Assignment 4, Fall 2018
CS4630, Defense Against the Dark Arts
File Infection via Tricky Jump
Jumpicus Captiosam Curse

Purpose

This assignment will explore what it takes to create a stealthy virus that employs a tricky jump. A tricky jump is a form of hijacking in which a jump is inserted to call some virus code. The jump is inserted in such a way that after the virus code runs, the program continues normal execution, thereby maintaining stealth.

Due

This assignment is due on **Thursday, 11-OCT-2018 at 11:59 pm**

Tricky Jump (Jumpicus Captiosam)

A tricky jump can be efficiently implemented using only six bytes with the following assembly code:

```
push <Address of Virus Payload>
ret
```

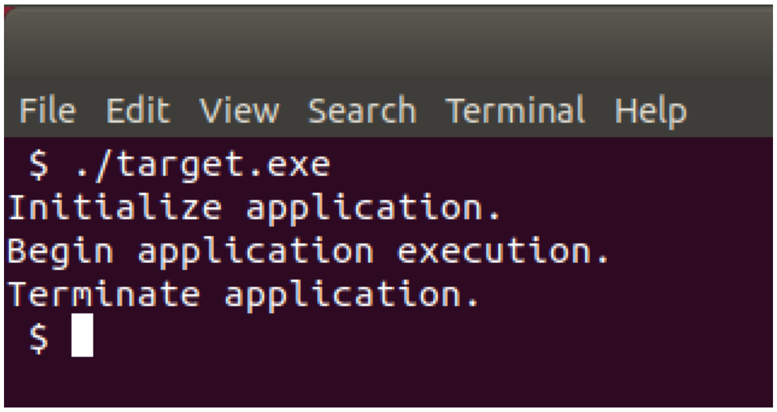
When this sequence is executed, control is diverted to the virus code. When the virus code returns, control returns to the function that called the function that contained the tricky jump. If the virus writer inserts the tricky jump at the end of an application function (i.e., the epilogue code), then the program, after the virus code executes, will continue to run as if nothing happened. Here is an example (taken from the output of `objdump` with the use of the `--disassemble` flag).

```
                        <deleted lines>
8048423:        83 ec 18                sub    $0x18,%esp
8048426:        c7 04 24 61 85 04 08    movl   $0x8048561,(%esp)
804842d:        e8 e6 fe ff ff          call   8048318 <puts@plt>
8048432:        c9                      leave
8048433:        c3                      ret
8048434:        90                      nop
8048435:        90                      nop
8048436:        90                      nop
8048437:        90                      nop
```

Notice the padding with nops at the end of the function. This is essentially a cavity that gives the virus writer some room to work. If a tricky jump is inserted starting where the `ret` instruction is located (address `0x08048433`) then the virus code will be invoked. When the virus code returns, control will be returned to the function that invoked the tricky jump.

For this assignment, you will write a C program that infects a Linux 32-bit ELF executable. The infected file causes some virus code to be executed. You will use the tricky jump method of infection and will also inject the virus payload which you will create.

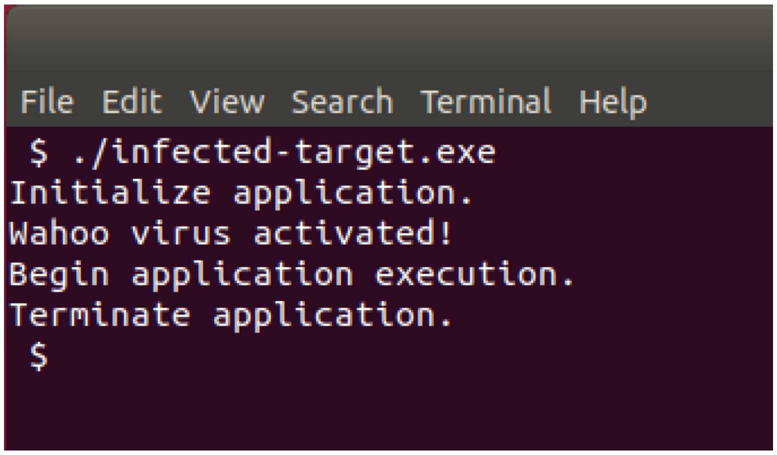
The Linux executable you want to infect is called `target.exe`. This executable is provided to you on the course Collab site. When you run an uninfected version of `target.exe`, it produces the following output.



```
 $ ./target.exe
Initialize application.
Begin application execution.
Terminate application.
 $ 
```

