Exercise 2 – Buffer Overflows

Deadline: 10.04.2013 15:00

Note: In order to solve this exercise you'll probably need to use IDA, OllyDbg and a hex editor. These tools will be introduced in the upcoming lectures. Whoever feels comfortable is welcomed to begin this exercise early, but if you don't know where to start from – no worries – all will be explained.

Note 2: The attachments include .EXE files and .DLL files. The .DLL files are simply run-time libraries that are not considered part of the exercise. You'll need the .DLL files in case you don't have the Visual Studio 2010 runtime libraries installed.

Part 1 – First Steps in Reverse Engineering (20 points)

Question 1

Later on, in this exercise, you'll be asked to exploit a buffer overflow vulnerability in the file `ex2_1.exe`. As always in the security research business, you don't really know what this executable does and you don't have access to its source code. Therefore, your first step is to reverse engineer the file.

Please answer the following questions:

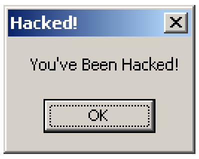
1. Locate the `main()` function. What is its address?
2. There are 2 functions which are called from `main()`. Where are they located? What does each of them do? What are the input and output parameters?
3. Sometimes it's a good idea to write a C function which is equivalent to a piece of assembly code in order to better understand it. The second out of the two functions mentioned above (the one which is called later) performs some manipulation on data. Write a C function which simulates that function **exactly**.
4. What does the code do? Explain the main function. There's no need to re-explain the sub-functions.
5. What's the weakness in this code? How would you change the code to prevent such a weakness?

Part 2 – Buffer Overflows (80 points)

Question 2 (50 points)

Now that you know where the weakness is and you've learned in class how to exploit this vulnerability, it's time to overflow some buffers.

Your goal is to make the program output a message box like this:



Your experience in the previous exercise with Win32 API and dynamic loading of functions can be of great help.

The code receives a filename as a parameter. Please submit such a file (named `q2.bin`). The file should cause a buffer overflow which will enable arbitrary code execution. The code will display the message box as explained above and resume normal execution of the program. The program must terminate normally with return value 0.

Note that your input buffer may be changed throughout the execution of the program. Avoid that by using local variable manipulation.

Also note that your exploit might cause some garbage to be displayed on screen. That is okay.

You'll also need to submit a file named `q2.txt` which includes the contents of `q2.bin` with the corresponding opcodes. For example, if `q2.bin` contains 2 bytes: `CC C3`, the text file will contain:

CC	INT 3h
C3	RET

Also in this file you'll need to include comments that will explain every meaningful block of code.

Question 3 (30 points)

Great success! You are a hacker. It's time to step it up a notch.

Take a look at `ex2_2.exe` – it's similar to `ex2_1.exe` with one "minor" difference. You should be able to figure out the difference without reversing the code, simply try your previous exploit and see what happens.

Your goal is exactly as before: display the above message box and continue executing normally.

Some tips:

- No need to completely redo the reverse engineering part. This is exactly the same code with small functionality added. Everything should look familiar. While the functions are operationally the same, the locations of the functions might be different, so don't count on offsets you've found in the previous question.
- You might want to create two parts in your shellcode. One is similar to the shellcode in the previous question (in an encoded format) and the other is some sort of a

decoding routine that decodes the first part and then runs it.
The decoding routine itself must pass the filter.

Your exploit should be submitted in a file named `q3.bin`. A corresponding `q3.txt` should also be submitted. Note that in `q3.txt` you'll also need to explain the instructions after decoding.

As always, a partial solution is better than nothing. If you didn't managed to pass the filter completely, submit your best try.

Submission

All solutions should work under the Windows machines in Ross 19.

The following files should be included in the submitted tarball file:

- `q2.txt`
- `q2.bin`
- `q3.txt`
- `q3.bin`
- README – Formatted according to the course guidelines and includes answers to the questions in the exercise.

A single TAR file named `ex2.tar` should be submitted. Make sure you run the pre-submit script before submitting the exercise.

Good luck!