Exercise 5

Reverse Shell and Polymorphic Shellcodes / `ex-remote-shell`

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

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

When you finish solving the assignment, submit your exercise with `infosec push ex-remote-shell`.

Motivation

In the previous exercise, we exploited a vulnerability in a local process, to gain a privilege escalation - i.e. run with higher permissions. While this sort of attack is useful, typically we wish to attack remote machines.

In this exercise you'll learn how to exploit a vulnerability on a remote machine (to achieve "Remote Code Execution" (RCE)), to make it connect back to you with a shell and wait for further instructions ("remote shell").

Background

In this exercise, we'll attack a simple log server - a server that receives messages over the network and logs them to a file.

- The server listens for incoming connections on a socket
- From each connection it then reads 4 bytes for the length of the message, followed by the bytes of the string with the actual message
  - The message length is as specified in the first 4 bytes[1]
  - The message must be `\0` terminated
  - **The `\0` is counted towards the message length**
- The server program is located at `./server/server`
- The source code of the server (which will help you in attacking it) can be found at `./server/server.c`

---

[1] The bytes are in "big endian" as that's the convention for sending data over the network (also known as "network order")

For your convenience, we attached an example client to communicate with the server. Using the server and client can be done as follows:

- Launch the server program (located at `./server/server`)

```
/home/user/ex-remote-shell$ ./server/server
Opened /tmp/log.txt.
Listening on 0.0.0.0:8000.
```

- In another terminal, send a message to the server using the included client program (which you can use with `./server/client.py <msg>`)

```
/home/user/ex-remote-shell$ python3 ./server/client.py
'Veni, vidi, vici'
```

- The server should now indicate that the message was received:

```
/home/user/ex-remote-shell$ ./server/server
Opened /tmp/log.txt.
Listening on 0.0.0.0:8000.
Connection accepted.
Message logged successfully.
```

Question 1 (10 pt)

First, to prove there is actually a vulnerability in the server, we wish to find a message to send to the server so that it would crash.

1. Use the source code and/or IDA to find a vulnerability in the `server` program.
2. Inside the `q1.py` script, implement the `get_payload` function - the function returns the data to be sent over the socket to crash the `server` program.
    a. The `main` function will already open the socket for you. Do NOT open the socket by yourselves(!).
3. Describe the vulnerability and your solution in `q1.txt`.

Remote Shell

As mentioned in class, while the attack we wish to carry is more or less the same (running `exec` with `"/bin/sh"`), opening a shell on a remote machine is pretty "useless if we can't control it". To use our shell, we'll need to:

1. Make the server open a socket to our C&C server
2. Redirect `STDIN`, `STDOUT` and `STDERR` to the socket
3. Only after these, run `exec` with `/bin/sh`

C&C servers

Now to the next question - where do we get a server for our C&C?

For this exercise, we will use the **netcat** (`nc`) utility - a program included with standard Linux distributions, which opens a socket and:

- Prints to `STDOUT` what it receives from the socket
- Reads from `STDIN` and sends that to the socket

To learn how to use `nc`, try the following:

- Open a terminal and run `nc -v -l 1337` to make `nc` listen for incoming connections on port 1337
- In another (new) terminal, run `nc -v 127.0.0.1 1337` to make `nc` connect to port 1337 (the address where the <u>other</u> `nc` listens on)
    - **Tip**: Working with multiple terminal windows is kind of annoying. If you are using the course VM, use **Ctrl+Shift+T** to open another tab in the same window, **Ctrl+Shift+O** to split the window horizontally, or **Ctrl+Shift+E** to split it vertically
- Now try typing in the terminal of any of the `nc`'s - everything you type in one terminal, will be sent to the other (after you hit `Enter↵`)
- Close one of the `nc` programs (you can kill a running program with **Ctrl+C** in it's terminal). This will cause the connection to terminate, which **will stop the `nc` on the other terminal as well**.

Once you understand how to use `nc`, you can proceed to the next questions.

Question 2 (45 pt)

In this question you will open a remote shell, using the technique described above. Your shellcode should connect to a C&C server listening at `127.0.0.1` on port `1337`, redirect `STDIN`, `STDOUT` and `STDERR` to the socket, and finally execute `/bin/sh`.