Figure 1: `target.exe` output

Now, we run the `infect` program which processes `target.exe` and produces `infected-target.exe` When we run `infected-target.exe`, it produces the following output.



```
 $ ./infected-target.exe
Initialize application.
Wahoo virus activated!
Begin application execution.
Terminate application.
 $
```

Figure 2: `infected-target.exe` output

Notice that after calling the initialization function, the virus payload has been executed which prints a message and then execution continues in the application as normal.

Assignment Details

1. Write a C program called `infect.c` that when compiled and executed reads a Linux executable called `target.exe` and produces a new file named `infected-target.exe` where a tricky jump and virus payload has been inserted so that when the infected executable is run, the payload which prints `Wahoo virus infection!` is executed before

the main application runs. See the previous example output to see the output the infected executable should produce.

2. Your program should not change the size of the executable. That is `target.exe` and `infected-target.exe` should have identical sizes. The size of file `target.exe` is 7324 bytes.

Methodology and Hints

1. You should use the utility `objdump` to examine the executable `target.exe`. The option `--disassemble` is useful. In particular, you need to determine the starting address of the virus payload code you inject. The dissasembly will also help you determine the opcodes of the instructions that you need to insert (i.e., a `push` instruction and a `ret` instruction).

2. The trick is that you must map the runtime address of the location in the executable to the offset of the corresponding byte in the file. You need to do this mapping because the file offset where you want to write is not the same as the address of the instruction when the program is loaded in memory (which is what `objdump` is showing you). `objdump --disassemble`'s output gives the runtime addresses. Using the disassembly from step 1 and the hexadecimal dump of the raw file, you can determine which bytes of the file must be overwritten.

3. A very useful program to examine the file is `ghex`. You can install it using `apt-get`. It is a visual binary file editor. Google around to see what is available.

4. Characters are encoded using ASCII and strings in C are null-terminated.

5. To simplify the assignment, you can hardcode the input and output file names in your infect program. That is, `infect.c` opens and reads `target.exe` and opens and writes `infected-target.exe`. After you produce `infected-target.exe` you will need to set the execute permissions on the file using the `chmod` utility.

6. The hard part is figuring out what locations in the file need to be changed and what they should be changed to. The code to perform the infection is small. My `infect.c` is roughly 37 lines of C code not counting comments. I wasnt trying to make it short so a solution could easily be about 25 lines of code.

7. **Troubleshooting tip:** If you are encountering segmentation faults while testing your `infected-target` using `gdb` to examine the stack and the instruction being executed (as well as the instruction address) can help you to understand what might be going wrong with the execution of your virus payload and when.

8. **Very important hint**: We are reading and writing **binary** files - not text files. You need to open the files in binary mode so the OS does not interpret the input characters (e.g., treat ˆD as an end-of-file).

Helpful Resources

- **Intel Software Developer's Manual**:
  `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf`
- **ASCII Table**:
  `http://www.asciitable.com/`

Items to Submit

Your submission consists of two files.

1. A text document named **`method.txt`** that describes the method that you used to solve the problem. That is, a description of how you determined the locations to overwrite in the file `target.exe`. A paragraph or two should be sufficient. In the file **`method.txt`** include answers to following questions:
   (a) What is the file offset of the start of the function `main`?
   (b) What is the file offset of the start of the function `TerminateApplication`?
   (c) What is the file offset of the start of the function `IntializeApplication`?
2. Your C code should be in a file called `infect.c`. Since the lines of code required is modest (25-40 lines), all your code should be in this one module.

It is mandatory that you use the file names given to ease the task of grading many different submissions of this assignment. Throughout the semester, you will be given file names. All assignments will be submitted using the class Collab Website.