack to copy itself to the remote webserver and then execute itself.

MIME Header Parsing

When Internet Explorer parses a file, the file can contain embedded MIME encoded files. Handling of these files occurs by examining a header, which defines the MIME type. Using a lookup table, these MIME types are associated with a local application. For example, the MIME type audio/basic is generally associated with Windows Media Player. Thus, MIME encoded files designated as audio/basic will be passed to Windows Media Player.

MIME types are defined by a Content-Type header. In addition to the associated application, each type has a variety of associated settings including the icon, whether to show the extension, and whether to automatically pass the file to the associated application when the file is being downloaded.

When receiving an HTML email with Microsoft Outlook and some other email clients, code within Internet Explorer actually renders the e-mail. If the e-mail contains a MIME embedded file, Internet Explorer would parse the email and attempt to handle embedded MIME file. Vulnerable versions of Internet Explorer would check whether the application should automatically be opened (passed to the associated application without prompting) by examining the Content-Type header. For example, audio/x-wav files are automatically passed to Windows Media Player for playing.

However, a bug exists in vulnerable versions of Internet Explorer where files are passed to the incorrect application. For example a MIME header may appear as:

```
Content-Type: audio/x-wav;
     name=”foobar.exe”
Content-Transfer-Encoding: base64
Content-ID: <CID>
```

In this case, Internet Explorer determines the file should be automatically passed to the associated application (no prompting) since the content type is audio/x-wav. However, when determining what the associated application is, instead of utilizing the Content-Type header (and the file header itself) Internet Explorer incorrectly relies on a default association which will be made according to the extension. In this case, the extension is .EXE and thus, is passed to the operating system for execution instead of passing the audio file to an associated application to be played.

Such a bug allows for the automatic execution of arbitrary code. Several Win32 mass mailers send themselves via an email with a MIME encoded malicious executable with a malformed header, and the executable will silently execute unbeknownst to the user. This occurs whenever Internet Explorer parses the mail and thus can happen when simply reading or previewing email. Thus, email worms can spread themselves without any user actually executing or detaching a file.

Any properly associated MIME file type that has not set the “Confirm open after download” flag can be utilized for this exploit. Thus, a definitive list is unavailable considering developers can

register their own MIME types.

Such an exploit was utilized by both Win32/Badtrans and Win32/Klez allowing them to execute themselves upon reading or previewing an infected email.

Application Rights Verification

While improper input validation may give applications increased access such as with URL canonicalization, other models simply give applications increased rights due to improper designation of code as safe. Such a design is employed by ActiveX and as a result numerous blended attacks have also used ActiveX control rights verification exploits.

Safe for Scripting ActiveX Controls

Generally by design, ActiveX controls are scriptable. They expose a set of methods and properties that can potentially be invoked in an unforeseen and malicious manner often via Internet Explorer.

The security framework for ActiveX controls requires the developer to determine if their ActiveX control could potentially be used in a malicious manner. If a developer determines their control is safe, they may mark the control 'safe for scripting'.

Microsoft notes that ActiveX controls that have any of the following characteristics must not be marked safe for scripting.

- Accessing information about the local computer or user.
- Exposing private information on the local computer or network.
- Modifying or destroying information on the local computer or network.
- Faulting of the control and potentially crashing the browser.
- Consuming excessive time or resources such as memory.
- Executing potentially damaging system calls, including executing files.
- Using the control in a deceptive manner and causing unexpected results.

However, despite these simple guidelines some ActiveX controls with these characteristics have been marked safe for scripting and thus, been used maliciously.

For example, VBS/Bubbleboy used the Scriptlet.Typelib ActiveX control to write out a file to the Windows Startup directory. The Scriplet.Typelib contained properties to define the path and contents of the file. Because this ActiveX control was incorrectly marked safe for scripting, one could invoke a method to write a local file via a remote webpage or HTML email without triggering any ActiveX warning dialog.

ActiveX controls that have been marked safe for scripting can be easily determined by examining the registry. If the safe for scripting CLSID key exists under the Implemented Categories key for the ActiveX control, the ActiveX control is marked safe for scripting.

For example, the Scriptlet.Typelib control has a class ID of {06290BD5-48AA-11D2-8432-006008C3FBFC} and the safe for scripting CLSID is {7DD95801-9882-11CF-9FA0-

00AA006C42C4}. In the registry an unpatched system would contain the key:

```
HKCR/CLSID/{06290BD5-48AA-11D2-8432-006008C3FBFC}/Implemented Cat-
egories/{7DD95801-9882-11CF-9FA0-00AA006C42C4}
```

allowing any remote webpage or incoming HTML email to create malicious files on the local system. Clearly, leaving such security decisions to the developer is far from foolproof.

System Modification

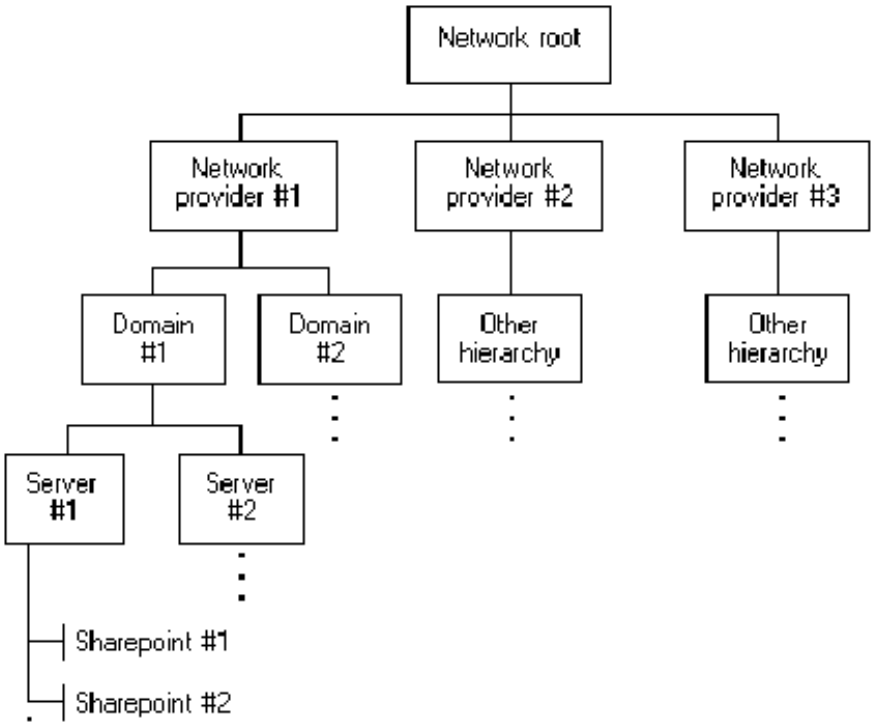
Once malicious software gains access to the system, the system is often modified to disable application or user rights verification. Such modifications can be as simple as eliminating a root password or modifying the kernel allowing user rights elevation or previously unauthorized access.

