System Security, Solution Bot Analysis

George Cojocar

December 1, 2015

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.

We recommend that you 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 folder `bot` contains the bot's executable.

1. 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.

2. 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.

3. 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.

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 encoded or encrypted? How did you find these answers?

Solution:

The bot communicates with the chat room using TCP. The chat room forwards all messages it receive to the bot. The bot replies back to the port `4567` of the chat room.

```
% netstat -ap | grep bot
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
tcp        0      0 localhost.localdo:47170 localhost.localdom:
   ↪ tram ESTABLISHED 3874/./bot
```

Using Wireshark one can verify if the transmission is encoded or encrypted by sending a text message from chat client. The network traffic captured by Wireshark for text "test" is the following:

```
chat client sends "test" -> chat room
47169->4567
...
Data (4 bytes)

0000  74 65 73 74      test
    Data: 74657374
    [Length: 4]

char room forwards "test" -> bot
4567->47170
...
Data (4 bytes)

0000  74 65 73 74     test
    Data: 74657374
    [Length: 4]

boot sends "bot-3874: Huh?" -> chat room
47170->4567

...
Data (14 bytes)

0000  62 6f 74 2d 33 38 37 34 3a 20 48 75 68 3f          bot
   ↪ -3874: Huh?
    Data: 626f742d333837343a204875683f
    [Length: 14]

chat room forwards "bot-3874: Huh?" -> chat client
4567->47169
...
Data (14 bytes)

0000  62 6f 74 2d 33 38 37 34 3a 20 48 75 68 3f          bot
   ↪ -3874: Huh?
    Data: 626f742d333837343a204875683f
    [Length: 14]
```

It appears that the messages are sent in clear text.

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 function is used to receive possible commands? What is its address in the code, e.g. `0x08040000`?

Solution:

The bot uses the `select` function to wait for incoming data on the socket descriptor opened with the chat room. Its address in the code is `0x08049535`.

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 and provide possibly used files. You should understand at least four commands.

Solution:

- **.info**

  Functionality:

  It collects more information about the system by executing the following system calls:

  - **gethostname**: to access the host name ("syssecvm")
  - **uname**: to gather information such as operating system name ("Linux"), operating system release ("4.1.6-1-ARCH") and hardware identifier ("i686")
  - **getenv**(USER): to read the USER environment variable ("syssec")

  Output:

  ```
  bot-696: syssec, syssecvm, Linux, 4.1.6-1-ARCH, i686
  ```

- **.processes**

  Functionality:

  It executes the command `/bin/ps -a` which lists a snapshots of the running processes.

  Output:

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

```
bot-696:
  PID TTY          TIME CMD
  653 pts/0    00:00:00 python2
  675 pts/2    00:00:00 python2
  696 pts/1    00:00:00 bot
  860 pts/3    00:00:00 sudo
  861 pts/3    00:00:03 gdb
 1088 pts/1    00:00:00 ps
```

- **.flash**

  Functionality:

  This command displays shortly an image with the text `"You got hacked !!! Hahaha!"`. In order to achieve this, it executes the following script:

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

  Output:

  ```
  bot-696: SUCCESS.
  ```

- **.kill**

  Functionality:

  This command terminates the `bot` process.

  Output:

  ```
  bot-696: One will fall, but others will rise. Hasta la vista.
  ```

- **.fight**

  Functionality:

  This command makes two bots to fight with each other.

  Output:

  ```
  .fight bot-4702 bot-4701
  ```

```
[+] bot-4701: I bot-4701, you botmaster.
[+] bot-4702: I bot-4702, you botmaster.
Sent: .fight bot-4702 bot-4701
[+] bot-4701: Two bots enter, one bot leaves.
[+] bot-4702: Two bots enter, one bot leaves.
[+] bot-4701: I attacked...
[+] bot-4702: I attacked...
[+] bot-4701: I win, I live...
[+] bot-4702: I lose, oh noooooooooo...
```

Advanced Questions

Know we get to more advanced questions. Getting the best possible grade requires solving these, however they might be quite time-consuming.

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

Solution:

There are commands which do not show in the `strings` output because either are encoded or the command verification takes place byte by byte without keeping in memory a contiguous string representation of the entire command.

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

Solution:

The bot filters out all commands which do not begin with a `dot(".")`. The goal of this filtering is to prevent a brute force verification of all strings that show up in the output of `strings` command. This approach makes more difficult the reverse engineering of the commands, because the bot application must be disassembled and debugged.

How does a bot generate its name, it uses in the chatroom?

Solution:

The bot generates its name at startup by concatenating the string `"bot-"` with the PID of the bot process. The resulted bot name is stored in a global variable. The PID is retrieved by using the system call `getpid`.

Which other commands can you find? Describe how you found them, how the bot parses them, and their functionality.

Solution:

- **.secret**

  Reverse Engineering:

  I sent from the chat client a random string beginning with a dot, while the `gdb` debugger was attached to the bot process and a break point was set after the `select` call. Then I debugged step by step through all checks that use `strncmp` function. At the end of these unsuccessful checks the function `func3` was called which compares character by character the received string against a predefined string. By analysing the assembly code, I extracted the predefined string **secret**.

