Exercise 6

Return-to-libc and Return Oriented Programming (ROP)

Log into your VM (`user` / `1234`), open a terminal and type in `infosec pull 6`.

- When prompted, enter your username and password.
- Once the command completes, your exercise should be ready at `/home/user/6/`.

When you finish solving the assignment, submit your exercise with `infosec push 6`.

NOTE: After solving `q1a.py` of the questions, you will get permission errors when submitting, unless you read and follow the instructions at the end of this exercise. Please read them carefully.

Motivation (and some history)

Most buffer overflow attacks we exploited so far, rely on writing our code into the stack and then hijacking the flow of the program to execute our code. After this sort of attacks became popular, some security measures were taken to make it harder to carry out these attacks.

During 2000-2005, many operating systems began implementing various protections on the executable space, typically by marking the stack (and/or other areas of the memory) as non-executable. This prevented the classic buffer overflow attack we exploited so far, and presented a new challenge for attackers.

In 2006, an attack called return-to-libc was published, describing a mechanism that enables obtaining a shell in some cases, even with a non-executable stack. We'll implement this technique in the first part of the homework.

In 2007, a significantly improved version of this attack was published under the name return oriented programming (ROP). This was presented in Blackhat 2008, and enabled executing far more sophisticated codes that return-to-libc. We'll implement this technique in the second part of the homework.

Our target, as in exercise 4 is the **sudo** program. It came for seconds, this time with a non-executable stack 😈.

Inspecting `libc` memory with GDB

Before starting to solve this question, we'll need to understand how to inspect the memory of `libc` (the C standard library). To do this, follow these instructions:

- Load sudo in GDB - `gdb ./sudo`
- To view the mapping in memory of the various section of our binary and libraries, run `info files`
  - Since no library was loaded yet (the program is not running yet), you will only see sections of the binary itself, without information about the libraries
- Set a breakpoint on the main function – `break main`
- Run the program (`run`).
  - Obviously, it will immediately stop and hit the breakpoint.
  - But, this time the libraries are loaded since we ran the program.
- Again, run `info files` and look for sections of `(…)libc.so(.6)`
  - The `.text` section is where the assembly code resides. **This section of the memory is executable**.
  - The `.rodata` section is where initialized constant data (such as strings) is stored
- To dump the binary contents of the memory at a certain range, use `dump binary memory {path_to_dump_file} {start_addr} {end_addr}`
  - You'll need this later :)

Question 1 (20 pt)

In this question, we'll implement the return-to-libc attack, to open a shell with root privileges. Use both the source code, and IDA to find a vulnerability in the `sudo` program, that will allow you to override the return address.

1. Inside `q1a.py` script, implement the `get_crash_arg` function - the function will return a password, that when sent to the `sudo` program it will make it crash and generate a core dump.
   a. Describe your solution in `q1a.txt`.
2. Inside the `q1b.py` script, implement the `get_arg` function - the function will return a password, that when sent to the `sudo` program it will make it **open a shell using the return-to-libc technique**. Specifically, the invocation you want to make is `system("/bin/sh")`.
   a. The program is allowed to crash after you exit the shell – this is OK.
   b. Since `system` is not in the PLT, use GDB to find the real address of `system` which will be loaded at runtime (as part of libc).

i. **Update** the address of the `system` in `addresses.py`

c. Search the `.rodata` section of libc for an occurrence of `"/bin/sh"`. To do this, you can use `find {start_addr}, {end_addr}, {string}` in GDB.

i. **Update** the address of the string in `addresses.py`

d. It's OK if the opened shell includes `\[\033[33m\]` and other similar control characters at the line of the command itself.

e. Describe your solution in **`q1b.txt`**.

3. Now we would like to not only open a shell, but also to make sure the program always exits with a status code of `0x42` (`66`) instead of crashing with a segfault. To do this, we want to first call `system("/bin/sh")` and then call `exit(0x42)` - we'll perform both calls as part of the return-to-libc attack.

a. Since `exit` is not in the PLT, use GDB to find it's address as you did for `system`.

i. **Update** the address of `exit` in `addresses.py`

b. Inside the **`q1c.py`** script, implement the `get_arg` function.

c. To check the exit code of your binary, you can call `echo $?` in the same shell right after it exited (call this right after calling `python q1c.py`).

d. Describe your solution in **`q1c.txt`**.

Question 2 (20 pt)

In the next questions, we're going to implement ROP attacks! However, implementing a ROP requires having a working "gadget search engine" that can search the memory and locate gadgets!