For example, CodeRed creates virtual webroots allowing general access to the compromised webserver and Win32/Bolzano patches the kernel disabling user rights verification on Windows NT systems.

Network Enumeration

Several 32-bit computer viruses enumerate the Windows networks using standard Win32 browsing APIs such as WNetOpenEnum(), WNetEnumResourceA() of MPR.DLL. The first use of this attack appeared in the Win32/ExploreZip.

The following figure shows the structure of a typical Windows network:



Resources that do not contain other resources are called objects. In the preceding figure, Sharepoint #1 and Sharepoint #2 are objects. A sharepoint is an object that is accessible across the network. Examples of sharepoints include printers and shared directories.

Win32/Funlove virus was the first file infector to infect files on network shares using network enumeration. Win32/FunLove caused major headache by infecting large corporate networks world wide. This is because the network aware nature of the virus. Many people often share directories without any security restrictions in place. Most people share more directories (such as a drive C:) than they need to and often without any passwords. This enhances the effectiveness of network aware viruses.

Some viruses such as Win32/HLLW.Bymer use the form \\nnn.nnn.nnn.nnn\c\windows\ (where nnn-s describe an IP address) to access the C: drive of any remote systems that have a Windows folder on it. Such an attack can be particularly painful for many home users running a home PC that is typically not behind a firewall. Windows makes the sharing of network resources very simple. However, extra attention is needed on the user's part to use strong passwords, to limit the access to the only necessary resources, and even further, to use other means such as personal firewall software.

CURRENT AND PREVIOUS THREATS

The follow section describes a variety of blended attacks in more detail. These include the infamous Morris worm and CodeRed, which use a buffer overflow; threats such as Win32/Badtrans and Win32/Nimda, which use input validation exploits; and Win32/Bolzano and VBS/Bubbleboy, which use application or user rights exploits.

Morris Worm

The Morris worm implemented a buffer overflow attack against the fingerd program. This program runs as a system background process and satisfies requests based on the finger protocol on the finger port (79 decimal). The problem in fingerd was related to its use of the gets() library function. The gets() function contained an exploitable vulnerability (there were a couple of other functions on BSD systems that had a similar problem).

Because fingerd declared a 512 byte buffer for gets() call without any bounds checking, it was possible to exploit this and send a larger string to fingerd. The Morris worm crafted a 536 byte "string" containing assembly code (so called shell code) on the stack of the remote system to execute a new shell via a modified return address.

First the 536 byte buffer was initialized with zeros, filled with data and sent over to the machine to be attacked, followed by an "\n" to indicate the end of the string for gets().

This attack worked only against VAX (Virtual Address eXtension) systems that were designed and built from 1977 through to retirement of the system in 1999/2000.

VAX is a 32-bit CISC architecture. The system was a successor to PDP-11. VAX uses 16 registers that are numbered from r0-r15 but several of these registers are special and map to SP (Stack Pointer), AP (Argument Pointer) and so on.

The actual instructions responsible for the attack were on the stack as follows, but inside the originally reserved buffer at position 400 decimal.

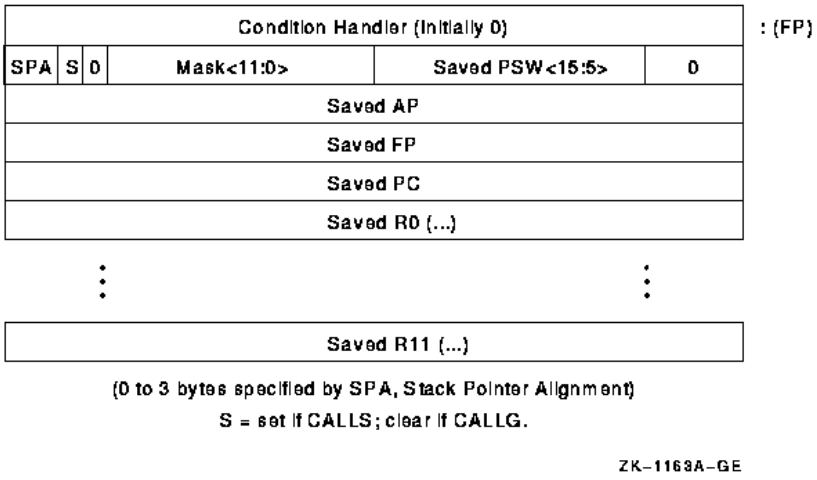
```
DD8F2F736800    pushl  $68732f      ; '/sh\0'
DD8F2F62696E    pushl  $6e69622f    ; '/bin'
D05E5A          movl   sp, r10      ; save pointer to command
DD00            pushl  $0           ; third parameter
DD00            pushl  $0           ; second parameter
DD5A            pushl  r10          ; push address of '/bin/sh\0'
DD03            pushl  $3           ; number of arguments for chmk
D05E5C          movl   sp, ap       ; Argument Pointer register = stack pointer
BC3B            chmk   $3b          ; change-mode-to-kernel
```

The above code is an execve("/bin/sh", 0, 0) system call. [spaff88].

Bytes 0 through 399 of the attack buffer were filled with the 01 opcode (NOP). An additional set of longword-s were also changed beyond the original buffer size, which in turn smashed the stack with a new return address that the author hoped would point into the buffer and eventually hit the shell code within it. When the attack worked the new shell took over the process and the worm could successfully send new commands to the system via the open network connection.

The worm modified the original return address of main() on the stack of fingerd. When main() (or any function is called) on a VAX machine with a calls or callg instruction, a call frame is generated on the stack. Since the first local variable of fingerd was the actual buffer in question, main's call frame was placed next to the buffer. Overflow of the buffer causes the call frame to be changed.

The Morris worm modified this call frame (rewriting 6 entries in it) and specified the return address (in the position of the saved PC - Program Counter) that would hopefully point into its own crafted buffer. (The PC is the equivalent of the EIP on the Intel CPU). The NOPs in the worm's attack buffer increase the chance that control will eventually arrive at the shell code. The worm's code specifies the call as a calls instruction by setting the S bit of the Mask field. The following picture shows the call frame layout on a VAX:



Eventually a ret instruction will access the altered call frame (which is likely to be very similar to its original content except for the Saved PC), pick up the new PC and return to a location that which will hit the shell code of the worm.

