Lab 5: Buffer Overflow

50.020 Security

Hand-out: February 28
Hand-in: 9 pm, March 14, 2019

1 Objective

- Familiarize yourself with buffer overflow attacks (on Linux)
- Use GDB/peda to execute buffer overflow attack
- Construct and perform a simple Return-to-LibC attack

2 Background

- A detailed guide for 64 bit stack overflow attacks
  - `https://blog.techorganic.com/2015/04/10/64-bit-linux-stack-smashing-tutorial-part-1/`
- If required, revise C programming language (e.g., character arrays and functions):
  - `http://www.tutorialspoint.com/cprogramming/`
- The attacks should work on most Linux machines, but we only guarantee the lab machines

3 Buffer Overflow - Preparation

- In order to perform the exercise for buffer overflow easier, you need to turn off the randomization (ASLR) of the library address. You may need to do this everytime after you turn on the system.

  ```
  $ sudo bash -c "echo '0' > /proc/sys/kernel/randomize_va_space"
  ```

- If you want to observe core dump for GDB, set the following:

  ```
  $ ulimit -c unlimited
  ```

- You will have to install the `peda` extension for gdb. Follow instructions here for that:
  - `https://github.com/longld/peda`
  - **YOU MUST** also install nasm: `sudo apt-get install nasm`
- Not really mandatory, but the green-on-black terminal color scheme does not work perfectly with gdb/peda. If you want, switch to another theme like "Tango".

4 Warming up: Analysing Stack Frames

- Vulnapp Program:
  - The vulnapp program asks for a user input (until the NUL character is encountered) through standard input and displayed it to the standard output.
  - The function `getlines()` copies the characters from stdin to a character array called `input`. If you want to manually provide input, terminate it with a newline $('\backslash n')$ character (i.e. hit ENTER).
  - To compile vulnapp, run:

    ```
    $ ./make_vulnapp.sh
    ```

  - To run vulnapp:

    ```
    $ echo hello | ./vulnapp
    Please put in the text, terminated with \n character:
    Your text is: hello
    ```

  - Another way to run vulnapp from a text file:

    ```
    $ echo hello > payload
    $ ./vulnapp < payload
    Your text is: hello
    ```

- To analyse vulnapp in GDB, start in the folder containing the binary: `gdb vulnapp`
- Useful commands in GDB:
  - `run ARGUMENTS` (where ARGUMENTS are command line arguments, like `< hello.txt`)
  - `b FUNCNAME` (set breakpoint at function FUNCNAME)
  - `b 43` (set breakpoint at code line 43 of vulnapp.c)
  - `b *0xdeadbeef` (set breakpoint at memory location 0xdeadbeef)
  - `c` (to continue execution after a breakpoint was hit)
  - `disassemble main` (show opcode of main, with indicator of current position)
  - `info stack` (prints the calling stack that lists in which subfunction you are currently)
  - `info frame` (prints information about the current function's stack frame)
  - `x/16x ADDRESS` (to print content of ADDRESS in hex, 16 values),
  - `x/16s ADDRESS` (to print content of ADDRESS interpreted as string, 16 values),
  - `telescope` smart visualization of memory contents, (similar to `x`, but dereferences pointers, etc)
  - `s` or `si` (to single step through c or assembly code)
  - `help FOO` (to get help on command FOO)
- Use `checksec` to learn about enabled/disabled security features
- Use `disass main` to get a disassembly of the main() function
- Analyse the stack before and after calling `getlines` from main().

  - To analyse the stack during the program flow, you must set appropriate breakpoints and run the program.
  - By using the command `info stack` and `info frame` you can examine the stack and the stack frame respectively.
  - Can you see where your stdin data gets written on the main() stack frame?
  - For your writeup, please provide output of `info frame` and `telescope` (with suitable offsets and length) to show the contents of the stack at both times.

- Try a large input of around 100 characters - does the programm execute successfully? What does this mean?

5 Buffer Overflow - Shellcode

- In this part, we are going to create a payload that injects some custom assembly code into our vulnapp program
  - Typically, such code is used to get a shell, and is also called *shellcode*
- The following commands may be helpful for you in this exercise:

  `info frame, pattern_offset, pattern_search.`

5.1 Find out how to overwrite main() return pointer

- Start by finding out which input exactly overwrites the stored return address of main()
  - Use peda to generate a pattern string to use as input to vulnapp.

```
gdb-peda$ pattern create 100 payload
r < payload
```

- Vary the length of the characters in your payload, and **find the location of the return pointer in the main() stack frame**.
- Now you should have an idea how many characters you can input before overwriting the RIP saved on the stack.
- Time to create a custom file called `payload` that will contain your exploit string. We provide a suitable skeleton file.
- An execution of `r < payloadgdb` in gdb should now result in a SIGSEV, and the return address on stack containing 'EEEEEEEE'

