# ENPM691 – Hacking of C Programs and Unix Binaries

# Homework – 3

*Author – Syed Mohammad Ibrahim*

*UMD ID:* *iamibi*

*Email:* [*iamibi@umd.edu*](mailto:iamibi@umd.edu)

## Ret2Text

* 1. Screenshot(s)

A screenshot of a computer

Description automatically generated

* 1. Walkthrough
     1. Created a file called ret2text.c with the given program as per the paper.
     2. I compiled it using the command “gcc ret2text.c -o ret2text -fno-stack-protector -zexecstack”
     3. ASLR was on during the compilation and execution.
     4. I opened the program in gdb to inspect the “public” function which contained the code for strcpy and which will help me execute the “secret” function if I figure out the buffer size and successfully overflow it.
     5. The disassembly of “public” function is as follows:

Text

Description automatically generated

* + 1. The esp is down by 0x18 (24 bytes), which means the buffer size that we must overflow is 24 bytes.
    2. Now we require the address of the “secret” function which will be added as part of the script payload once the buffer overflows. After the buffer overflows, the return address to secret function will be called as ESP will be overwritten. The address of secret function can be found by disassembling main function

A picture containing text

Description automatically generated

* + 1. Executing the perl command with the return address of secret function: `perl -e ‘print “A” x 24; print “\x96\x84\x04\x08”’`
    2. We get the buffer overflowed and the function secret gets executed as seen in 1.1 section.

## Ret2Bss

* 1. Screenshot(s)

A screenshot of a computer

Description automatically generated

* 1. Walkthrough
     1. I wrote the code mentioned in the aslr attack paper for ret2bss.
     2. ASLR was on during the time of compilation.
     3. The command used for compiling the program was: “gcc ret2bss.c -o ret2bss -fno-stack-protector -zexecstack”
     4. Opening the program in gdb and disassembling “function” gave the buffer size of 0x108 (264).

Text

Description automatically generated

* + 1. The perl script will be modified to adjust the padding, shell code and return address as shown in the 2.1 screenshot.
    2. Padding is calculated by 264 – 24 (shellcode to spawn a bash shell in bytes) + 4(return address) = 244 bytes.
    3. For return address, we checked the “globalbuff” address which was 0x804a040.
    4. Entering the above data in the perl script and compiling it using “perl exploit\_ret2bss.pl” gave an executable binary.
    5. Finally running the exploit on the program using “./ret2bss `cat ../PerlScript/payload\_bss`” resulted in getting the bash shell.

## StrPtr

* 1. Screenshot(s)

A screenshot of a computer

Description automatically generated

* 1. Walkthrough
     1. I wrote the program mentioned in the ASLR paper.
     2. Compiled the program using: “gcc strptr.c -o strptr -fno-stack-protector -zexecstack”
     3. ASLR was turned off during the compilation and execution. However, with ASLR on we can make this work.
     4. Next step was to get the address of the variable “license” as mentioned in the paper. After looking through the assembly of main function, 0x8048582 was the address of that variable. I was able to identify it using x/s command in gdb which printed the value.
     5. Now, we need to get the buffer value that we must overflow to get the shell. To do that, we have buffer offset of 0x110 (272). Brute forcing few numbers less than that gave me the number 260 which gave me the shell.
     6. There was a file called “THIS” created in the local directory and the PATH was added. This file contained the string “/bin/sh”. Reason for this file is that the program tries to execute “THIS” expecting it to be an executable file which we made it to be.
     7. Finally, we got the shell mentioned in 3.1.

## Divulge

* 1. Screenshot(s)

