System Security
Bot Analysis

Luca Di Bartolomeo

December 6, 2018

In this exercise you will reverse engineer a bot. Bots that are part of a botnet can be controlled by a botmaster in various ways. In this case the botmaster uses a chatroom to communicate with one or more bots. Your task is to find the commands that the botmaster can use and that the bot understands.
You should run the bot inside the provided VM. This way things will work as expected and you are protected from possible vulnerabilities of the bot. The VM also contains useful tools for the analysis, such as gdb and Cutter.

1. Run `setup.sh`. Provide the syssec password when requested.
2. Start the chatroom by running `python2 chatroom.py`. By default the chatroom will listen for local connections on port 4567. It simply forwards all messages it receives to all connected parties.
3. Run the chatclient using `python2 chatclient.py`. The client will be your interface to the chatroom. It allows you to send commands and receive output.
4. Start the bot, e.g. by running `./bot 127.0.0.1 4567`. This way the bot will connect to the chatroom. You should see a greeting message from the bot in your chat client.
5. You can run multiple clients and bots.

Note: You are not allowed to answer the questions by referring to the python scripts.
Reverse Engineering can be done in different ways. Among variations there are black- and white-box approaches. In black-box approaches the externally observable data is used. It often provides a good start. Good tools for black-box analysis are `strings`, `strace`, `lsof`, `ps`, `netstat` or `wireshark`. Especially `strings` might be useful for you as it extracts readable strings from the executable.

How does the bot communicate with the chat room? Using TCP or UDP? Is the transmission encrypted? How did you find these answers? (1 point)

Solution:

The bot communicates with the chat room through TCP sockets. The transmission is not encrypted.
I found out those things by looking directly with wireshark at the traffic generated by the bot.

White-box approaches use introspection of the program, e.g. using a debugger. However, this can be a very time-consuming task. As your task is to identify the bot commands you do not have to analyse the whole program. Think about where the received commands will be handled. If you are unfamiliar with network sockets check which function is used to receive the data [1].
You can either set a breakpoint directly after the data is received or attach to the process when the bot is waiting for new commands and then continue stepping through the program to observe the data handling.

Which system call is used to receive possible commands? Where is the code called (e.g., `0x08040000`)? (1 point)

Solution:

The system call used is the `read()`, called at address `0x0804969d`. A maximum of 0x1000 bytes are read, and are placed on a buffer on the stack.

Now you have to understand which commands are accepted by the bot. Check how the commands are filtered and parsed. Once you think you found a command try it and observe its functionality.

Which commands does the bot accept? List their names, their functionality, describe the output. If external files, such as images are used, include those files into your report. You should understand at least four commands. (2 points for each command)

Solution:

- **.info**:
  Prints information about the system in a maximum of 0x1000 bytes. The information printed is: "USER" env variable, hostname, the first field of the struct `utsname` (which is the name of the implementation of the operating system), the third field of the same struct (which is the release level of the implementation of the operating system), and the fifth field of the same struct (which is the hardware type the system is running on.

  In other words, this command basically executes: `snprintf(fd, 0x1000, "%s %s %s %s %s", env("USER"), hostname, uname->sysname, uname->release, uname->machine);`.

  See "sys/utsname.h" for reference about the struct utsname.

- **.processes**:
  Prints the output of the command `ps -a` by doing a call to `popen()` and using then `fgets()` to read the output of the command.

[1] `https://en.wikipedia.org/wiki/Berkeley_sockets`

- **.flash**:
  This command opens file `/tmp/tmp_bot_image.png`, writes an image one char at a time using function `putc()`, and then runs this command with `system()`:

  ```
  zsh -c \"qiv -i -f /tmp/tmp_bot_image.png &; sleep .4;
  pkill qiv; rm /tmp/tmp_bot_image.png\"
  ```

  This command displays the image for 0.4 seconds, and then deletes it.

  The image is the following:

  

- **.kill**:
  This command will make the bot print "One will fall but others will rise. Hasta la vista."

  Then, the bot will close the sockets it has opened and terminate.

- **.fight**:
  The bot will try to parse two bot names, using `sscanf(s, "bot-%i bot-%i", &a, &b`.

  If both number `a` or `b` are not the bot's own pid, it will print "I will just watch" and do nothing else.