5.2 Add bogus RIP

- So far, your payload should only overwrite the stored RIP with 'EEEEEEEE'. Now, we want to execute our own code instead.
- In addition to the current payload, we will have to add the shellcode as well

- We have prepared the `payload.py` file for you, which contains python code with a shellcode string. Extend that file to construct and write the full `payload` file.
- The shellcode should come after your characters that overwrite stored RIP. That way, it will not be modified by main().
- Now we will put in the right return address into `payload.py`
  - Recall that you have overwritten the stored RIP with 'EEEEEEEE'. Now, change the 'EEEEEEEE' characters with the address of your shellcode.
- Use GDB to find the memory address where the payload instructions are placed on the stack. Use that address in `payload.py` to replace the 'EEEEEEEE' string. Run the program with the payload inside GDB. You should see a "Hello World!"
- Try again on the shell. Most likely, it will not work right away wity the return address you found in GDB, because there are address offsets when run within GDB. The `payload.py` script is set up to produce two versions of the payload, one for inside GDB, and the other one for outside.

1. Debugging using core files
   - Use GDB to analyse cores that are dumped by a crashing application

   ```
   gdb ./vulnapp core
   ```

   - You should then be able to explore the stack and other info at the time the program crashed
   - Hint: Make sure ASLR is still deactivated, and delete cores before generating new ones
   - Hint: `info frame 0` will show you info on the stack frame at the point of crash. You can inspect registers with `info registers`. Remember that your shellcode is located just above the stored RIP.

5.3 Remote debugging

- Open another terminal, find the process number of vulnapp

  ```
  $ ps -ef | grep vulnapp
  user       4093  2758  0 12:57 pts/10   00:00:00 ./vulnappROP
  ```

  The process number is the second column
- Start gdb with the `-p` argument to attach to the running process

  ```
  $ sudo gdb -p 4093
  ```

- Now you should be able to inspect the memory of the running process.

6 Buffer Overflow - Return To Libc

- In this part, we use the `vulnappROP` binary, which has NX enabled. Your previous exploit should not work any longer, as we cannot execute our shellcode directly.
- We can overcome non-executable stack by performing a return-to-libc attack. The idea is that we can find libc functions somewhere in the memory address and use those functions to run our code. For this exercise, we will find the address for printf function and use it to print 'Hello, world!'. To print this string, printf function only needs one parameter, which is the string to print.
- To perform return-to-libc attack on a 64-bit machine, we need to do the following:
  - Pop the address of the string to print from the stack into rdi register.
  - Provide the address of printf function as a return pointer.
- In order to pop values from the stack, we are going to use 'gadgets'. Gadgets are small assembly codes that we find somewhere in the memory, which in this case will be used to pop values from the stack to the register and return to the original instruction set. In this exercise, we only need one gadget since printf only takes one parameter.

  ```
  pop %rdi
  retq
  ```

  The hex code for `pop %rdi` is 5f, while the hex code for `retq` is 3c.
- Peda makes it very easy to find the address of this gadget

  ```
  gdb-peda$ ropsearch "pop rdi" libc
  ```

  - Pick one of the gadgets' address.
- Now, we want to find the address for printf function.

  ```
  gdb-peda$   p printf
  ```

  - We can do the same to obtain the address for exit system function. This will make the program exit nicely after calling printf.
- We need to provide the address of the string into our payload. To do this, we can inject our own string as before (e.g. before or after the 4 main addresses we will inject). Use either core files, or an attached `gdb` session to find those addresses for `vulnapp` running outside `gdb`.
- Now we need to create the payload that fill the stack properly as shown below.

  ```
  ---------------------- low memory address
  |   input XY bytes   |
  |   "Hello world"    |
  |  if you want here  |
  ```

```
|--------------------|
| rip:gadget address |
|--------------------|
|   string address   |
|--------------------|
|   printf address   |
|--------------------|
|   exit address     |
|--------------------| high memory address
```

- Generate the payload and feed in to vulnappROP.
- Make sure that your attack works in GDB, and outside GDB

6.1 Extra: spawn a shell (not mandatory, not graded)

- If you want, you can edit both code injection and return-to-libc to get a shell. For that, you either have to replace the shellcode, or jump into `system()` instead of `printf()`.
- Use the following command to get your payload into the program to avoid closing stdin after file end

```
cat payload - | ./vulnapp
```

7 Hand-in

- Hand in the following
  1. A heavily commented script for Section 5 to generate the first attack payload with your own injected shellcode
     - Mention how you found the right parameters, and the addresses
  2. A heavily commented script for Section 6 to generate the second attack payload with the ROP
     - Mention how you found the right parameters, and the addresses
- Make sure to put your name/ID in the header of the script