Shell code is always crafted to be as short as possible. In the case of the Morris worm, the shell code is 28 bytes. Shell code often needs to fit in small buffers to exploit the maximum set of vulnerable applications. In the case of the finger daemon, the actual available buffer size was 512 bytes, but obviously some other applications may not have "that much" space for the shell code.

Linux/ADM

In 1998, around the 10th anniversary of the Morris worm a group of hackers created a new Linux worm, called Linux/ADM, which quickly spread in-the-wild. The worm utilized a buffer overflow technique to attack BIND (Berkeley Internet Name Domain) servers.

BIND is a service listening on the NAMESERVER_PORT (53 decimal). The worm attacked the server with a malformed IQUERY (Inverse Query) by specifying a long request body (packet) for the query. Certain BIND versions have had a couple of similar buffer overflow vulnerabilities in several places, but the bug in question was in the ns_req.c module. A function called req_iquery() is called to satisfy any incoming IQUERY request.

The packet size for the query is crafted so that it will be long enough to hit a return address. Thus the function does not return but hopefully executes the attack buffer which is filled with NOP instructions and shell code to call execve on Intel based Linux systems (Function=0x0b, INT 80h). Thus Linux/ADM's attack is very similar to that of the Morris worm. Linux/ADM also uses a shell code based attack and the important difference is that Linux/ADM recompiles itself entirely while Morris worm did not have more than a short boot code that was compiled to target platforms.

Linux/ADM worm consists of several C files as well as other script files in a TAR file. It compiled these modules as test, Hnamed, gimmeRAND, scanco and remotecmd respectively.

Once the worm is executed it looks for new hosts to infect at a random IP address generated using gimmeRAND, and then by checking the vulnerable systems using scanco.

When a vulnerable system is detected, the module Hnamed is used with parameters passed to it. The parameters to Hnamed specify the machine to be attacked as well as the attack shell string, which is a piped /bin/sh command chain.

The worm can snatch its source from the attacker machine and restart the compilation process on the new host. Linux/ADM makes sure to install an additional remote command prompt.

This remote command prompt gets particularly interesting since a former white hat security person, Max Butler created a counter attack in the form of a worm to install security patches on Linux systems to stop the Linux/ADM worm and similar attacks. Butler modified the worm to install patches, but it appears that he forgot to remove the remote prompt which opened the systems to outside attacks in the first place.

Max Butler was sentenced to 18 months in prison for launching the worm that crawled through hundreds of military and defence contractor computers over a few days in 1998. [Security Focus]

CodeRed

The CodeRed worm was released to the wild in July of 2001. The worm replicated to thousands of systems in a matter of a few hours. It was estimated that well over 300.000 machines were infected by the worm within 24 hours. All of these machines were Windows 2000 systems running vulnerable versions of Microsoft IIS.

Interestingly enough the worm did not need to create a file on the remote system in order to infect it, but existed only in memory of the target system. The worm accomplished this by getting into the process context of the Microsoft IIS with an extremely well crafted attack via port 80 (web service) of the target system.

First the IIS web server receives GET /default.ida? followed by 224 characters, URL encoding for 22 Unicode characters (44 bytes), an invalid Unicode encoding of %u00=a, HTTP 1.0, headers, and a request body.

For the initial CodeRed worm, the 224 characters are N, but there were other implementations that used other filler bytes such as X. In all cases, the URL encoded characters are the same (they look like %uXXXX, where X is a hex digit). The request body is different for each of the variants known.

IIS keeps the body of the request in a heap buffer (probably the one it read it into after processing the Content-length header indicating the size to follow). Note that a GET request is not allowed to have a request body, but IIS dutifully reads it according to the header's instructions anyway. For this very reason, an appropriately configured firewall would cut the request body (and the worm), and CodeRed could not infect the attacked system.

Buffer Overflow Details

While processing the 224 characters in the GET request, functions in IDQ.DLL overwrites the stack at least twice (Figure 1) once when expanding all characters to Unicode, and again when decoding the URL escaped characters. However, the overwrite that results in the transfer of control to the worm body happens when IDQ.DLL calls DecodeURLEscapes() in QUERY.DLL.

The caller is supposed to specify a length in wide chars, but instead specifies a number of bytes. As a result, DecodeURLEscapes() thinks it has twice as much room as it actually has, so it ends up overwriting the stack. Some of the decoded Unicode characters specified in URL encoding end up overwriting a frame-based exception block. Even after the stack has been overwritten, processing continues until a routine is called in MSVCRT.DLL. This routine notices that something is wrong and throws an exception.

Exceptions are thrown by calling the KERNEL32.DLL routine RaiseException(). RaiseException() ends up transferring control to KiUserExceptionDispatcher() in NTDLL.DLL. When KiUserExceptionDispatcher() is invoked, EBX is pointing to the exception frame that was overwritten.

The exception frame is composed of 4 DWORDs, the second of which is the address of the exception handler for the represented frame. The URL encoding whose expansion overwrote this frame starts with the third occurrence of %u9090 in the URL encoding, and is:

%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3

This decodes as the four DWORDs: 0x68589090, 0x7801CBD3, 0x90909090, and 0x00C38190.

The address of the exception handler is set to 0x7801CBD3 (2nd DWORD), and KiUserExceptionDispatcher() calls there with EBX pointing at the first DWORD via CALL ECX.

Address 0x7801CBD3 in IIS's address space is within the memory image for the C runtime DLL MSVCRT.DLL. MSVCRT.DLL is loaded to a fixed address. At this address in MSVCRT.DLL is the instruction CALL EBX. When KiUserExceptionDispatcher() invokes the exception handler, it calls to the CALL EBX, which in turn transfers control to the first byte of the overwritten exception block. When interpreted as code, these instructions find and then transfer control to the main worm code, which is in a request buffer in the heap.

The author of this exploit needed the decoded Unicode bytes to function both as the frame-based exception block containing a pointer to the "exception handler" at 0x7801CBD3, and as runable code. The first DWORD of the exception block is filled with 4 bytes of instructions arranged so that they are harmless, but also place the 0x7801CBD3 at the second DWORD boundary of the exception block. The first two DWORDs (0x68589090, 0x7801CBD3) disassemble into the instructions: nop, nop, pop eax, push 7801CBD3h which accomplish this task easily.