Graphical user interface, text

Description automatically generated

A screenshot of a computer

Description automatically generated

* 1. Walkthrough
     1. Implemented the program as mentioned in the ASLR paper.
     2. The program was compiled using: “gcc divulge.c -o divulge -g -fno-stack-protector -zexecstack”
     3. ASLR was on during compilation and execution.
     4. As suggested by the author of the paper, I needed the address of the writebuf variable. So opening the program in gdb and adding a breakpoint at the write() function call. Run the program in gdb.
     5. Open a new terminal and execute the command “echo AAAA | nc localhost 7776”
     6. This will trigger the breakpoint that I had put in.
     7. Printing the address of writebuf variable, which came out to be 0xbfffea10
     8. Next, getting the bottom address of the stack. It can be done by “cat /proc/`pidof divulge`/stat | awk ‘{ print $28 }’ ”. It came out to be 3221221632 (0xbffff100)
     9. The difference between the two addresses gives us the offset value 0xbffff100 - 0xbfffea10 = 1776 (6f0)
     10. Writing an exploit script, I used C to achieve this. The C program takes a command line argument which is the output of the above process id command.
     11. We calculate the offset by subtracting the base address (argument 1 to the program) with 1776 and converting it to bytes.
     12. We calculate the padding by looking at the disassembly of the divulge program. As it can be seen, the “function” contains strcpy and the offset of EIP to strcpy is 268 bytes

Text

Description automatically generated

* + 1. Running the divulge program normally in another terminal to start the server listening to port 7776 in localhost.
    2. Executing the exploit using “./divexploit `cat /proc/$(pidof divulge)/stat | awk ‘{ print $28 }’` | nc localhost 7776” gives us the shell on the terminal executing the divulge program.

## FuncPtr

* 1. Screenshot(s)

Text

Description automatically generated

* 1. Walkthrough
     1. Program was replicated from the given ASLR paper.
     2. Program was compiled using: “gcc funcptr.c -o funcptr -fno-stack-protector -zexecstack”
     3. ASLR was turned off during the program compilation and execution.
     4. Going over the assembly of the compiled program in gdb. This exploitation requires overflowing the buffer to the extent that the second argument becomes a command to be executed. Thus, our second command line argument will be “/bin/sh”.
     5. To figure out the first argument buffer size, we check the main function assembly code, and we find that the offset is 64 bytes (array size and pointer size adjustment).
     6. Now I needed the address of the “system” function. This was again taken out from the gdb disassembly.
     7. Executing the command “./funcptr `perl -e ‘print “A” x 64; print “\x40\x83\x04\x08”’` “/bin/sh” ” gave us the bash shell as mentioned in section 5.1 screenshot.

## Ret2Ret

* 1. Screenshot(s)

A screenshot of a computer

Description automatically generated

* 1. Walkthrough
     1. Replicated the program code as mentioned in the ASLR paper.
     2. The program was compiled using: “gcc ret2ret.c -o ret2ret -fno-stack-protector -zexecstack”
     3. The ASLR was turned on during the compilation and execution of the program.
     4. Executing a test statement using “./ret2ret AAAA” returned the value of 1.
     5. The exploit script I wrote contained the shellcode for exit(0) function call. Thus, if we successfully exploit the ret2ret binary, the return value should be 0 instead of 1.
     6. As per the paper and internet search, it was found that I would require the return address of the function that calls another function containing strcpy code. In our case it is “main” function as the main function is calling “function” which has the code for strcpy.
     7. The return address of “ret” instruction in main function is required and in my case it was 0x0804846c

Text

Description automatically generated

* + 1. Creating an exploit script in perl as shown in the screenshot of 6.1, we input the return address, padding and shellcode in the combination of pad + shellcode + return address.
    2. The return address is entered 8 times as the difference of pointer address and the buffer offset address comes out to be 32 bytes which upon dividing by 4 (as each return address is 4 bytes) we get 8.
    3. Padding is calculated using 0x108 (264) – 0x07 (shellcode size of exit(0)) + 4 bytes (offset to return address) = 261
    4. Running the program with the given exploit gives the return value of 0 instead of 1.

## Ret2Pop

* 1. Screenshot(s)

A screenshot of a computer

Description automatically generated

* 1. Walkthrough
     1. Implemented the program as per the ASLR paper.
     2. The program was compiled using: “gcc ret2pop.c -o ret2pop -fno-stack-protector -zexecstack”
     3. ASLR was on during the compilation and execution of the program.
     4. Shellcode used in the exploit script is of exit(0) function.
     5. To calculate the return address, as per the paper we need to get the address of the instruction “pop %ebp”. Looking over the object dump of the binary, we get the address of the pattern. The address is 0x080484cb

Text

Description automatically generated

* + 1. Next is the padding size. It can be calculated by the disassembly of the program. As it can be seen, the buffer is 264 (0x108).

