CS 558 Introduction to Computer Security, Spring 2024
Programming Assignment 3

Out: 3/29/2024 Fri.
Due: 4/13/2024 Sat. 23:59:59

For this programming assignment, the goal is for students to gain practical experience with buffer overflow vulnerabilities, reinforcing their classroom learning on this subject. A buffer overflow occurs when a program writes data outside the fixed boundaries of a buffer, which can lead to serious security breaches. Malicious users can exploit this weakness to modify a program's control flow or execute arbitrary code. This issue often originates from the overlapping storage areas for data (like buffers) and control mechanisms (such as return addresses), where an overflow in data storage can corrupt control flow by altering return addresses.
In this task, students will work with a program that contains a buffer overflow vulnerability. Their objective is to write programs to generate attack string to exploit this vulnerability. Additionally, there is an opportunity for students to earn **bonus** points by successfully exploiting the buffer overflow to execute provided shellcode and launch a shell.

Setup

You are provided with the vulnerable code `vuln_program.c`. You can download this code from Brightspace assignments section.
This program should be compiled with the following command on your CS department machine(remote or not). (Be aware, for the following exploitation, run it on department machine to make it coherent for the grading. Failed to do so may have to schedule an appointment with TA for a code demo).

```
gcc -fno-stack-protector -z execstack -fno-pie -no-pie -m32
    -O0 -g vuln_program.c -o vuln_program
```

The above command will use GNU C compiler to compile the vulnerable code with options(in the order they applied) no stack protector, executable stack, no position independent code, Intel x86 32 bits mode, no optimization applied and GDB debugger information enabled. The output is the vulnerable program `vuln_program`.

Exploitation

The goal of this exploitation is to write your own program and generate an attack string and store it in a file named `attack_string`.
For the base implementation, you are required to launch an exploitation that without providing a correct password for the vulnerable program, the program can still output `"You have entered the correct passwd"`.
To successfully fulfill this task, you have to

1. Compile the the `vuln_program` using the commands provided above.
2. Use a binary analyzing tool to get the address of `target` function. Tools like nm, readelf and objdump can fulfill this job. Read the corresponding manual to figure out how to achieve this.

3. After got the `target` function address, now you have to construct the attack string using your program. And store it in the `attack_string` file. You program should be named as `attack` and take the target function address as the input. For example if 0xdeadbeaf is the target function address, and your code is written in python, then run `"python3 attack.py deadbeef"` will create an string stored in `attack_string` file to be further used to exploit the vulnerability in the vulnerable program.

4. To verify your exploitation, run the program with command `./vuln_program < attack_string` This should give you the expected output `"You have entered the correct passwd"`. If you failed the exploitation, the program usually give you `"Segmentation fault"` error. Then, try to fix the bug in your string generation program, and make it work.

Despite the vulnerable program being written in C, you may use any programming language you are comfortable with to generate the attack string. Make sure the department machines(Lab G7) or remote machines support your language choices. The assignment will be graded on these machines. Contact the TA if your language is not supported by the department machines.

Construct the attack string

The following will try to help you understand how to construct an attack string.

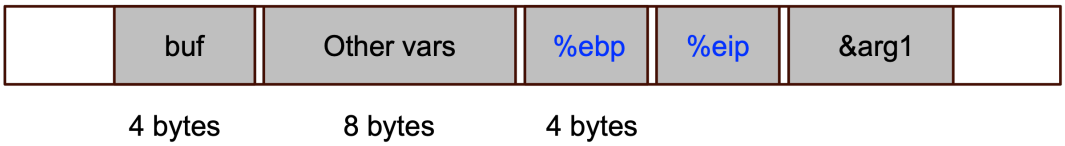

Figure 1: Stack layout of vulnerable program.

Figure1 shows the stack layout while the `prompt` function invoking. The goal is to overwrite the buffer until return address(%eip) on the stack has contains the target function address. Based on this, construct you attack string carefully. One thing to be aware is the address in little-endian format. For example, if target address is "0xdeadbeef", then the bytes of return address at RA will be RA[0]:ef, RA[1]:be, RA[2]:ad, RA[3]:de.

