Directions

A. Due Date: 29 June 2025 at 11:55pm

B. The homework may be done in groups of up to two students.

What to turn in

C. Turn in all related source code (.c files) along with any headers or supplemental libraries needed to compile the code.

How to submit

D. Turn your results in using the dedicated Moodle assignment.

Lab 1: Set UID Buffer Overflow Attacks

1 Task 1: Invoking the Shellcode

The code above includes two copies of shellcode, one is 32-bit and the other is 64-bit. When we compile the program using the -m32 flag, the 32-bit version will be used; without this flag, the 64-bit version will be used. Using the provided Makefile, you can compile the code by typing make. Two binaries will be created, a32.out (32-bit) and a64.out (64-bit). Run them and describe your observations. It should be noted that the compilation uses the execstack option, which allows code to be executed from the stack; without this option, the program will fail.

Include screen shots of the 32-bit running and the 64-bin running.

1.1 Answers

The student response must include the following elements:

1. Screen shot of the 32 bit executable running
2. Screen shot of the 64 bit executable running
3. An explanation of what they see. There should be something about seeing the shell running and the user name that the shell code invoked has. Ideally they will use the make setuid make command and see that the shell code runs as root due to setuid. At least they should use the regular make command and see that it shows the user name of the regular user as a result.

2 Task 2: Understanding the Vulnerable Program

The compilation and setup commands are already included in Makefile, so we just need to type make to execute those commands. The variables L1, ..., L4 are set in Makefile; they will be used during the compilation.

Show screen shots of the results of running the Makefile

2.1 Answers

The result is just showing the output of running make on the code directory. Nothing else is required.

3 Task 3: 5.2 Launching Attacks

To exploit the buffer-overflow vulnerability in the target program, we need to prepare a payload, and save it inside badfile. We will use a Python program to do that. We provide a skeleton program called exploit.py, which is included in the lab setup file. The code is incomplete, and students need to replace some of the essential values in the code.

Fill in the code for the file exploit.py. Put the code you wrote here.

3.1 Answers

There are three values to fill in here. The students must get values for start, ret, and offset.

1. The start value should be 517 minus the length of the shell code. The result is 400, so either the absolute value or the calculation should be there.

2. The ret value should be the base pointer ebp plus the size of the buffer in code. That's 200 based on the value provided in the initial code. The value should be either absolute or with a calculation with the base pointer and the size.
3. The offset should be 200.

4 Task 3: 5.2 Launching Attacks - exploit

After you finish the above program, run it. This will generate the contents for badfile. Then run the vulnerable program stack. If your exploit is implemented correctly, you should be able to get a root shell:

Show a screen shot of your program after it gets the shell.

4.1 Answers

The result must show a root shell being opened. It should have the # icon at the beginning of the line. Ideally they should write whoami to show that they're root.

5 Task 3: 5.2 Launching Attacks - explanation

In your lab report, in addition to providing screenshots to demonstrate your investigation and attack, you also need to explain how the values used in your exploit.py are decided. These values are the most important part of the attack, so a detailed explanation can help the instructor grade your report. Only demonstrating a successful attack without explaining why the attack works will not receive many points.

Write an explanation about how the values in your exploit.py are decided.

5.1 Answers

There are three values that they must explain.

start The value should be something to do with where the shell code must be in the attack buffer. They can measure the length of the shell code and then make it relative to the buffer size of 517. It should say something about the sled and the place of the shell code inside it.

ret Must say something about using GDB to find the ebp value. Adding some safe value on top of it to make sure the return address lands in the nop sled.

offset That must be calculated from the size of the buffer and return addresses using gdb.

6 Task 4: Launching Attack without Knowing Buffer Size (Level 2) Bonus

In the Level-1 attack, using gdb, we get to know the size of the buffer. In the real world, this piece of information may be hard to get. For example, if the target is a server program running on a remote machine we will not be able to get a copy of the binary or source code. In this task, we are going to add a constraint: you can still use gdb, but you are not allowed to derive the buffer size from your investigation. Actually the buffer size is provided in Makefile, but you are not allowed to use that information in your attack

Your task is to get the vulnerable program to run your shellcode under this constraint. We assume that you do know the range of the buffer size, which is from 100 to 200 bytes. Another fact that may be useful to you is that, due to the memory alignment, the value stored in the frame pointer is always multiple of four (for 32-bit programs)

Please be noted, you are only allowed to construct one payload that works for any buffer size within this range. You will not get all the credits if you use the brute-force method, i.e., trying one buffer size each time. The more you try, the easier it will be detected and defeated by the victim. That's why minimizing the number of trials is important for attacks.

Turn in the code exploit.py for this part of the assignment.

Explain how you derived the values in the code.

6.1 Answers

This one requires them to use spraying, where they put the return address in multiple places in the payload. That's usually done with a loop at the end of the fill in your own section, something that steps every 4 bytes and put the target address. They also need to say something about what they did to make it work.

7 Task 5: Launching Attack on 64-bit Program (Level 3) Bonus

In this task, we will compile the vulnerable program into a 64-bit binary called stack-L3. We will launch attacks on this program. The compilation and setup commands are already included in Makefile. Similar to the previous task, detailed explanation of your attack needs to be provided in the lab report.

Using gdb to conduct an investigation on 64-bit programs is the same as that on 32-bit programs. The only difference is the name of the register for the frame pointer. In the x86 architecture, the frame pointer is ebp, while in the x64 architecture, it is rbp.

