**Practical 14 : Buffer Overflows**

Objective:

To be able to:

* Understand the Instruction Pointer
* Describe a buffer overflow
* Distinguish between a Stack Overflow and a Heap Overflow
* Examine ways to prevent buffer overflows

**Exercise Instruction Pointer (IP) or Extended Instruction Pointer (EIP)**

The Instruction Pointer (or EIP) contains the address of the next instruction to execute. In this example, we will use a simple 16-bit assembly program on the old WinXP virtual machine and use the default debug command to try out debugging to view and change the 16-bit Instructor Pointer.

In WinXP VM:

1. Create the following assembly program “nop.asm”.

dosseg

.model small

.stack

.code

main proc

nop

mov ax,34h ; move the value 34 to ax

nop

mov ax,4C00h ; exit program with return code 0

int 21h

; this section of code will not be reached

nop

mov ax,36h ; move the value 36 to ax

nop

main endp

end main

1. In a Command Prompt, use the Turbo Assembler to assemble the program.

tasm nop.asm

1. If no errors are reported, check that a nop.obj file has been created in the folder.
2. In the Command Prompt, use the Turbo Linker to link the object file.

tlink nop.obj

1. If no errors are reported, check that the executable file nop.exe has been created in the folder.
2. Download and install a hex editor (eg WinHex or HxD Hex Editor). Both hex editors are also available on Brightspace or the usual download link, under Files-for-Topic13 :

If you need to copy files from your HostPC to your WinXP VM, and the Copy and Paste File does not work, you could try using shared folders.

1. Use a hex editor to open the executable program. Look for the line that looks like following. These bytes represent the compiled instructions in the program.



**Questions**

1. There are four “NOP”s in the assembly program. Which bytes in the executable do you think represent the instruction “NOP”?

Ans : \_\_\_\_\_90\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(The byte value “90” appears four times in the executable. So it is likely the byte “90” represent the instruction “NOP”)

1. What is the offset address for the instruction ‘mov ax,34h’?

Ans : \_\_\_\_\_201h\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(In the example above, the byte “34” is the third byte. So the bytes “B8 34 00” likely represent the instruction “mov ax,34h”. The offset of the first byte in “B8 34 00” is 201h.

1. What is the offset address for the instruction ‘mov ax,36h’?

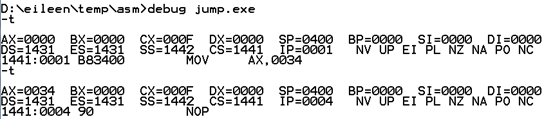
Ans : \_\_\_\_\_\_20Bh\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(In the example above, the byte “36” is the thirteenth byte. So the bytes “B8 36 00” likely represent the instruction “mov ax,36h”. The offset of the first byte in “B8 36 00” is 20Bh.

1. Use the debugger to load the program.

debug <executable filename>

1. Trace through the program by entering “t”. Note the address stored in the Instruction Pointer (IP).



IP contains the address of the next instruction to be executed

After running ‘MOV AX,0034’, IP is incremented to the address of the next instruction

AX now contains 0034h

1. Quit the debugger.

This time, you will use the debugger to change the address stored in the IP so that the “unreachable” code will be run.

1. Use the debugger to load the program again.

debug <executable filename>

1. Trace through the first line of code “NOP”



1. Set the value in IP to the offset address of the instruction “mov ax,36h”.



1. When you trace through the next line of code, the instruction “mov ax,36h” is executed.



AX now contains 0036h

By changing the value in the Instruction Pointer, you can control which instruction will be executed next.

**Exercise A Buffer Overflow**

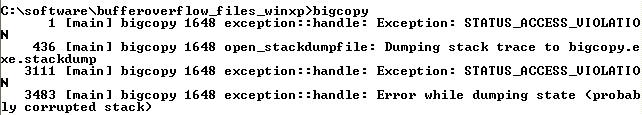
In July 2000, it was discovered that Microsoft Outlook and Outlook Express let attackers compromise target computers simply by sending e-mail messages. No one even had to open a message; as soon as the user downloaded the message, message-header routines went into action - with unchecked buffers that could overflow and trigger code execution. Microsoft has since created a patch that eliminates the vulnerability.

With buffer overflows being able to do anything from escalating privileges to crashing whole systems, the need to ensure your code is free of buffer overflow vulnerabilities is very necessary.

Using libraries which are not prone to buffer overflows, validating data length checking string formats and integer sizes are all excellent ways of eradicating buffer overflows.

In WinXP VM:

1. Download the bufferoverflow\_files\_winxp.zip from the usual download link under Files-for-Topic14 or from Brightspace.
2. Extract the contents of the zip file.
3. In a Command Prompt, run the bigcopy.exe. (Make sure the file cygwin1.dll is in the same directory as bigcopy.exe.) What is the output of the program?