Having gained execution control on the stack (and avoiding a crash while running the "exception block"), the code finds and executes the main worm code.

This code knows that there is a pointer (call it pHeapInfo) on the stack 0x300 bytes from EBX's current value. At pHeapInfo+0x78, there is a pointer (call it pRequestBuff) to a heap buffer containing the GET request's body, which contains the main worm code. With these two key pieces of information, the code transfers control to the worm body in the heap buffer. The worm code does its work, but never returns - the thread has been hijacked (along with the request buffer owned by the thread).

Exception Frame Vulnerability

This technique of usurping exception handling is complicated (and crafting it must have been difficult). The brief period between the eEye description of the original exploit and the appearance of the first CodeRed worm leads us to believe that this technique is somewhat generic. Perhaps the exception handling technique has been known to a few buffer overflow enthusiasts for some time, and this particular overflow was a perfect opportunity to use it.

Having exception frames on the stack makes them extremely vulnerable to overflows. In other words the CodeRed worm relied on the Windows exception frame vulnerability to carry out a quick buffer overflow attack against IIS systems.

Buffer Overflow Usage in Computer Viruses

Based on the set of previous examples, it is easy to see that computer worms typically attack server systems; usually service processes and daemon programs that are waiting to handle incoming requests by listening on various TCP/IP ports. Any such communication service could

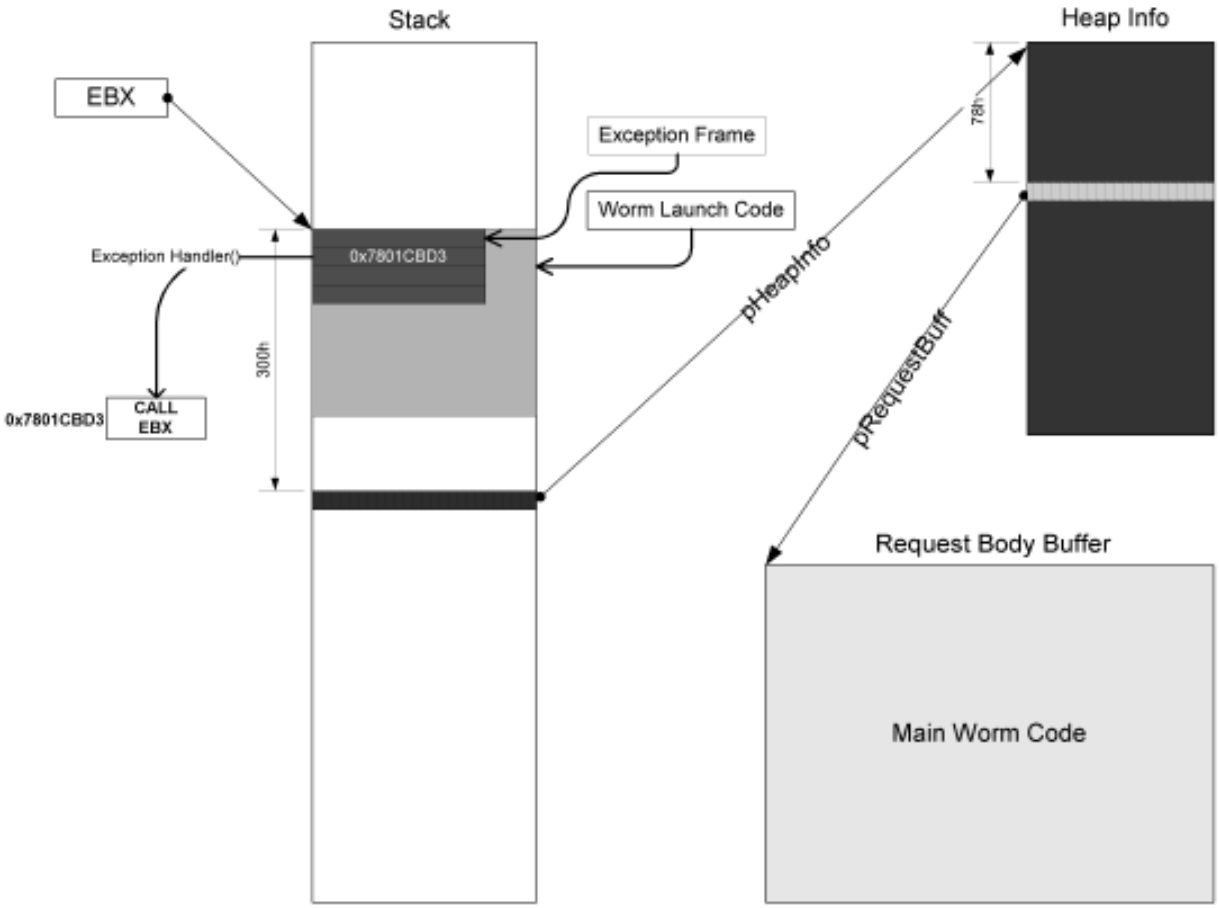


Figure 1. Stack, Heap and Frame Layout of a CodeRed attack.

potentially contain flaws, as did the fingerd (in the case of Morris worm), the BIND (in the case of the AMD worm) and Microsoft IIS (in the case of the CodeRed worm).

We should note that best practices prescribe the removal of all unneeded services from the system. Any such services should be removed to make the execution environment safer and to reduce the opportunities for malicious hackers and virus attacks to break into internal networks. In the case of CodeRed, for example, the vulnerable IDQ.DLL was only loaded to support the very rarely used indexing service. Had this service been disabled on systems that didn't need it (which was likely most systems), CodeRed would not have been so successful.

Win32/Badtrans.B@mm

Description

Win32/Badtrans.B@mm was discovered on November 24, 2001. Win32/Badtrans.B@mm is a worm that emails itself out using different file names and steals data such as passwords from the infected computer.

MIME Exploit

The worm used a MIME header exploit, which allowed the worm to execute when previewing or

reading an infected email message. Microsoft Outlook (Express) or potentially any mail client, which utilizes Internet Explorer for rendering of HTML mail were affected. Users did not have to detach or execute the attachment in order to become infected — simply reading or previewing the message caused the attachment to be executed.

The MIME exploit utilized is described in the previous section MIME Header Parsing.

The MIME header vulnerability was corrected by Microsoft in security bulletin MS01-20 in March, 2001 yet the worm was still effective in November, eight months later. If a vulnerable system was patched, the risk would have been significantly lower.

Unfortunately, many machines today are still vulnerable to this exploit. All variants of W32.Klez (first discovered in October of 2001) uses the same exploit and topped the virus infection charts in May, 2002 more than a year after a patch has been available.

