Assignment 5, Fall 2018
CS4630, Defense Against the Dark Arts
Detecting Virus Code Patterns

Purpose

This assignment will teach you to recognize virus code patterns unique to viruses using regular expressions. The regular expressions will be defined using `flex`, a modern version of the well-known utility `lex`.

Due

This assignment is due on **Thursday, 18-OCT-2018 at 11:59 pm**

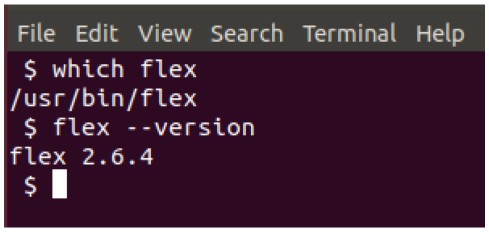
Installing `flex`

For this assignment, you will be using the `flex` utility. It is a computer program which generates lexical analyzers, also known as ”scanners” or ”lexers.”

To use `flex`, you will need to install the package on your Ubuntu 18.04.1 LTS virtual machine. You can do so using the `apt-get` utility:

```
sudo apt-get install flex
```

To test whether `flex` was installed, you can use the `which` utility to find out the installation path, or you can invoke `flex --version` to find out the installed version.



```
 $ which flex
/usr/bin/flex
 $ flex --version
flex 2.6.4
 $ 
```

Figure 1: Testing `flex` installation

1 Virus Code Pattern #1 - Tricky Jump Detection

This assignment will focus on the recognition of the virus code that we just inserted into an executable using the tricky jump. A typical example is:

```
push <virus_code_addr>
ret
```

We are not concerned with the particular function name. It would be unusual for a legitimate application to push an address immediately before returning, and most compilers have no occasion to generate this sequence of instructions. The presence of this sequence likely indicates that a virus has infected the executable.

Using `flex`

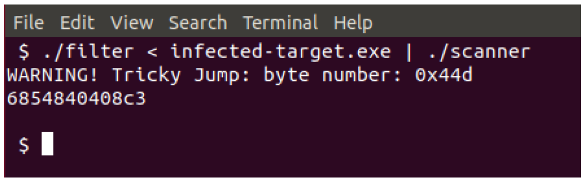
1. As an initial test case, use the infected file that you created in Assignments 4 . An additional test case containing a tricky jump infection will be provided on the class Collab site (`Wahoo_target.exe`).

2. Because `flex` works on text files, we will need to write a simple filter that reads the binary file from standard input and writes a stream of ASCII bytes in hexadecimal to standard output. (Note: This type of preprocessing is a standard programming trick when you want to use a tool, but the tool doesnt quite handle the input that you have.) The hexadecimal bytes should be written in lowercase. Here is what filter produces for the first bytes of the file `infected-target.exe` when executed.

   ```
   $ ./filter < infected-target.exe
   7f454c460101010000000000000000000200030001000000030 .......
   ```

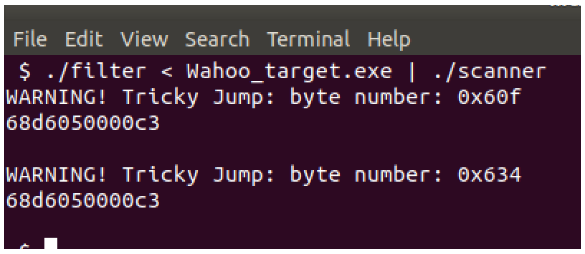
   Hint: The `stdio` function `freopen(NULL, "rb", stdin)` will help you reopen `stdin` in binary mode. The program filter is small. You should be able to implement it in fewer than 25 lines of C/C++ code.