Bonus: Launch a shell (Extra 30 points)

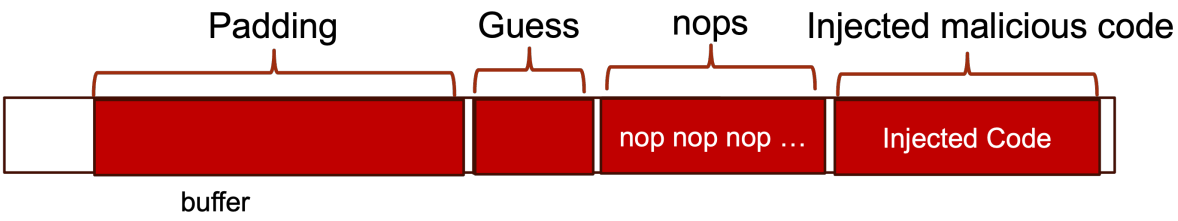

Figure 2: Stack layout of launching shellcode.

As shown in the Figure2, this time you have to overwrite the buffer in a specific way that:

1. Overwrite the buffer with padding.
2. Overwrite the return address(%eip) on the stack with a guessed address that **probably** will jump to the injected malicious code.
3. nops(0x90) can be filled in the between the return address and injected malicious code to increase the chance that injected malicious code will be executed. The nop instruction will do nothing but jump to the instruction.
4. The shellcode then provided as payload at the end of the overwrite.

The shellcode that is used to launch a shell is provided as following:

```
"\x31\xc0\x31\xdb\xb0\x06\xcd\x80\x53\x68/tty\x68/dev\x89\xe3\x31\xc9
\x66\xb9\x12\x27\xb0\x05\xcd\x80\x31\xc0\x50\x68//sh\x68/bin\x89\xe3
\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80"
```

Be aware, due to the space limitation, I have to break this shellcode into 3 lines. If you copy this code into your program, make sure it's a **single line** string, which does not contain any newline characters.

For this task, you are going to reuse the attack program to generate the attack payload for this shellcode exploitation. Provide the argument as "shellcode" to the attack program will generate the shellcode attack payload. For example, if your code is written in python, run your program as `"python3 attack.py shellcode"`. The output of your program should be a file named `"shell_string"` which stores the attack payload for launching the shellcode.

To verify your shellcode payload, you have to run this command:

```
setarch `uname -m` -R ./vuln_program < shell_string
```

The reason we have to do this is the department machines have already enabled address layout randomization(ASLR) protection, which will make the program have a different stack address each time you run it. This can fail your exploitation for most of the time. The above command is a workaround that we can apply to the vulnerable program to disable the ASLR for this running round.

A successful exploit will show you a `$` sign in a new line, which basically launches a new shell program. To get back to the previous shell, run `exit`. A failed exploit will probably give you "Segmentation fault". To debug the payload generated for the vulnerable program, you could use GDB to help you. Launch the vulnerable program with `gdb ./vuln_program`. Then you can add breakpoints inside the program by calling `break`. Run the program with attack payload using `run < shell_string`. An detailed introduction to GDB can be founded here.

Log and submit your work

Log your work: besides the source files of your assignment, you must also include a README file which minimally contains your name, B-number, programming language you used and how to compile/execute your program. Additionally, it can contain the following:

- The status of your program (especially, if not fully complete).
- Implemented the bonus or not. If you implement the bonus, explain why your code work for bonus.

- A description of how your code works, if that is not completely clear by reading the code (note that this should not be necessary, ideally your code should be self-documenting).
- Any other material you believe is relevant to the grading of your project.

Compress the files: compress your README file, all the source files in the same folder, and any additional files you add into a ZIP file. Name the ZIP file based on your BU email ID. For example, if your BU email is "abc@binghamton.edu", then the zip file should be "proj3_abc.zip".

Submission: submit the ZIP file to Brightspace before the deadline.

Grading guidelines

(1) Prepare the ZIP file on a Linux machine. If your zip file cannot be uncompressed, 5 points off.

(2) If the submitted ZIP file/source code files included in the ZIP file are not named as specified above (so that it causes problems for TA's grading scripts), 10 points off.

(3) If the submitted code does not compile or execute:

```
1   TA will try to fix the problem (for no more than 3 minutes);
2   if (problem solved)
3     1%-10% points off (based on how complex the fix is, TA's discretion);
4   else
5     TA may contact the student by email or schedule a demo to fix the problem;
6     if (problem solved)
7       11%-20% points off (based on how complex the fix is, TA's discretion);
8     else
9       All points off;
```

So in the case that TA contacts you to fix a problem, please respond to TA's email promptly or show up at the demo appointment on time; otherwise the line 9 above will be effective.

(4) If the code is not working as required in this spec, the TA should take points based on the assigned full points of the task and the actual problem.

(5) Late Penalty: Day1 10%, Day2 20%, Day3 40%, Day4 60%, Day5 80%

(6) Lastly but not the least, stick to the collaboration policy stated in the syllabus: you may discuss with your fellow students, but code should absolutely be kept private.