Text

Description automatically generated

* + 1. To calculate the padding, 264 – 7 (size of exit(0) shell code) + 4 (return address size) = 261 bytes
    2. Put all the variables in the exploit script and compile it.
    3. Executing the ret2pop binary with the exploit as first argument we get the return address of 0 instead of 1 as shown in section 7.1.

## Ret2Esp

* 1. Screenshot(s)

A screenshot of a computer

Description automatically generated

* 1. Walkthrough
     1. Implemented the program as mentioned in the ASLR paper.
     2. The program was compiled using “gcc ret2esp.c -o ret2esp -fno-stack-protector -zexecstack”
     3. ASLR was on during the compilation and execution of the program.
     4. The shellcode being used in the exploit script is of exit(0) which is 7 bytes.
     5. As per the paper, ret2esp exploit requires the address of “jmp \*%esp” instruction.
     6. Looking over the object dump of the ret2esp binary, we get the address 0x0848442

Text

Description automatically generated

* + 1. Check the buffer size to calculate the offset in gdb. As it can be seen, it is 264 bytes. To calculate the padding we do 264 + 4 bytes as the padding will be at the bottom instead of the top.

Text

Description automatically generated

* + 1. Putting all the variables in the exploit script and executing the ret2esp program with the exploit as first argument, we get the program exited with return address 0 instead of 1.

## Ret2Got

* 1. Screenshot(s)

A screenshot of a computer

Description automatically generated

* 1. Walkthrough
     1. Implemented the program as per ASLR paper.
     2. Compiled the program using “gcc ret2got.c -o ret2got -fno-stack-protector -zexecstack”
     3. ASLR was on during the program compilation and execution.
     4. We need the address of second printf statement as it will be resolved to a “system” function call. To do that, we look at the assembly of the program and examine the second printf. The address is 0x08048320

A picture containing graphical user interface

Description automatically generated

* + 1. Next, we check what’s the address of the system call that will be passed as second parameter. It was 0x08048346

Text

Description automatically generated

* + 1. As the buffer size is 8 bytes, we pass the padding of “A” eight times along with the return address of the first printf. The second parameter is return address to system function call.
    2. If I execute the program using “./ret2got `perl -e ‘print “A” x 8; print “\x0c\xa0\x04\x08”’` `perl -e ‘print “\x46\x83\x04\x08”’` ”, the program throws an error saying that “Array” doesn’t exist.
    3. I created a file called “Array” in the local path with the bash script location in it using
* echo “/bin/sh” > Array
* chmod 777 Array
* PATH=.:$PATH
  + 1. After executing the same command for ret2got, I got the shell.

## Format String

* 1. Screenshot(s)

Text

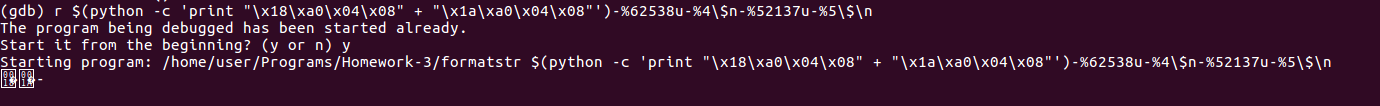
Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated with medium confidence



Text

Description automatically generated

* 1. Walkthrough
     1. Implemented the program given in the slide page 33.
     2. The program was compiled using: “gcc formatstr.c -o formatstr -fno-stack-protector -zexecstack”
     3. We need the address of the putchar statement as it will be used as part of exploitation.
     4. Going over the disassembly of the main function, the putchar address is 0x0804a018.
     5. Executing the above address with additional format string parameters in the command “$(python -c ‘print “\x18\xa0\x04\x08”’)-%10u-%4\$n”, we get a segmentation fault.
     6. As per instructions given in the slides, created an environment variable EGG containing the shell script of bash shell and padding, and executed the command again.

export EGG=$(python -c 'print "\x90" \* 64 + "\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x99\x52\x52\x89\xe1\xb0\x0b\xcd\x80"')

* + 1. After the segmentation fault, ran the command “find $esp, $esp+2000, 0x90909090” to get the address of the last nop. It was 0xbffff456.
    2. Since the input can’t hold such a big value, I split the address into two. Just taking the last two bytes 0xf456 (62550).
    3. Calculating the difference in the storage of 0x1bfff and 0xf456, we get 52137
    4. Using these two numbers in the input and brute forcing the value of second number, I got the shell.