Win32/Nimda.A@mm

Description

Win32/Nimda.A@mm is a worm that utilizes multiple methods to spread itself. The worm sends itself out by email, searches for open network shares, attempts to copy itself to unpatched or already vulnerable Microsoft IIS web servers, and is a virus infecting both local files and files on remote network shares.

The worm uses the Web Server Folder Traversal exploit to infect IIS web servers and a MIME header exploit allowing it to be executed just by reading or previewing an infected email.

Win32/Nimda.A@mm began infecting around 12:00 GMT on September 18, 2001. Within the first twelve hours, Win32/Nimda.A@mm infected at least 450,000 unique hosts with 160,000 hosts simultaneously infected at the peak rate of infection. The ability to infect IIS web servers via an input validation exploit and auto-execute upon reading or previewing email played a large role in Win32/Nimda.A@mm's ability to spread so rapidly.

IIS Exploit Explained

Win32/Nimda.A@mm exploits the Microsoft Internet Information Server MS01-026 vulnerability or previously compromised installations to copy itself to the server via TFTP and then executes itself remotely.

First, Win32/Nimda.A@mm probes the server by issuing a dir command using an exposed cmd.exe from a previously compromised machine or winnt\system32\cmd.exe by using multiple variations of a URL canonicalization and encoding vulnerability as described earlier.

If the server responds with a '200' success message, Win32/Nimda.A@mm records that the server is vulnerable and begins its uploading routine. If the server does not respond with a success message another malformed URL is tried. In total, Win32/Nimda.A@mm attempts thirteen different URL encoding attacks and four specific URLs for servers that have been previously compromised.

The worm uploads itself by executing tftp.exe on the remote server again using a malformed URL to break out of the web root. The worm uses the TFTP client to connect back to the infect-

ing server, downloading a copy of itself as the filename httpodbc.dll.

Once copied to the victim web server, Win32/Nimda.A@mm then executes itself by simply performing a HTTP GET request of itself on the remote web server (e.g., GET /directory/<exploit string>/<filename>.dll).

Win32/Bolzano

Description

Win32/Bolzano is a direct action appending virus that infects PE files. While the virus replication routine is simple, the modification of the Windows NT kernel to turn off user rights verification was novel.

Kernel Modification

Viruses such as Win32/Bolzano (and later Win32/FunLove with the same trick) modify kernel files on the machine to give a virus an advantage.

Win32/Bolzano as well as Win32/FunLove needs administrative rights on a Windows NT server or Windows NT workstation during the initial infiltration. Therefore it is not a major security risk, but still a potential threat. Viruses can always wait until the Administrator or someone with equivalent rights logs on.

In such a case, Win32/Bolzano has the chance to patch ntoskrnl.exe, the Windows NT kernel, located in the WINNT\SYSTEM32 directory. The virus modifies only 2 bytes in a kernel API called SeAccessCheck() that is part of ntoskrnl.exe. In this way, Win32/Bolzano is able to give full access to all users for every file regardless of its protection whenever the machine is booted with the modified kernel. This means that a Guest -having the lowest possible rights on the system- will be able to read and modify all files including files that are normally accessible only by the Administrator.

This is a potential problem since the virus can spread everywhere it wants to regardless of the actual access restrictions on the particular machine. Furthermore after the attack, no data can be considered protected from any user. This happens because the modified SeAccessCheck() API is forced to return 1 always, instead of 0 or 1. 1 means that the particular user has the necessary rights to access a particular file or directory placed on an NTFS partition while 0 means the user has no access. SeAccessCheck() is called each time when the file access rights should be checked.

Unfortunately, the consistency of ntoskrnl.exe is checked in only one place. The loader, ntldr, is supposed to check ntoskrnl.exe when loading it into physical memory during machine boot-up. If the kernel gets corrupted ntldr is supposed to stop loading ntoskrnl.exe and display an error message even before a “blue screen” appears. In order to avoid this particular problem, Win32/Bolzano also patches the ntldr so that no error message will be displayed and Windows NT will boot just fine even if its checksum does not match with the original.

Since no code checks the consistency of ntldr itself, the patched kernel will be loaded without notification to the user. Since ntldr is a hidden, system, read-only file Bolzano changes the attributes of it to “archive” before it tries to patch it. (The modification of ntoskrnl.exe and other

system executables were somewhat resolved by the System File Checker feature of Windows 2000/XP systems. Unfortunately, malicious code can easily disable this feature. Furthermore, the primary function of SFC is not virus protection, but a solution to the so called “DLL hell” problem.)

On certain other systems such as Linux, the kernel source files are commonly directly available on the systems (even on those not used for development). Therefore any virus or other malware can easily alter the source of the system to gain root privileges once the kernel is recompiled (assuming the virus has write access to the source at some point).

VBS/Bubbleboy

Description

VBS/BubbleBoy is a Visual Basic Script worm that works under English and Spanish versions Windows 98 and Windows 2000. The worm emails itself out to every email address in the Microsoft Outlook address book. The worm utilizes an ActiveX control safe for scripting exploit to execute itself without the user detaching or executing the attachment. This exploit was first used by JS/Kak.

ActiveX Safe for Scripting Exploit

VBS/Bubbleboy uses the Scriptlet.Typelib control to create a malicious HTA (HTML Application) file in the Windows Startup directory. Thus, the next time the computer is restarted, the HTA file executes the replication routine. By exploiting a safe for scripting vulnerability, the file is created when one simply reads or previews email using a vulnerable version of Internet Explorer 5.

Scriptlet.Typelib allows one to dynamically generate a type library for a Windows Script Component. Such a type library would normally contain information about its interfaces and members. Type libraries are useful for statement completion in Visual Basic and also for displaying exposed properties and methods in the Object Browser.

Thus, Scriptlet.Typelib provides a Path property specifying where the type library will be written and a Doc property, which is normally used for a string with the type library information. Finally, the Write method creates the type library file.

Unfortunately, the Scriptlet.Typelib control was marked safe for scripting. This allowed remote scripts (scripts files in the Internet Zone such as HTML email or remote webpages) to utilize the methods and properties of Scriptlet.Typelib and thus, write out a file (which normally would be a .tlb type library file) to the local system with arbitrary content defined by the Doc property. The following code snippet demonstrates how to use Scriptlet.Typelib to write out a file.

```
Set oTL = CreateObject(“Scriptlet.Typelib”)
oTL.Path=”C:\file.txt”      ‘ .tlb path/filename
oTL.Doc=”Hello World!”      ‘ .tlb file contents
oTL.Write          ‘ write out file to local disk
```

Microsoft provided a patch to fix this problem in security bulletin MS99-032 on August 31, 1999.