When done, it should look like so:

```
~/ex-remote-shell$ ./server/server
Opened /tmp/log.txt.
Listening on 0.0.0.0:8000.
```

```
~/ex-remote-shell$ nc -l 1337 -v
Listening on [0.0.0.0] (family 0, port 1337)
```

With both shells open, run `python3 q2.py` (in yet another terminal) - if successful, it should make the `server` program connect to your C&C server - which will look like so:

```
Connection accepted.
Message logged successfully.
```

```
Connection from [127.0.0.1] port 2562 [tcp/*]
```

Then, verify the shell works by typing commands as input to `nc` as if it was a shell. The output of the commands should reflect on the output of `nc`, and **nothing should be visible on the output of the server**.

```
echo "I am `whoami`"
I am user
pwd
/home/user/ex-remote-shell
```

1. Write your shellcode in **`shellcode.asm`**
2. Inside the **`q2.py`** script, implement the `get_shellcode` function, which **returns the assembled shellcode**.
   a. Further instructions are available inside the python file itself - please read and follow them
3. Also inside the **`q2.py`** script, implement the `get_payload` function, which returns the data to be sent over the socket to the **`server`** program to exploit the vulnerability
   a. The payload is made of the 4 byte message size, the shellcode, a <u>NOP slide</u> before the shellcode, return address, etc.
4. As usual, document your solution in **`q2.txt`**.

Hints:

- All the functions you need for the shellcode are in the PLT - **you don't need to use syscalls**. The functions we made available for you can be seen inside the code of the `dummy` function at **`./server/server.c`**.
- **When you open your shell, it replaces the original server process.**
   a. I.e. - <u>you don't need to keep the original server alive</u> (via fork() or other similar tricks).
- To open a socket, you will need to manipulate structs, and to do that you will need to see how to work with the structs related to sockets. To get a reference for that:
   a. Write a C program that does exactly what you want the shellcode to do (i.e. connect, redirect input and output, then do exec)
      - If you don't have much experience in socket programming, don't worry - we got you covered.
      - <u>Here</u> is an explanation on the various functions to handle sockets, and <u>here</u> is a short example in C for opening a socket.
   b. Setup a C&C server using **`nc -v -l 1337`**
   c. Verify the program works by running it to connect to the C&C, and see you can run commands from the **`nc`** shell and see their outputs there

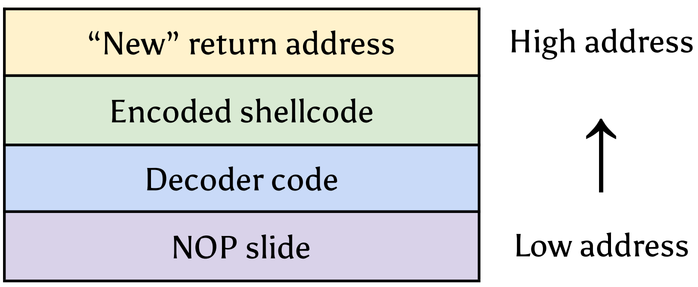
d. Finally, when you verified everything works, **inspect the assembly of your program to mimic how to work with sockets**

- `./server/server` was compiled <u>without</u> “debugging symbols” - meaning that you can’t debug it in GDB with references to the source code - you can only debug it on the assembly level.
    a. If you wish to debug something similar that does have a reference to lines in the source code (**this is optional**), you can recompile your own copy of the server by running
    `gcc ./server.c -masm=intel -fno-pic -fno-pie -fno-stack-protector -z execstack -o ./server_debug`
    b. Note that after you make your code work with your compiled server - you still need to make sure it works with ours (some things will be slightly different due to compiler differences)

Question 3 (45 pt)

Now that you understood how to open a remote shell, let’s do it again under one constraint - the entire shellcode is going to be in ASCII. This means that except for the first 4 bytes (message length) and last 4 bytes[2] (new return address), all other bytes should be valid ASCII - i.e. with values lower than `0x80` (at most `0x7f`).

