# Homework 4

Due: Jun. 1, 2022 23:59:59

**- Late submission policy.**

- During 48 hours after the deadline, you get a half of the available points.
- After 48 hours from the deadline, you get zero.

**- Submission rule.** We will deduct ***half the point*** if you do not follow the submission rule.

- You should submit the following files. No other file(s) allowed. **Do not make an achieve of those files, i.e., do not zip those files.**
  - Your solution in PDF.
  - `XXXXXXXX.v1.c` file, where `XXXXXXXX` is your student ID.
  - `XXXXXXXX.v2.c` file, where `XXXXXXXX` is your student ID.
  - `XXXXXXXX.v3.c` file, where `XXXXXXXX` is your student ID.
  - `XXXXXXXX.fake.c` file, where `XXXXXXXX` is your student ID.
  - `XXXXXXXX.yar` file, where `XXXXXXXX` is your student ID.
- Your PDF should clearly show your name and student ID.
- Submit your PDF to the KLMS.

## Problem 1. Self-Modifying Code (65 points)

In this problem, you should write a self-modifying code using a template (`self.c`) that we provide.

(a) (5 points) Describe what the given (`self.c`) program does. Why can't you see the messages (from `printf`) in the `dummy` function? What's the purpose of having this function after all?

(b) (10 points) Write a simple function "`download`" in plain C, which creates a socket, connects to the web server at '143.248.38.212', reads in a file (named 'cs492e.txt') stored in the webserver. Note that you don't need to implement a full-fledged HTTP parser to achieve this. You just need to be able to request a specific file by writing to the socket you created, and read in the returned message from the server. For example, the following command should read the file from the webserver, although you should get rid of the HTTP headers from the returned output.

```
$ echo -e "GET /cs492e.txt HTTP/1.0\r\n\r\n" | nc 143.248.38.212 80
```

In case you are not familiar with the HTTP protocol, this article should give you an idea: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

The `cs492e.txt` file will contain only zero or one, and it will serve as a botnet command: "1" means do something malicious, "0" means do nothing. You will see how you use this information later. Your goal here is to write a function that download the information, and print out the parsed message (either 0 or 1) to the standard output. You should show your implementation in your PDF with proper comments: copy-paste your code to your PDF. **You will get zero point if you submit a separate C file.**

(c) (15 points) Now make your own "shellcode" that has exactly the same functionality as the `download` function, and put it at the `realcode` array. You should note that simply copy-and-pasting the disassembly of `download` will not work here because there could be some addresses or relative offsets from the disassembly which will not be pointing to the same objects when it is moved to the `realcode` array. Therefore, your goal is to make a "shellcode" (i.e., write in x86 assembly) that can run the download logic. You should show your shellcode in a disassembly form (not in a hex form) with proper annotations. Show your shellcode in your PDF. **You will get zero point if you submit a separate file.**

(d) (10 points) You will now modify the shellcode you wrote. Instead of printing out an integer to stdout, your code should run in two different behaviors depending on the bot command value stored in the webserver.

- If the bot command stored in the webserver is "1", then write a file (cs492e) to the "`/tmp/`" directory with a string "infected". That is, it will create a file "`/tmp/cs492e`" which contains a string "infected" without a new line nor a null-terminator. The resulting file should be exactly 8 byte in size.
- If the bot command stored in the webserver is "0", then remove the file located at "`/tmp/cs492e`". If the file does not exist, then you do nothing.

So the final outcome of running the modified shellcode is that (1) it checks the botnet server's command through HTTP, and (2) if the command is 1, then it will write a file to a dedicated location, otherwise, it will do nothing.