The bigcopy.exe program crashes, and you may see output like this

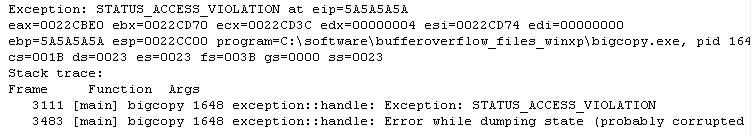
1. The program bigcopy.exe was compiled from the source code below. Can you see anything wrong with the code?

Source code for bigcopy.exe

|  |
| --- |
| void copy(char \*large) {  char small[16];  strcpy is a C function.  strcpy(destination, source) will copy a string from “source” to “destination”  strcpy(small, large);  }  int main() {  char large[256];  int i = 0;  for(i = 0;i < 255;i++) {  large[i] = 'Z';  }  copy(large);  } |

(What’s wrong with the code : The strcpy command will copy from “large” buffer (256 characters) to “small” buffer (16 characters), resulting in buffer overflow)

1. After the program bigcopy.exe crashed, look at the contents of the folder where bigcopy.exe is stored. There is a new file called “bigcopy.exe.stackdump” that has been created.
2. Open the stackdump file using a text editor like WordPad. The stackdump file contains the values of the registers and stack when the program crashed.



The instruction pointer eip contains the value “5A5A5A5A”. “5A” is the ascii value for “Z”. So the 512 “Z”s in the large buffer overflowed the small buffer and caused the instruction pointer to have the wrong value.

**Questions:**

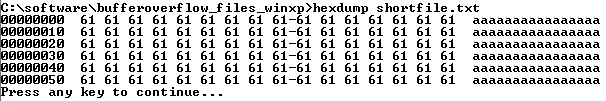
The strcpy function is prone to buffer overflows. How would you ensure that a string is copied safely so that a buffer overflow would not occur?

Check the size of the string to be copied and the size of the destination buffer before doing the copying.

**Exercise Another Buffer Overflow example**

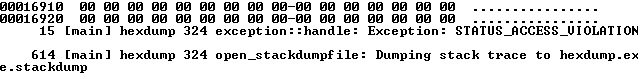
In WinXP VM:

1. The program hexdump.exe opens a file, reads the file contents, changes the characters into the ascii hex value and displays to the screen.
2. In a Command Prompt, run the hexdump program with shortfile.txt as the argument.



The hexdump program runs without errors with shortfile.txt

1. Run the hexdump program with longfile.txt as the argument.



With longfile.txt, the hexdump program crashes.

1. The program hexdump.exe was compiled from the source code below.

Source code for hexdump.exe

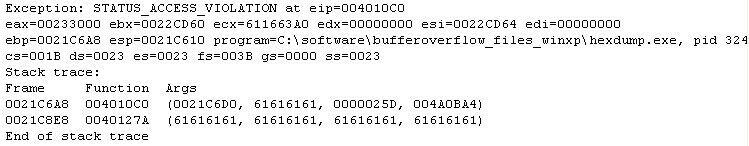
|  |
| --- |
| #include <windows.h>  #include <stdio.h>  #include <string.h>  static void DumpHex(unsigned char\* pData, int dwLen)  {  char szAscii[17];  char szHex[17\*3];  int dwCol = 0;  int leftI = 0;  szAscii[0] = 0;  szHex[0] = 0;  int i = 0;  for (i=0; i<dwLen; i++)  {  char szDigit[4];  if ((dwCol == 8) && ((i + 1) < dwLen))  sprintf(szDigit, "%2.2x-", pData[i]);  else  sprintf(szDigit, "%2.2x ", pData[i]);  strcat(szHex, szDigit);  char asciiDigit = '.';  if ((pData[i] >= ' ') && (pData[i] < 127))  asciiDigit = (char)pData[i];  szAscii[dwCol + 1] = 0;  szAscii[dwCol] = asciiDigit;  dwCol++;  if (dwCol >= 16)  {  printf("%8.8x %s %s\n", leftI, szHex, szAscii);  szAscii[0] = 0;  szHex[0] = 0;  dwCol = 0;  leftI = i + 1;  }  }  if (dwCol > 0)  {  // Pad the line to align it  while (dwCol < 16)  {  strcat(szHex, " ");  strcat(szAscii, " ");  dwCol++;  }  printf("%8.8x %s %s\n", leftI, szHex, szAscii);  }  }  void DoHexDump(FILE\* file)  {  A buffer called “data” is declared. It is able to hold 512 characters  int len;  unsigned char data[512];  FILE\* fp = file;  fseek(fp, 0, SEEK\_END);  len = ftell(fp);  fseek(fp, 0, SEEK\_SET);  //Possible Overflow!  The fread function will read the contents of the file and copy it to the data buffer  fread(data, 1, len, fp);  fclose(fp);  DumpHex(data, len);  }  int main(int argc, char\* argv[])  {  char fileName[1024];  char cmdLine[65536];  if (argc < 2)  {  printf("Please specify file name ie hexdump <filename>");  return 0;  }  strcpy(fileName, argv[1]);  FILE\* fp = fopen(fileName, "rb");  int len;  if (!fp)  {  MessageBox(NULL, "Invalid file", "Error", MB\_ICONSTOP | MB\_OK);  return 0;  }  DoHexDump(fp);  printf("Press any key to continue...\n");  getchar();  ExitProcess(0);  return 0;  } |

**Questions:**

Why did the shortfile.txt work while the longfile.txt failed?