3. Write a `flex` pattern file to scan the binary executable and detect the virus code pattern discussed above. You should detect any tricky jump pattern regardless of the function address pushed by the first instruction. Keep track of the byte number in the input file as you process the file. When a virus is detected, print a warning message with the byte number and the name of the pattern. Start numbering at byte 0 and report the byte number of the first byte of the the byte sequence that matches the pattern. After the warning message, print the bytes that matched the pattern, followed by a blank line. Do not write anything else to the output stream. A sample run and output would be:



```
 $ ./filter < infected-target.exe | ./scanner
WARNING! Tricky Jump: byte number: 0x44d
6854840408c3

 $
```

Figure 2: Virus detection in `infected-target.exe`



```
 $ ./filter < Wahoo_target.exe | ./scanner
WARNING! Tricky Jump: byte number: 0x60f
68d6050000c3

WARNING! Tricky Jump: byte number: 0x634
68d6050000c3
```

Figure 3: Virus detection in `Wahoo_target.exe`

   Note: The output in Figure 2 is for illustrative purposes. It may be that a file contains multiple instances of a tricky jump. Your scanner should detect all instances of a tricky jump (i.e., do not terminate the scan after reporting the first occurence of a tricky jump).

4. Call your `flex` file `virus-patterns.l`. Compile and link the flex code into an executable. To jump start you, here are the commands necessary to create your scanner:

   ```
   $ flex virus-patterns.l
   $ gcc -o scanner lex.yy.c -lfl
   $
   ```

   The `-lfl` option to `gcc` tells the linker to search the `flex` library for any support routines necessary.

5. Run and test your solution. You might want to create some additional input files that contain sequences that the pattern should match as well as some sequences that are close but that your scanner should not match.

6. Note: we will be testing your code on a variety of inputs to see if you get false positives or false negatives.

2 Virus Pattern #2 - Cavalier Encrypted Virus

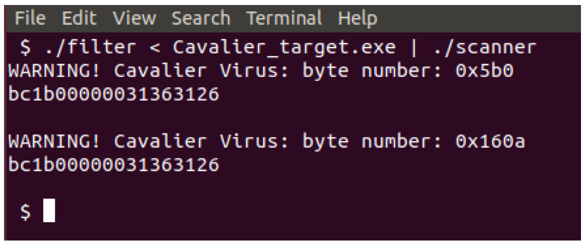
The Ministry of Magic (MOM) has discovered a new virus, the Cavalier virus. Through static and dynamic analysis, MOM Aurors have discovered that the Cavalier virus uses encryption to disguise the virus code. To make the analysis even more difficult, the virus writer uses the stack pointer (`%esp`) in the decryption routine (anti-debugging technique). This fact can be used to detect the virus. Infected samples analyzed by MOM revealed that the Cavalier virus code sequence begins with the following code (comments added by MOM Aurors):

```
leal Start, %esi ; load start address of location of virus payload
movl $0x682, %esp ; put length in stack pointer
xorl %esi, (%esi) ; begin decryption
xorl %esp, (%esi)
```

MOM Aurors have determined that the virus uses different payloads (different sizes), so a good signature for recognizing the Cavalier virus is:

```
movl <constant>, %esp  ; put length in stack pointer
xorl %esi, (%esi)  ; begin decryption
xorl %esp, (%esi)
```

Write a `flex` pattern file to scan the binary executable and detect the Cavalier virus code pattern discussed above. Your output should be formatted like the output in Figure 4.



```
 $ ./filter < Cavalier_target.exe | ./scanner
WARNING! Cavalier Virus: byte number: 0x5b0
bc1b00000031363126

WARNING! Cavalier Virus: byte number: 0x160a
bc1b00000031363126

 $
```