Put your code for exploit.py in your solution here. You can find the steps for making the exploit file in the explanation chapter on Moodle.

Write an explanation for how you wrote the exploit code.

7.1 Answers

The new code here must have the larger addresses for x64 as expected. The explanation must explain where the values for offset came from and where the rbp value came into play. They can't have 0 bytes in their return address or else the copying won't work. Need to check that they considered that and are aware of the little endian plus 0 issue in their solution.

8 Task 6: Launching Attack on 64-bit Program (Level 4) Bonus

The target program (stack-L4) in this task is similar to the one in the Level 2, except that the buffer size is extremely small. We set the buffer size to 10, while in Level 2, the buffer size is much larger. Your goal is the same: get the root shell by attacking this Set-UID program. You may encounter additional challenges in this attack due to the small buffer size. If that is the case, you need to explain how your have solved those challenges in your attack.

Turn in your exploit.py code here. You can find instructions on how to solve the challenge in the chapter on Moodle.

Explain how your exploit code works here.

8.1 Answers

The solution involves the same techniques as before, just this time with some different values for the buffer and ret. The students must give an explanation how they got what they did and the values they used. Some

students used environment variables for the exploit, but in that case they need to make it clear where the environment variable is defined and how they used it. The also might try to use return to libc ideas, but that's outside of the scope of the lab.

9 Task 7: Defeating dash's Countermeasure - Experiment

The dash shell in the Ubuntu OS drops privileges when it detects that the effective UID does not equal to the real UID (which is the case in a Set-UID program). This is achieved by changing the effective UID back to the real UID, essentially, dropping the privilege. In the previous tasks, we let /bin/sh points to another shell called zsh, which does not have such a countermeasure. In this task, we will change it back, and see how we can defeat the countermeasure.

9.1 Experiment

Compile call shellcode.c into root-owned binary (by typing "make setuid"). Run the shellcode a32.out and a64.out with or without the setuid(0) system call. Describe and explain your observations.

9.2 Answers

The result should explain what happens with and without the setuid system call. It would be best if they have screen shots of the results and clear explanations of all four states - a32, a64, with setuid, without setuid.

10 Tasks 7: Launching the attack again

Now, using the updated shellcode, we can attempt the attack again on the vulnerable program, and this time, with the shell's countermeasure turned on. Repeat your attack on Level 1, and see whether you can get the root shell. After getting the root shell, please run the following command to prove that the countermeasure is turned on. Although repeating the attacks on Levels 2 and 3 are not required, feel free to do that and see whether they work or not.

```
# ls -l /bin/sh /bin/zsh /bin/dash
```

Show a screen shot of the results of the attack after the change along with the above command.

10.1 Answers

The results must show that the exploit still works even after moving to dash. Need to show the attack still works and that the shell is the right one.

11 Task 8: Defeating Address Randomization

On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have $2^{19} = 524,288$ possibilities. This number is not that high and can be exhausted easily with the brute-force approach. In this task, we use such an approach to defeat the address randomization countermeasure on our 32-bit VM. First, we turn on the Ubuntu's address randomization using the following command.

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

Then run the same attack against stack-L1. Take a screen shot of the results and explain what happens.

We then use the brute-force approach to attack the vulnerable program repeatedly, hoping that the address we put in the badfile can eventually be correct. We will only try this on stack-L1, which is a 32-bit program.

You can use the shell script in bruteforce.sh (included in the lab setup files) to run the vulnerable program in an infinite loop. If your attack succeeds, the script will stop; otherwise, it will keep running. Please be patient, as this may take a few minutes, but if you are very unlucky, it may take longer.

Take a screen shot of the results and explain what happens.

11.1 Answers

The results should show that the first attempt fails with a segmentation fault. After running the brute force tool, there should be a success message after a few thousand attempts. The screen shot must show failure and then success. The explanation must say something about how the randomization makes most attempts fail, but after enough tries it hits the correct number for the return pointer.

12 Task 9.a: Turn on the StackGuard Protection

Many compilers, such as gcc, implement a security mechanism called StackGuard to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will not work. In our previous tasks, we disabled the StackGuard protection mechanism when compiling the programs. In this task, we will turn it on and see what will happen.

First, repeat the Level-1 attack with the StackGuard off, and make sure that the attack is still successful. Remember to turn off the address randomization, because you have turned it on in the previous task. Then, we turn on the StackGuard protection by recompiling the vulnerable stack.c program without the `-fno-stack-protector` flag. In gcc version 4.3.3 and above, StackGuard is enabled by default.

Launch the attack.

Turn in a screen shot of the result. Explain what you see.

12.1 Answers

The screen shot should show running the attack and the message that the stack was smashed and therefore the program crashed. The explanation must have something about the canary value.

13 Task 9.b: Turn on the Non-executable Stack Protection

Operating systems used to allow executable stacks, but this has now changed: In Ubuntu OS, the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the gcc, which by default makes stack non-executable. We can specifically make it non-executable using the "`-z noexecstack`" flag in the compilation. In our previous tasks, we used "`-z execstack`" to make stacks executable.

In this task, we will make the stack non-executable. We will do this experiment in the shellcode folder. The call shellcode program puts a copy of shellcode on the stack, and then executes the code from the stack. Please recompile call shellcode.c into a32.out and a64.out, without the "`-z execstack`" option.

Run the two programs.

Take screen shots of the results. Describe what you see.

13.1 Answers

The screen shot should show running the attack and the segmentation fault message. The explanation must have something about the stack being non-executable and therefore when trying to run code on the stack, the program fails.