  Otherwise, a "fight" will start. The bot will print "*Two bots enter, one bot leaves*".

  The bot will then generate a random number. The bot will then write its generated number to the UNIX socket of the adversary bot.

  Afterwards, the bot will keep reading from its corrisponding UNIX socket, until it reads a number. If the number it reads is bigger than the number it generated, it will lose, print "*I lose, oh nooooooooooo...*", and terminate. Otherwise, if the number it reads is smaller than the number it generated, it will "win", print "*I win, I live...*", and wait for other commands. If the two numbers are equal, it will print "*A tie, how neat.*", and wait for other commands.

There are certain commands, that do not show up in the output of `strings`. Why? (1 point)

Solution:

The command that do not show up in the output of `strings` are ".secret" and ".e4stere66!1". They do not show up because, contrary to other commands, they are not parsed with a `strcmp()`.
The ".secret" command is parsed letter by letter, and the ".e4stere66!1" command is parsed letter by letter too, but also stored encrypted with `xor` key 0x81.

Which kind of messages are filtered out first by the bot? Can you imagine why? (1 point)

Solution:

All messages which start with "bot" are filtered out by the bot. This is because bots sometimes answer to commands, and each bot broadcasts the answer to everyone else in the chat. So in this way, only messages sent by humans are parsed by the bot.

How does a bot generate the name it uses in the chatroom? (1 point)

Solution:

The bot generates its name by appending its pid (obtained by a call to `getpid()` at address 0x80493e7) to the string "bot-".

Which other commands can you find? Describe how you found them, how the bot parses them, and their functionality. You can find up to seven total commands. (2 points for each command)

Solution:

- **.secret**:
  This command will change the background of the desktop to a random color chosen between four different ones.

  It does that by calling the `system()` function with the following string as argument:

  `dconf write /org/mate/desktop/background/primary-color "'#ff33cc'"`

  The bot also prints "*Do you see a change? You may have to try a few times...*"

  This string was kept in memory encrypted through `xor` with key 0x81.

- **.e4stere66!1**:
  This command will print the following:

  "*Congrats, you found the easteregg. Document how you found it and the command for extra points*".

  I found it by reversing `func2`. I saw that it checked if a particular command was equal to some bytes in memory encrypted with `xor`. Then, I manually decrypted each byte using python and converted them to ASCII.

The bots can enter into some form of "conflict". How do they communicate? What is therefore required for two bots to communicate? How do they find a winner? (2 points)

Solution:

By using the command ".fight [bot_a] [bot_b]" where bot_a and bot_b are the names of two bots.

The bots will try to parse the two bot names, using `sscanf(s, "bot-%i bot-%i", &a, &b`, where "s" is the buffer with the contents read from the socket connected to the server.

If both number `a` or `b` are not the bot's own pid, it will print "I will just watch" and do nothing else.

Otherwise, a "fight" will start. The bots will print "*Two bots enter, one bot leaves*".

The bots will then generate a random number. The bots will then write its generated number to the UNIX socket of the adversary bot.

Afterwards, the bots will keep reading from their corrisponding UNIX socket, until they reads a number. If the number they read is bigger than the number they generated, they will lose, print "*I lose, oh nooooooooooo...*", and terminate. Otherwise, if the number they read is smaller than the number they generated, they will "win", print "*I win, I live...*", and wait for other commands. If the two numbers are equal, it will print "*A tie, how neat.*", and wait for other commands.

In the rare event that a bot receives a number from its UNIX socket but it hasn't generated a number itself, it will print "*Got attacked when unprepared...*" and continue normally.

Notes

- Normally symbols, such as function names would be stripped out. We left them in to make this exercise a bit easier.
- Pasting into the chatclient might not work correctly. Try to add a space after pasting.
- When using gdb it can be helpful to use the `layout asm` command.
- To understand the advanced features, it helps to know about select[2].

[2] `https://en.wikipedia.org/wiki/Select_%28Unix%29`

- Normally only the bot executable would run inside the VM, while the chatroom and chatclient would run on different machines. However, for this exercise it is fine to run all inside the VM. If you want to run the chatroom and chatclient outside the VM, you need to adjust the command line parameters of bot and possibly chatclient (see `python2 chatclient.py --help`).
- You can find a maximum of seven commands.

References

[1] Please cite your sources, Example Author, `http://www.example.org`