Win32/Blebla

Description

This worm uses variable subject lines and has two attachments named Myjuliet.chm and Myromeo.exe. Once a message is read or previewed in vulnerable installations of Microsoft Outlook (Express), the two attachments are automatically saved and launched. When launched, this worm attempts to send itself out to all names in the Microsoft Outlook address book using one of several Internet mail servers located in Poland.

ActiveX and Cache Bypass Exploit Explained

Win32/Blebla uses a combination of an ActiveX control, which is marked safe for scripting and another vulnerability that allows saving files locally to known locations.

The worm uses the showHelp method of the HHCtrl (HTML Help) ActiveX control. This method allows one to display (and thus, execute) CHM (Compiled HTML) files via scripting. Originally, showHelp contained a vulnerability that allowed one to provide a full UNC path as the filename to open. This allowed one to locally launch remote CHM files. Microsoft fixed this vulnerability in security bulletin MS00-37.

However, while the ability to launch remote CHM files was corrected, the control was still marked safe for scripting. Furthermore, the control could still launch local CHM files. Thus, one can launch local CHM files via remote scripts (such as HTML email and remote webpages). This wasn't viewed a vulnerability since the malicious CHM file would already need to exist on the local system.

Thus, the worm combines using the local execution ability of showHelp along with a cache bypass vulnerability. Normally, when receiving HTML e-mail, inline files such as images (<IMG SRC=inline.gif>) are automatically saved to the Temporary Internet Files folder. By design, this cache directory is treated as the Internet Zone and is off limits to remote scripts. Thus, even if Win32/Blebla was able to save a local CHM file to this directory, showHelp would not have permission to load files from that directory.

Using a cache bypass vulnerability in Microsoft Outlook, the worm was able to save a CHM file to the known location of C:\Windows\Temp, bypassing the requirement to utilize the restricted Temporary Internet Files directory.

Once the malicious file is saved to C:\Windows\Temp, the worm then launches the malicious CHM file using the showHelp method.

This demonstrates a dual-vulnerability attack. The showHelp method by itself is not vulnerable. However, by combining the showHelp method with another vulnerability, one is able to remotely execute arbitrary code on the local system.

Microsoft fixed the cache bypass vulnerability in security bulletin MS00-046.

If the HHCtrl ActiveX control is simply marked not safe for scripting, then this type of attack can not take place whether or not another vulnerability exists. Unfortunately, today, the HHCtrl is still marked safe for scripting and thus, can be used remotely to launch local files.

CURRENT SECURITY

Many of these blended threats are effective today because most current security products can not prevent the threats. Furthermore, only some products can detect the threats once they have arrived on the system or simply alert an attack has taken place.

For example, traditional anti-virus products do not scan Windows memory. This means blended threats such as CodeRed, which reside solely in memory might remain undetected. In addition, most intrusion detection products merely alert rather than block when a signature matches. Thus, while an administrator may receive an alert, the worm has already infected the machine. Some modern intrusion detection systems employ gated technology, which means threats are actually blocked when detected.

One of our own solutions included the use of a memory scanner to detect the worm in memory. The program also determined the need for the patch. Unfortunately, CodeRed's code can appear on the heap of IIS processes that are not vulnerable to the attack. Therefore a scanning solution had to prove the active existence of the worm on the machine. (Appendix A. shows some detailed of thread information on a system infected by two different variants of CodeRed).

Such on-demand memory scanning solutions are still unable to prevent the worm from entering the system. Active protection for such an attack needs to be identified elsewhere.

Firewall software can be configured to prevent certain types attacks from happening in the first place. Symantec Raptor Firewall stopped CodeRed without requiring new rules because it enforces the rule that GET requests are not allowed to have a body. Since CodeRed's main code was sent as the body of a malformed GET request, Raptor dropped the illegal body and CodeRed's main code could not reach its intended target.

However, we know that firewall software by itself will never prevent virus attacks entirely. In addition, firewalls will not prevent against all new kinds of threats and will require expertise in dynamic maintenance of firewall configuration. Using a firewall to mitigate the risk is not limited to the standard enterprise firewall, but also includes personal firewall. Personal firewall can perform similar tasks, but geared for a single workstation rather than a whole network. The combination of personal and enterprise firewall can be extremely powerful to deal with not only incoming but also outbound attacks. Half of the damage is done when the attack enters the internal network; the other half occurs when it leaves the internal network. The secondary damage can often be more costly than the primary damage.

Host based IDS solutions also need to be considered. Such software exists for a various operating systems, but we need to realize that providing protection on all of the platforms in a diversified internal network can be very complex. Nevertheless, even if attacks are stopped only on Intel and Windows platform host systems, a large part of possible virus problems can be prevented effectively. IDS needs to cover a wild variety of operating systems managed from a single central location.

Other quick "solutions" have included a counter attack (causing a buffer overflow to crash the attacker process) on the infected nodes. Some individuals even implemented worms to attack vulnerable machines and execute the appropriate patch. Obviously these solutions have some interesting ethical context.

Unfortunately, once a machine has been compromised, full system reinstallation should be considered based on the location of the system on the network.

COMBATING BLENDED THREATS IN THE FUTURE

Combating future threats requires the interaction of multiple existing technologies including host and network based IDS and anti-virus software and future scanning technologies. Combining the technologies will provide for more comprehensive coverage and clearly protective measures must be deployed at all levels of the enterprise.

Anti-virus scanners need to implement memory scanning. Otherwise, threats that are injected directly into memory will be undetected. Currently, anti-virus products do have DOS memory scanning, but not the equivalent for Windows or other operating systems. Such memory scanning functionality is already in development by many vendors and will most likely be implemented on-demand.

Preventing threats that inject themselves into memory via the network requires scanning incoming network data via IDS. Current anti-virus scanners generally detect threats only when they have been saved to disk. For example, when Win32/Funlove infects via network shares, anti-virus programs can only detect the infections once they have taken place. Worms that copy themselves over network shares similarly will only be detected once they have actually finished copying themselves. This poses a variety of problems.

Occasionally, Win32/Funlove may accidentally corrupt files when infecting via network shares. Anti-virus products can't prevent this from happening, but can only detect the virus (and corrupted file) after the infection (and corruption) has already taken place. Thus, the customer is effectively left unprotected and anti-virus becomes only an alerting mechanism rather than a prevention mechanism. Furthermore, Win32/Funlove repeatedly infects open network shares and thus, a loop occurs where anti-virus products disinfect files and then Win32/Funlove infects them again, over and over again.