The search engine we'll implement will be fancier than what you might expect at first - it will support searching for the same instruction with multiple combinations of registers at once, so that we don't have to try all combinations manually. For example:

```
s = search.GadgetSearch('./libc.bin')

s.find('POP EAX')
=> 0x123456789 # Address of 'POP EAX; RET'

s.find_format('POP {0}; POP {1}')
=> ('POP ecx; POP edx', 0x98765432) # Address of a matching gadget
                                    # (including a RET at the end)
```

1. Using the techniques mentioned above, use GDB to create a dump of the `.text` section of libc. Call this file `libc.bin` and save it in your exercise directory for submission.
2. Implement all the non-implemented functions in `search.py` (i.e., all the functions that currently raise a `NotImplementedError`)
    a. Full details available inside `search.py`.
    b. For clarification (in case of doubt) - `find_all_formats` finds all the addresses of a gadget (which means that if it appears more than once, return all occurences)

Question 3 (20 pt)

In this question, we'll use ROP to create a basic "write gadget". A write gadget is a gadget that receives a memory address and a (4-byte) value, and writes the value into the given memory address - similar to (`MOV [addr], value`).

We will use this gadget to override the `auth` variable of the **sudo** program, to make it print `Victory!`

1. First, find all the required components:
    a. Using the search engine from above, find a combination of instructions (one or more) that allows you to write a custom value to any memory address.
    b. Use IDA/GDB to find the address of the `auth` variable.
    c. Use IDA/GDB to find the original return address from the `check_password` function.
2. Inside the `q3.py` script, implement the `get_arg` function so that the argument it returns will make the **sudo** program execute the write gadget to override the `auth` variable and print `Victory!`
    a. Find the address of the `auth` variable and **update** it in `addresses.py`
    b. Note that after the gadget to modify auth, the next return address should be the original return address you overrode - so that the program continues and we can see the new `auth` value is being used.
    c. **The program is allowed to crash after the printing** – this is OK.
    d. For each of the addresses you found with the gadget search engine, call the gadget search engine explicitly from the python code of `q3.py`.
        i. **i.e. don't just hard code gadget <u>addresses</u> in your code.**
        ii. It's OK to hard code the gadget instructions (such as `"POP EBP"`), but don't hard code the result address from this search
    e. Describe your solution in `q3.txt`.

Question 4 (40 pt)

In this final question, you'll implement a ROP that will cause the `sudo` program to run in an endless loop and print the string `"Take me (<YOUR_ID>) to your leader!"`. This means that the ROP you want to create, if your ID is 123456789, is the equivalent of:

```
while (1) {
  puts("Take me (123456789) to your leader!");
}
```

Your ROP should be constructed roughly as follows:

1. Load the address of `puts` into `EBP`
2. Jump to `puts`
3. Address of a gadget to "skip" 4 bytes on the stack
4. Address of your string
5. Loop back to the second step (2 - Jump to `puts`)
    a. You can assume the stack is always in the same address, so you can know where on the stack the loop begins

Notes:

- Find the address of the `puts` function and **update** it in `addresses.py`
- As before, inside the `q4.py` script, implement the `get_arg` function so that the argument it returns will create the desired ROP when passed to `sudo`.
- For each of the addresses you found with the gadget search engine, call the gadget search engine explicitly from the python code of `q4.py`.
    a. **As before - don't just hard code gadget addresses in your code.**
- Document your solution in `q4.txt`. In your description, you MUST address the following issues (in addition to the rest of the explanation):
    a. Explain why the loop works after we call `puts`:
        - Theoretically, when we call `puts`, the space `puts` allocates on the stack should overwrite the parts of the ROP in lower addresses (parts that happened before `puts`).
        - Specifically, this means it should be overriding its own address on the stack!
        - However, **our loop works – explain why!**
            - Hint: Look at the disassembly of the `puts` function itself.

    b. The next instruction in the ROP after `puts`, is skipping 4 bytes on the stack. **Explain why** this 4-byte skip is necessary.

c. Explain where and how did you include your string in the ROP.

Final notes:

- IMPORTANT: Your submission is going to (automatically) include a core dump in q1a, and the core dump is "owned" by root, so we can't access it directly. This means you'll have to do the submission as root for submission to work: `sudo infosec push 6`
    - **Use the real sudo**, not the sudo inside the exercise directory
- Document your code (and your shellcode!).
- We added a place for documenting all the addresses you use in `addresses.py` - always use addresses from that file and don't hard-code addresses in the code, so that we can test your code with other addresses.
    - Document addresses as numbers (such as 0x80841325), not strings or anything like that.
        i. To convert the addresses to bytes in little endian, there's the `address_to_bytes` function
    - If you use any other addresses/offsets/magic values, document what they are so we can understand your solution.
        i. Preferrably document these also in `addresses.py` (by adding new values in the file)
    - We want to give you a great grade, and we can't forgive mistakes when we don't know where the numbers came from :/
- Don't use any additional third party libraries that aren't already installed on your machine (i.e. don't install anything).
- While some code answers will be longer in this exercise than in previous ones, none of them should be more than roughly ~100 lines. If it's way more than that, it may be that you picked the won't solution strategy.
- Each student is getting slightly different binaries for this question - don't be surprised if your solution is different from those of other students.