Figure 4: Virus detection in `Cavalier_target.exe`

3 Virus Pattern #3 - Rotunda Anti-Debugging Virus

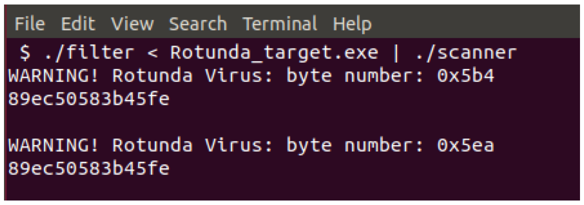
The Rotunda virus attempts to detect the debugger and alter its behavior if it believes it is running in a debugger. Again, the MOM Aurors have analyzed some infected programs and determined that the Rotunda virus detects the debugger using the following code sequence (comments added by MOM Aurors):

```
mov %ebp, %esp ; Monitor the stack pointer's value
push %eax ; Store %eax on the stack
pop %eax ; Pop the top of stack back into %eax
cmp -2(%ebp), %eax ; Compare against the stack
jne DEBUG  ; Debugger detected!
```

The MOM Aurors have determined that a good signature consists of detecting the following pattern:

```
mov %ebp, $esp
push %eax
pop %eax
cmp -2(%ebp), %eax
```

Write a `flex` pattern file to scan the binary executable and detect the virus code pattern discussed above. Your output should be formatted like the following sample output

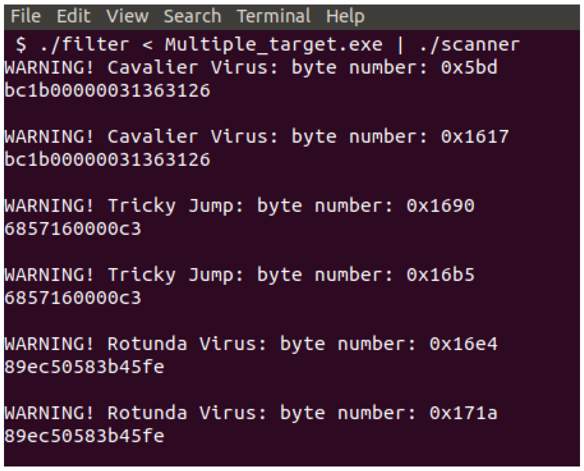


```
 $ ./filter < Rotunda_target.exe | ./scanner
WARNING! Rotunda Virus: byte number: 0x5b4
89ec50583b45fe

WARNING! Rotunda Virus: byte number: 0x5ea
89ec50583b45fe
```

Figure 5: Virus detection in `Rotunda_target.exe`

4 Detecting infections by multiple viruses and testing your viruse patterns

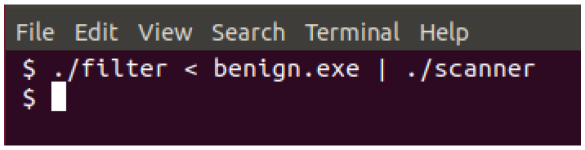
Once you have implemented a scanner containing all three virus signatures, your scanner should be able to detect if a file is infected with multiple viruses and display the offset and byte string which matches the infection signature. Figure 6 shows a sample detection of infection by multiple viruses.



```
 $ ./filter < Multiple_target.exe | ./scanner
WARNING! Cavalier Virus: byte number: 0x5bd
bc1b00000031363126

WARNING! Cavalier Virus: byte number: 0x1617
bc1b00000031363126

WARNING! Tricky Jump: byte number: 0x1690
6857160000c3

WARNING! Tricky Jump: byte number: 0x16b5
6857160000c3

WARNING! Rotunda Virus: byte number: 0x16e4
89ec50583b45fe

WARNING! Rotunda Virus: byte number: 0x171a
89ec50583b45fe
```

Figure 6: Virus detection in `Multiple_target.exe`

Your scanner should also not detect viruses for files which do not contain the virus patterns. Figure 7 shows output if no virus pattern is matched.



```
$ ./filter < benign.exe | ./scanner
$ 
```

Figure 7: Virus detection in `benign.exe`

Items to Submit

There are two files to submit. Both files should include your name (as a comment). This assignment is due on **Thursday, 18-OCT-2018 at 11:59 pm** .

1. Upload your `flex` patterns file. It must be called `virus-patterns.l`. This file should contain the patterns for detecting all three viruses: Wahoo, Cavalier, and Rotunda viruses.

2. You should also submit your filter code. It must be called `filter.c`.

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.