  Functionality:

  It changes the theme of the window manager by executing the following command

`xfconf-query -c xsettings -p /Net/ThemeName -s \"Xfce-dusk\"`

Output:

```
[+] bot-1585: Do you see a change? You may have to try a few times.
```

- **.e4stere66!1**

  Reverse Engineering:

  I sent from the chat client a random string beginning with a dot, while the `gdb` debugger was attached to the bot process and a break point was set after the `func3` call. Then I debugged step by step until the beginning of function `func2`. In this function, I debugged further the first iteration of the loop which decodes a predefined string, and I analysed carefully the decoding procedure. The decoding method performed a `xor` operation between a fixed value `0xffffff81` and each encoded byte.

  I was finally able to extract the encoded string from memory and to decode it byte by byte as follows:

  encoded: `0xe4 0xb5 0xf2 0xf5 0xe4 0xf3 0xe4 0xb7 0xb7 0xa0 0xb0 0x00`

  xor

  mask: `0x81 0x81 0x81 0x81 0x81 0x81 0x81 0x81 0x81 0x81 0x81 0x81`

  =

  decoded: `0x65 0x34 0x73 0x74 0x65 0x72 0x65 0x36 0x36 0x21 0x31 0`

  ASCII: `e 4 s t e r e 6 6 !  1`

  Functionality:

  It sends back to the chat client the string *Congrats, you found the easteregg. Document how you found it and the command for extra points.*

  Output:

```
[+] bot-3722: Congrats, you found the easteregg. Docu-
ment how you found it and the command for extra points.
```

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?

Solution:

The bots communicate using a UNIX domain socket (IPC socket). They need to know a path name which identifies the UNIX domain socket.

The bots use file descriptor 6 to communicate with each other. It can be observed in the trace bellow that this file descriptor is a UNIX domain socket.

```
sudo strace -p 8108
Process 8108 attached
```

```
select(6, [4 5], NULL, NULL, NULL)      = 1 (in [4])
read(4, ".fight bot-8106 bot-8108", 4096) = 24
write(4, "bot-8108: Two bots enter, one bo"..., 41) = 41
fsync(4)                                = -1 EINVAL (Invalid
    ↪ argument)
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL, [], 0}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
nanosleep({1, 0}, 0xbfffadbc)           = 0
socket(PF_LOCAL, SOCK_STREAM, 0)        = 6
connect(6, {sa_family=AF_LOCAL, sun_path=@"bot-8106"}, 110) = 0
write(6, "5\0\0\0", 4)                  = 4
write(4, "bot-8108: I attacked...", 23) = 23
fsync(4)                                = -1 EINVAL (Invalid
    ↪ argument)
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL, [], 0}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
nanosleep({1, 0}, 0xbfffadbc)           = 0
close(6)                                = 0
select(6, [4 5], NULL, NULL, NULL)      = 2 (in [4 5])
read(4, "bot-8106: Two bots enter, one bo"..., 4096) = 64
select(6, [4 5], NULL, NULL, NULL)      = 2 (in [4 5])
read(4, "bot-8106: I win, I live...", 4096) = 26
select(6, [4 5], NULL, NULL, NULL)      = 1 (in [5])
accept(5, 0, NULL)                      = 6
read(6, "\332\3\0\0", 4)                = 4
close(6)                                = 0
write(4, "bot-8108: I lose, oh noooooooooo"..., 36) = 36
fsync(4)                                = -1 EINVAL (Invalid
    ↪ argument)
close(4)                                = 0
close(5)                                = 0
exit_group(0)                           = ?
```

The bots exchange two values `"332 3 0 0"` and respectively `"5 0 0 0"` as can be seen in the trace bellow:

```
sudo strace -e trace=read,write -e read=29,30 -e write=29,30 -p
    ↪ 8106
[sudo] password for syssec:
Process 8106 attached
^[[1;5Aread(4, ".fight bot-8106 bot-8108", 4096) = 24
write(4, "bot-8106: Two bots enter, one bo"..., 41) = 41
write(6, "\332\3\0\0", 4)                = 4
write(4, "bot-8106: I attacked...", 23) = 23
```

```
read(4, "bot-8108: Two bots enter, one bo"..., 4096) = 64
read(6, "5\0\0\0", 4)                              = 4
write(4, "bot-8106: I win, I live...", 26) = 26
read(4, "bot-8108: I lose, oh noooooooooo"..., 4096) = 36
```

Then they use an algorithm to compare this value against some thresholds. The winner will keep running while the looser will terminate its execution.

The output of a combat is:

```
Sent: .fight bot-6430 bot-6352
[+] bot-6352: Two bots enter, one bot leaves.
[+] bot-6430: Two bots enter, one bot leaves.
[+] bot-6352: I attacked...
[+] bot-6430: I attacked...
[+] bot-6352: I win, I live...
[+] bot-6430: I lose, oh noooooooooo...
```

Notes

- 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].
- 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`).

References

[1] Assembly Language for x86 Processors (6th Edition), Kip R. Irvine

[2] The GNU C Library Manual, `http://www.gnu.org/software/libc/manual/`

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