Since writing an ASCII shellcode directly is very cumbersome, what you really want to do is to use the shellcode from the previous question; you will “encode” it to ASCII chars and then you will dynamically “decode” it at runtime. Specifically, we’ll put the decoder directly before the encoded shellcode, so when it finishes, the next instruction will simply be the (decoded) shellcode. Here’s how it will look like:



The method we will use[3] for encoding/decoding is as follows:

- The encoding will XOR every non ASCII byte (i.e. >=`0x80`) with `0xff`
- The decoder is going to be a series of `XOR` instructions, to decode the memory in the XOR-red locations

[2] Or 5 bytes if you also count the terminating \0 in the message
[3] There are many options for writing an encoder/decoder, we chose the one described because it’s simplistic and allows understanding the general idea

    - While encoding, you will keep track of the indices that were XOR-ed
    - You will then use the same indices to generate the decoder code
- Note that our encoding keeps the shellcode at the same length
    - The decoder grows in size as more bytes are XOR-ed, but the encoded data stays in the same length as the original data

Now, let's implement the code for this question inside the `q3.py` script:

1. Implement the `encode(data)` function. The function receives data, and encodes it to be valid ASCII by XOR-ing each non-ASCII byte (i.e. bytes larger than `0x7f`) XOR-ed with `0xff`. The function returns a pair of values, where:
    - The first value is the encoded data (after XOR-ing each non-ASCII byte)
    - The second value is an array, with the list of all indices of bytes that were XOR-ed
2. Implement the `get_decoder_code(indices)` function. The function will generate decoding code, to XOR bytes at all specified indices, relative to `EAX` as the base address, with `0xff`. I.e., your function will generate code similar[4] to:
   ```
   XOR byte ptr [EAX + i1], 0xff
   XOR byte ptr [EAX + i2], 0xff
   ...
   ```
   Where `i1` and `i2` are indices from the array. Notes:
    - The function will return the **assembled code**
    - The returned code must be valid ASCII
    - You can't use `0xff` directly since `0xff` itself is not a valid ASCII byte. Use some trick to get this value into `BL` (the lower 8 bits of `EBX`) or any other register, and then XOR with `BL` (or the other register you chose)
    - You may use/modify any register you need
    - Note that we are running a smoke test on this function (look at `smoketest.py`) with some random indices, to see it is indeed generic and will work with other shellcodes
        i. If you get a smoke test failure in `get_decoder` even though you think everything is fine - this may be the problem. Look at `smoketest.py` for details on how to reproduce
        ii. Make sure it will indeed work with more indices that make the smoke test pass
3. Implement the `get_ascii_shellcode()` function. The function will return **assembled code**, of an **encoded shellcode** prefixed with a **decoder**:
    - Use `get_raw_shellcode()` to get the shellcode from question 2, and then encode it using your `encode(data)` function

[4] Your code doesn't have to look exactly like this. You can use any set of instructions that achieves the same effect

- In this decoder-based shellcode, you'll need to get the address of the encoded shellcode into `EAX`.
    i. Note that you can't use the call trick[5] as the opcode for `CALL` is non-ascii[6].
    ii. Instead, remember that when your code starts, `ESP` is right after the return address. This means that the offset between the beginning of the **encoded shellcode** and `ESP` is constant!
    iii. Use math to figure out the start address from `ESP`, and then store the result in `EAX`

4. Finally, implement the `get_payload()` function, which returns the data to be sent over the socket to the **`server`** program to exploit the vulnerability
    - The payload is made of the 4 byte message size, a NOP slide before the ascii decoder shellcode, the shellcode, return address, etc.
    - Note that your NOP slide cannot use `NOP` since the opcode for NOP is non ASCII… Find another x86 opcode that is a single byte, and that we don't care if executes, and use that instead of `NOP`

Final notes:

- Document your code (and your shellcode!).
- Explicitly write all addresses you use - addresses in the PLT, addresses of buffers, offsets, and so on. 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 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 wrong 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.

[5] As we did in the regular shellcode to get the address of the string with `"/bin/sh"`

[6] Furthermore, even if you could, you would need a `CALL`/`JMP` with a negative offset, and negative numbers have the most significant bit on, which means the most significant byte would be >= `0x80`