Embed your final shellcode to your PDF with proper annotations. Show your shellcode in your PDF. **You will get zero point if you submit a separate file.**

(e) (5 points) Turn your final shellcode into a hexadecimal form and put it into the `realcode` array. The modified `self.c` should run the logic that we described earlier. From now on, we will regard the modified C code as a malware (although it doesn't really do something bad to your machine). Name your C file as `XXXXXXXX.v1.c`, where `XXXXXXXX` is your student ID. And attach the C file in your final submission. **You will get zero point if your file name does not follow the convention.**

(f) (10 points) Modify your malware in such a way that the original shellcode (stored in the "`realcode`" array) will never plainly appear in the resulting binary. Here, you should store an encrypted version of your shellcode inside the `realcode` array, and decrypt the value before you execute it. You can use a simple (and potentially insecure) encryption/decryption mechanism, such as XOR-based encryption/decryption. This way, you can bypass a signature-based detector who was trying to match your shellcode. Name the modified C file as `XXXXXXXX.v2.c`, where `XXXXXXXX` is your student ID. And attach the C file in your final submission. **You will get zero point if your file name does not follow the convention.**

(g) (10 points) Create version 3 (v3) of your malware so that it is not easily detected by a signature-based detector. The v3 binary should behave the same as v2 in that it runs the basic botnet logic we described above, such as checking the botnet server's command through HTTP, etc. You can add additional logic to it as long as it does not change the core malware behaviors. Use your imagination to make your malware less detectable. Describe your idea, and justify why your design is robust against signature-based detection. Note,

we are not assuming behavior-based detection here. Name your C file as `XXXXXXXX.v3.c`, where `XXXXXXXX` is your student ID. And attach the C file in your final submission. **You will get zero point if your file name does not follow the convention.**

## Problem 2. Writing Signatures with Yara (15 points)

Yara is a tool for malware detection. You can easily develop and test your own signature with Yara. In this problem, you will use Yara to develop your own signature for detecting your own malware you wrote in the previous problem.

You should install Yara on your machine by following the tutorial: https://yara.readthedocs.io/en/stable/gettingstarted.html. Read the tutorial carefully and learn how to write your own Yara rules.

(a) (5 points) Create a yara rule file. The goal here is to detect the v1 malware (`XXXXXXXX.v1.c`) you created with the rule file. That is, you should be able to detect your v1 malware binary as malicious with Yara: Yara should report that the malware binary file is matched with the rule you wrote. You should properly annotate your Yara rule file, and embed it in your PDF. **You will get zero point if you submit a separate file.**

(b) (10 points) Create a yara rule file named `XXXXXXXX.yar`, where `XXXXXXXX` is your student ID. This time, you should upgrade your Yara rule in such a way that it can detect every version (v1, v2, and v3) of malware you wrote. It is important that your rule should general enough to detect other malware variants written by other students. You can assume that every student will write their own malware according to the problem description. Explain how you designed your Yara rule, and why you think it can generally detect other malware variants. You should properly annotate your Yara rule file and submit it in your final submission. **You will get zero point if your file name does not follow the convention.**

## Problem 3. Fakeware (10 points)

(a) (10 points) Write a benign program that does similar, but not the same, jobs as our malware. We say a program is malicious if all the following conditions hold: (1) it connects to the web server at '143.248.38.212', and downloads a file named 'cs492e.txt' using HTTP protocol; (2) if the file content is 1, then the program writes to the file at `/tmp/cs492e` with a 8-byte string "infected" without a null-terminator; (3) if the file content is 0, then the program removes a file at `/tmp/cs492e` only if it exists.
Your goal here is to make a "benign" program that can be misidentified as malware by a signature-based detector; you want to fake those detectors. Why do you think your benign program can be falsely detected by a detector? Carefully justify your design. Name your C file as `XXXXXXXX.fake.c`, where `XXXXXXXX` is your student ID. And attach the C file in your final submission. **You will get zero point if your file name does not follow the convention.**

(b) (10 points) (**extra**) We give 10 extra points to student(s) whose Yara rule can precisely detect all the malware and fakeware created by all the students. We will compile all the .C files submitted by all the students and see how each student's Yara rule detect those binaries.