The DoHexDump function will copy all content from the specified file to the data buffer (size 512 characters). The longfile.txt is bigger than 512 chars, so it will cause a buffer overflow.

After the program hexdump.exe crashed, look at the contents of the folder where hexdump.exe is stored. There is a new file called “hexdump.exe.stackdump” that has been created. Open this stackdump file using a text editor like WordPad.



Some of the bytes in the stack contain the value 61616161. Why is this so?

When longfile.txt is copied to the data buffer, its contents overflowed the data buffer and overwrite other parts of the memory. “61” is the ascii code for ‘a”.

How would you prevent a buffer overflow in this situation?

One suggestion : If the file is too big for the buffer, read and display the file contents in batches.

**Exercise Using Buffer Overflow to modify variables**

1. The program buffervalue.exe was compiled from the source code below.

Source code for buffervalue.exe

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define BUFSIZE 16  int main(int argc, char\* argv[])  {  char \*buffer1 = (char \*)malloc(BUFSIZE);  if(buffer1 == NULL)  return 0;  char \*buffer2 = (char \*)malloc(BUFSIZE);  if(buffer2 == NULL)  return 0;  memset(buffer1, 'A', BUFSIZE-1);  buffer1[BUFSIZE-1] = '\0';    memset(buffer2, 'A', BUFSIZE-1);  buffer2[BUFSIZE-1] = '\0';    printf("buffer1 pointer = %p, buffer2 pointer = %p", buffer1, buffer2);  printf("\n\nValue in buffer1: %s", buffer1);  printf("\nValue in buffer2: %s", buffer2);    printf("\n\nEnter new value to be placed in buffer1 with gets(): ");  gets(buffer1);    printf("buffer1 pointer = %p, buffer2 pointer = %p", buffer1, buffer2);  printf("\n\nValue in buffer1: %s", buffer1);  printf("\nValue in buffer2: %s", buffer2);    printf("\n\nPress enter to (try) and free buffer1...");  getchar();  free(buffer1);    printf("\n\nPress enter to (try) and free buffer2...");  getchar();  free(buffer2);  return 0;  } |

1. Try to run the program buffervalue and trigger a buffer overflow.

**Questions:**

How many bytes separate the start of buffer1 and buffer2? (Look at the values contained in buffer1 pointer and buffer2 pointer)

Eg. If the program output is as follows :



So the memory space allocated for buffer1 and buffer2 is as follows :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 2A | 2B | 2C | 2D | 2E | 2F |
|  |  |  |  |  |  |  | buffer1 |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 30 | 31 | 32 | 33 | 34 | 34 | 36 | 37 | 38 | 39 | 3A | 3B | 3C | 3D | 3E | 3F |
|  |  |  |  |  |  |  |  |  |  |  |  | buffer2 |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 4A | 4B | 4C | 4D | 4E | 4F |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Number of bytes separating start of buffer1 and buffer2 = 0x438 – 0x420 (in hexadecimal)

= 0x18 (in hexadecimal) = **24 bytes** (in decimal)

How would you use a buffer overflow to modify other variables in a program? (Hint : enter about 30 characters to be placed in buffer1 and then observe the values of both buffer1 and buffer2.

If we input more than 24 chars into buffer1, the 25th char onwards will overflow into buffer2.

How would you prevent a buffer overflow situation in your code?

Suggestion : Control size of user input.

Is this a stack or heap overflow?

(Hint: something to do with the malloc command and how space is allocated for the data)

Heap overflow. Malloc will allocate memory space from the heap.

**Exercise Using buffer overflow to crack a program**

1. The program login.exe is an implementation of a simple login module.
2. Some of the code for the login module was leaked out.

int authenticate() {

char terminator = '\0';

char authPass = 'F';

char pw[8];

puts("Please enter your Password:");

gets(pw);

if(!strcmp(pw, "password"))

authPass = 'T';

return (authPass == 'T');

}

1. Run login.exe and try to crash the application.
2. Using buffer overflow techniques, try to gain access into the system without the actual password.

**Questions:**

How did you manage to crash the application?

What steps did you take to gain access without the actual password?

Looking at the code, the pw buffer can be “overflowed”, If we put enough “T”s in pw, we may overwrite authPass with a “T”. However, entering too many Ts for the password can cause the program to crash. By trial and error, we can find that entering fifteen “T”s will give us the message “You have gain accessed” (yes, bad grammar)

~~~ End of Practical ~~~