CMSC 426 – Principles of Computer Security

Spring 2019

Lab 1 – Overflows

Full name: Daniel Roh
UMBC username: Droh1

Part 1: Gaining access to the VM

- **Investigate the dog.c file. Describe in detail what will allow you to cause a stack buffer overflow in the dog executable.**
  [5 pts]

-The use of "gets(user_input);" would allow an attacker to cause a stack buffer overflow.

- **Without the use of shell code, how can you exploit dog to allow the execution of arbitrary commands?**
  [10 pts]

-By using dog's program, you can type in a bunch of spaces with the linux command to want to run.

- **Write an exploit for dog that causes the command /bin/sh; to be executed. Include the exact input used to achieve your exploit below (i.e., what was your command line input?). Draw what the stack will look like after your exploit is run (if you insert a picture taken with your phone, make sure it is readable).**
  [20 pts]

```
"                                                                 bash"
```

65 spaces then "bash"

Main
variables
EBP
“/bash”
“   “ <- a lot of spaces

- **Access the flag1.txt file and follow the instructions contained within it. Put your response below.**
  [5 pts]

-Dog Breed = Dalmatian.

-I didnt know which one to choose so I asked a few friends and they said to choose a dalmation because because, "I was put on the spot" (True story). (>.<)

Part 2: Privilege escalation

- **Investigate the pokemon executable's properties. Why is it a good target for privilege escalation?**
  [5 pts]

-Pokemon has the SUID bit set giving privileges of the owner when the program is being run

- **Investigate the pokemon.c file. How is it possible to cause a stack overflow in pokemon? What constraints are there on the input you provide to the program?**
  [5 pts]

-Its possible to run a stack overflow because of the use of 'strcpy' in the 'play_pokemon' function.
-Constraints are the program that the program due to the use of strncmp, we need to trick the program into thinking that we gave it a valid input so that the program does not exit.

- **What is the minimum number of bytes of input required to fully overwrite the return address? Describe how you got your answer. (Protip: Your answer should be a multiple of 4. If you overwrite the return address and get a segfault, you can use gdb to find out which address the program attempted to return to.)**
  [10 pts]

-The minimum needed to overwrite the return address would be 152 bytes.

-I found this by adding 'A' into the input until there was a segfault, then used gdb to continue until gdb showed that the return address on the stack was overwritten.

- **Find the address of the vulnerable buffer using gdb. List and explain which commands you used to find the address.**
  [10 pts]

User_choice [ ] = 0xbfffee28

Command used: "break 4"
"p &user_choice"

I used GDB and made a breakpoint after the initialization of both char arrays. Then I ran the p &user_choice command to get the address.

- **Using the provided pokesploit.c file as a starting point, develop an exploit for pokemon. You will need to fill in many of the missing values, such as the locations of the NOP sled, shellcode, and return address in the exploit payload. Describe how you determined each of these missing values and how the exploit works.**
  [5 pts]

exploit_len = 152.
->This was chosen based on the length of the input needed to for gdb to show that the return address was overwritten.

nop_sled_offset = 0
nop_sled_len = 152
->The length of nop_sled_len was chosen, since the length to overwrite the return address was 152, that would mean that the length of the nop_sled would need to the same to overwrite the address. No offset was given as with the nop_sled as the offset I wanted to overwrite everything with nops.

shellcode_offset = 20
-> I chose 20, as it was 4 byte aligned and was close to the return address. It was also away from the return address to avoid interfering with the address values.

ret_addr = 0xbfffeda8
-> This was chosen as this is where the buffer is in memory where the beginning of Computer_choice was at, and thus was close to the location of user_choice.

*mystery = "Bulbasaur"
mystery_offset = 8
-> Not exactly sure what the mystery was for. But my guess is, due to the use of strncmp, to allow the program to continue to the next part of the program. A Bulbasaur was put at the

beginning of the user_choice buffer to make the program think that "Bulbasaur" was the input to avoid the program from exiting..

-The exploit created by first generating an input file that is longer then the buffer that the program is expecting. In this case, user_choice buffer. In this input file, a sequence of characters are fed to overwrite the null characters that would be in place once the strcmp is used. It is also used to overwrite the return address location on the stack. The return address to the end of the exploited buffer is then placed in the position to where the original return address should have been. Then nop's are added to the rest of the file to create the nopsled. Lastly, the shell code is added near the end of the nop sled and a null character is added to the end of the file to tell strcmp to finish the copy.

-This exploit works once the program hits the modified return address. Once it hits the return address, the code moves back to the buffer which is now a nop sled. Since the buffer is filled with nop's, the program keeps moving up the stack until it hits the shell code. Once the shell code is done, the code moves the argument to create a shell into the registers and then runs a system call to run the shell code. At this point, a new shell is created with the permissions that the program has access too.

- **Turn in your pokesploit.c file separately and make sure to fill out the header at the top.**
  [20 pts]

[Submitted as separate file (do not paste the contents here)]

- **Access the flag2.txt file and follow the instructions contained within it. Put your response below.**
  [5 pts]

->Space Mission = Parker Solar Probe
->I choose this mission because this one of the missions that a professor I know worked on. Did you know that the wires connected to the magnetic field sensor are made of niobium and the insulator for the wires are made of sapphire beads.