Thus, future technologies might require anti-virus scanners to scan incoming network data similarly to network IDS, which could reduce this problem considerably. (However network performance problems might arise.) For example, an anti-virus scanner may detect Win32/Funlove transferring over its code via Microsoft Networking and block the stream of data preventing files from becoming infected on the open share. Network scanning technology could also detect CodeRed transferring itself over port 80 and block it from injecting itself into memory. Furthermore, networked based IDS can help to find the source of an attack that is crucial to remove infected host from the internal network as quickly as possible.

In addition, anti-virus solutions need to incorporate modern behavior blocking technology. A set of new patent pending features will be available in new versions of Symantec Norton AntiVirus to not only prevent script treats but mass mailing Win32 worms also. Such technology will need to be extended further to prevent infections of network shares and local systems. Although host based behavior blocking technology is not new, it needs to evolve as new attacks appear.

These solutions will not be deployed in a stand-alone fashion, but integrated into a network of products that communicate and correlate data. For example, in the future a local anti-virus-like

scanner may detect a suspicious file on the system, which appears to have a malicious e-mailing routine. However, this alone isn't enough to trigger on the file. A behavior analysis component then notices this file has actually made a connection to a mail server. Next, a network traffic scanner tracks the file being sent to another machine on the network. While still suspicious, the actions may be legitimate. A local file scanner on the victim machine then verifies the file is the same as the original one on the original host machine adding even more evidence that the threat is malicious.

The same components continue to track the actions from one machine to another. Other anomalous behavior is also tracked such as unusual network traffic or system modifications. Once, a pattern is developed where the file is seen to be automatically sent from one machine to another and to another, an automated signature or filtering is created and the mail server is auto-updated to block the transfer of the file. Furthermore, all the scanning products are updated to block at their scanning point in the network. The file and logs can then be automatically sent to the vendor for analysis and confirmation. The correlation of data can help to reduce the number of events.

The aforementioned smart scanning scenario is in development already today. Cooperation of multiple scanning components allows them to build a bigger picture of what is occurring on the network. By combining this data, blended threats may be more easily detected and in many cases prevented.

SUMMARY

While blended threats have existed for more than ten years, their reappearance today is of greater concern. In the past, the usage of networking and the Internet was limited to governments and university research. Today, Internet usage is main stream and being utilized in many aspects of business.

Blended threats can spread faster and further than classic virus threats and unfortunately, effective solutions are still only on the horizon. The best line of protection still remains vigilance in applying critical patches. Host and network based vulnerability assessment tools can help to identify outdated systems with security holes inside the internal networks quicker and thus security patches can be delivered faster. Furthermore vulnerability assessment tools can help to ensure that passwords are set up in accordance to the corporate requirements and they can identify unneeded and insecure system services that need to be uninstalled. Enterprise and personal firewall software can help to fight with inbound and outbound attacks reliably. Preventing technologies needs to be installed on the workstations, servers and the gateways respectively.

Simple email worms are now considered the last generation threat. Today and the near future will be composed of blended threats and their damage is still yet unseen. A downed mail server is now the least of our worries when threats can now effectively shutdown Internet backbones. Hopefully, the appearance of threats such as Win32/CodeRed and Win32/Nimda has given security professionals a wake-up call to prepare for the future as either threat could easily have been more damaging.

REFERENCES

[spaff88] Eugene H. Spafford, The Internet Worm Program: An Analysis.

[Gordon88] Sarah Gordon, 'The worm has turned', *Virus Bulletin*, August, 1998.

[McCorkendale-Szor] 'Code Red Buffer Overflow', *Virus Bulletin*, September 2001.

http://www.cve.mitre.org/, (Common Vulnerabilities and Exposures).

http://www.cert.org/.

[Szor99] 'Memory Scanning Under Windows NT', *Int. Virus Bull. Conf.*, 1999, pp.325–346, also see http://securityresponse.symantec.com/avcenter/reference/memory.scanning.winnt.pdf.

[Szor2000] 'Bolzano Bugs NT', *Virus Bulletin*, September 1999.

[Litchfield] Windows 2000 Format String Vulnerabilities, May 8, 2002.

[Howard-LeBlanc] Writing Secure Code, Chapter 12.

[Personal Communication] Bruce McCorkendale, Frederic Perriot.

[One] Smashing The Stack For Fun And Profit, Phrack 49, Vol. 7, Issue #49, File 14.

http://www.peterszor.com/badtrans.pdf 'Bad Transfer', *Virus Bulletin*, February 2002.

http://msdn.microsoft.com/workshop/components/activex/safety.asp.

http://msdn.microsoft.com/library/en-us/script56/html/letcreatetypelib.asp.

APPENDIX A.

This is a partial log of the threads inside INETINFO.EXE process (Microsoft's IIS) after being infected by both CodeRed I and CodeRed II. Any thread is identified as an active one and detected based on the signature of the virus code found at a thread start address. This is to ensure complete avoidance of ghost positives that could appear since unsuccessful worm attacks could still place worm code to application heap in inactive form. In tests, attempts to freeze the detected CodeRed threads were successful in stopping the worm from spreading further, and in gaining sufficient CPU time for patch installation processing.

Notice the high context switch number for worm related threads even after only a few seconds of infections. CodeRed II infections were new fresh and they have a lower context switch number, but notice that most CodeRed II thread have almost identical context switches values.

```
PID: 0x03b0

Threads:
TID CTXSWITCH  LOADADDR  WIN32STR  STATE
3ac    63  77e878c1  01002ec0  Wait:Executive
260    458  77e92c50  77dc95c5  Wait:Userrequest
410    927  77e92c50  78002432  Wait:Userrequest
414    921  77e92c50  78002432  Wait:Userrequest
418    131  77e92c50  00000000  Wait:Lpcreceive
41c    459  77e92c50  77dc95c5  Wait:Userrequest
.
```

```
.
.
494    2  77e92c50  6a176539  Wait:Userrequest
498    8  77e92c50  6d703017  Wait:Userrequest
49c    7  77e92c50  69de3ce1  Wait:Userrequest
4a0    1  77e92c50  69e0d719  Wait:Eventpairlow
4a4    1  77e92c50  69e0d719  Wait:Eventpairlow
4a8    1  77e92c50  69e0d719  Wait:Eventpairlow
4ac    1  77e92c50  69e0d719  Wait:Eventpairlow
4b0  186  77e92c50  69dda9b7  Wait:Userrequest
4b4    2  77e92c50  77d4b759  Wait:Lpcreceive
4b8    2  77e92c50  74a82a31  Wait:Userrequest
4bc  178  77e92c50  6783b085  Wait:Userrequest
348  10507  77e92c50  730c752b  Wait:Userrequest
594    6  77e92c50  730c75e6  Wait:Lpcreply
2a4  702  77e92c50  6e9036de  Wait:Userrequest
598  10509  77e92c50  010ce918  CodeRed I Thread
59c  10509  77e92c50  0230fe7c  CodeRed I Thread
5a0  10510  77e92c50  0234fe7c  CodeRed I Thread
5a4  10509  77e92c50  0238fe7c  CodeRed I Thread
5a8  10509  77e92c50  023cfe7c  CodeRed I Thread
5ac  10510  77e92c50  0240fe7c  CodeRed I Thread
5b0  10510  77e92c50  0244fe7c  CodeRed I Thread
5b4  10510  77e92c50  0248fe7c  CodeRed I Thread
5b8  10511  77e92c50  024cfe7c  CodeRed I Thread
5bc  10509  77e92c50  0250fe7c  CodeRed I Thread
5c0  10510  77e92c50  0254fe7c  CodeRed I Thread
5c4  10509  77e92c50  0258fe7c  CodeRed I Thread
5c8  10509  77e92c50  025cfe7c  CodeRed I Thread
5cc  10510  77e92c50  0260fe7c  CodeRed I Thread
5d0  10509  77e92c50  0264fe7c  CodeRed I Thread
5d4  10509  77e92c50  0268fe7c  CodeRed I Thread
5d8  10509  77e92c50  026cfe7c  CodeRed I Thread
5dc  10509  77e92c50  0270fe7c  CodeRed I Thread
5e0  10509  77e92c50  0274fe7c  CodeRed I Thread
5e4  10509  77e92c50  0278fe7c  CodeRed I Thread
5e8  10509  77e92c50  027cfe7c  CodeRed I Thread
5ec  10509  77e92c50  0280fe7c  CodeRed I Thread
5f0  10510  77e92c50  0284fe7c  CodeRed I Thread
5f4  10509  77e92c50  0288fe7c  CodeRed I Thread
.
.
.
708  10509  77e92c50  039cfe7c  CodeRed I Thread
70c  10509  77e92c50  03a0fe7c  CodeRed I Thread
710  10510  77e92c50  03a4fe7c  CodeRed I Thread
714  10509  77e92c50  03a8fe7c  CodeRed I Thread
718  10509  77e92c50  03acfe7c  CodeRed I Thread
71c  10509  77e92c50  03b0fe7c  CodeRed I Thread
720  10509  77e92c50  03b4fe7c  CodeRed I Thread
724    2  77e92c50  03b8fe7c  CodeRed I Thread
26c  65  77e92c50  00000000  Wait:Lpcreceive
518    1  77e92c50  6d70175a  Wait:Eventpairlow
320    7  77e92c50  6d70175a  Wait:Eventpairlow
568  839  77e92c50  004202a1  CodeRed II Thread
58c  810  77e92c50  004202a1  CodeRed II Thread
```

```
390    810  77e92c50  004202a1  CodeRed II Thread
4d8    810  77e92c50  004202a1  CodeRed II Thread
190    810  77e92c50  004202a1  CodeRed II Thread
564    810  77e92c50  004202a1  CodeRed II Thread
294    810  77e92c50  004202a1  CodeRed II Thread
4dc    810  77e92c50  004202a1  CodeRed II Thread
488    811  77e92c50  004202a1  CodeRed II Thread
120    810  77e92c50  004202a1  CodeRed II Thread
.
.
.
800    814  77e92c50  004202a1  CodeRed II Thread
804   7868  77e92c50  74fd68fd  Wait:Eventpairlow
808    813  77e92c50  004202a1  CodeRed II Thread
80c    812  77e92c50  004202a1  CodeRed II Thread
810    812  77e92c50  004202a1  CodeRed II Thread
814    812  77e92c50  004202a1  CodeRed II Thread
818    812  77e92c50  004202a1  CodeRed II Thread
81c    812  77e92c50  004202a1  CodeRed II Thread
820    812  77e92c50  004202a1  CodeRed II Thread
.
.
.
b3c    812  77e92c50  004202a1  CodeRed II Thread
b40    812  77e92c50  004202a1  CodeRed II Thread
b44    814  77e92c50  004202a1  CodeRed II Thread
b48    812  77e92c50  004202a1  CodeRed II Thread
b4c    812  77e92c50  004202a1  CodeRed II Thread
b50    812  77e92c50  004202a1  CodeRed II Thread
b54    812  77e92c50  004202a1  CodeRed II Thread
```

APPENDIX B

Character	Type	Output format
c	**int** or **wint_t**	When used with **printf** functions, specifies a single-byte character; when used with **wprintf** functions, specifies a wide character.
d	**Int**	Signed decimal integer.
i	**int**	Signed decimal integer.
o	**int**	Unsigned octal integer.
u	**int**	Unsigned decimal integer.
x	**Int**	Unsigned hexadecimal integer, using "abcdef."
X	**Int**	Unsigned hexadecimal integer, using "ABCDEF."
e	**Double**	Signed value having the form [–]*d***.***dddd* **e** [*sign*]*ddd* where *d* is a single decimal digit, *dddd* is one or more decimal digits, *ddd* is exactly three decimal digits, and *sign* is + or –.
E	**Double**	Identical to the **e** format except that **E** rather than **e** introduces the exponent.
f	**Double**	Signed value having the form [–]*dddd***.***dddd*, where *dddd* is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.
g	**Double**	Signed value printed in **f** or **e** format, whichever is more compact for the given value and precision. The **e** format is used only when the exponent of the value is less than –4 or greater than or equal to the precision argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	**Double**	Identical to the **g** format, except that **E**, rather than **e**, introduces the exponent (where appropriate).
n	Pointer to integer	Number of characters successfully written so far to the stream or buffer; this value is stored in the integer whose address is given as the argument.
p	Pointer to **void**	Prints the address of the argument in hexadecimal digits.
s	String	When used with **printf** functions, specifies a single-byte–character string; when used with **wprintf** functions, specifies a wide-character string. Characters are printed up to the first null character or until the *precision* value is reached.

Source: MSDN Library January 2002.