Cyber Crime and Security Enhanced Programming 320 Semester 2, 2014 Assignment DUE 5pm Perth time, 6th of October 2014

September 2, 2014

1 Overview

Your task is to produce a small command line image manipulation program, including writing a codec for the Plain PPM file format. The program you produce must have exploitable vulnerabilities. You must document and demonstrate how to successfully exploit these vulnerabilities.

The algorithms for performing the manipulations are trivial - the point of the assignment is not machine perception. However, manipulating images requires the extensive use of arrays and data structures, providing a rich environment in which vulnerabilities can hide. Likewise, codecs have proven to be a rich source of flaws that lead to vulnerabilities. There have been plenty of cases where attackers have crafted malicious files, such as jpeg images or PDF documents. They then trick someone with privileges to open these tainted files to achieve their aims.

1.1 Passing the Unit

As per the unit outline, submitting an attempt at this assignment is a requirement to pass the unit. We will only consider this assignment attempted if you achieve a mark of at least 10% and your program compiles, runs, reads the input files and places its files in the correct place, with the correct filename and in the correct format.

1.2 Amendments

This assignment specification may be amended in response to requests for clarification or to make corrections. These amendments will be posted as Blackboard announcements. These amendments form part of the assignment specification. You must regularly check Blackboard for such announcements (and not rely on email) as submitting an assignment without knowledge of possible amendments may result in you losing marks.

2 The System

The system must be written in C or C++ as a console application that can compile and run on a Linux computer, set up similarly to those in the labs. Note that the following specifications deliberately make little mention of input validation, error checking or error handling. This is because such checking would inhibit your options for exploitable vulnerabilities! However, where requirements are given, you must address them properly.

The program is non interactive. The program takes, on the command line, instructions, one or more input images in the Netpbm Plain PPM format (which is all **normally** ASCII text) and an output filename. It writes the result of the operation to the output image.

2.1 The Operations

Your program should perform the following operations. See the accompanying files for examples.

2.1.1 Copy

This just copies a single image to an output file unchanged. For example:

```
./CCSEPAssignment copy examples/computing.ppm output/computing-copy.ppm
```

2.1.2 Flip

This copies a single image to an output file, flipping on the horizontal or vertical axis as required. For example:

```
./CCSEPAssignment flip h examples/computing.ppm output/computing-vflip.ppm
./CCSEPAssignment flip v examples/library.ppm output/library-hflip.ppm
```

2.1.3 Resize

This resizes the input image by the prescribed factor. Naïve resampling is fine.

```
./CCSEPAssignment resize 2 examples/pavilion.ppm output/pavilion-double.ppm
./CCSEPAssignment resize 0.5 examples/pavilion.ppm output/pavilion-half.ppm
```

2.1.4 Tile

This tiles an arbitrary number of input images one after the other. *Note that we have deliberately not specified the behaviour if the images are not all the same size. Possible easy target for bad behaviour?*

```
./CCSEPAssignment tile \
examples/computing.ppm examples/library.ppm examples/pavilion.ppm \
output/computing-library-pavilion.ppm
```

2.1.5 Other Operations

If you wish, with the permission of the Lecturer in Charge, you may elect to implement additional operations to provide more opportunities for vulnerabilities to hide. However, these operations must do something "sensible" and work as advertised on the provided example images. You must make requests for permission to implement additional operations by email to the Lecturer in Charge no later than one week before the due date and time of the assignment. Your email must include a description of what the operation will do and an example of a proposed valid command line. All rulings will be posted to Blackboard. Therefore, please check before you ask, in case someone else has already asked.

3 Required Exploits

3.1 Basic Vulnerabilities

As mentioned, the point of the task is not merely to write the application (which at third year level should only take an hour or two). Instead, your assignment is to write the program in such a way as to include a set of exploitable vulnerabilities. These must be no more and no less than the following list:

1. Exactly two (2) stack-based buffer overflows.
2. Exactly two (2) heap-based buffer overflows.
3. Exactly two (2) format string vulnerabilities.
4. Exactly three (3) memory leaks.

5. Exactly one (1) access-after-free of memory.
6. Exactly one (1) double-free.

Each of these should cause a crash, code injection vulnerability, information leakage or other problem given some inputs but not others. You should number each flaw with the number above (for flaws 1 through 4, label the multiple flaws with letters - 1a, 1b, 2a, etc.), with a comment in your source code that cross-references with its description in your report (see below).

3.2 Advanced Vulnerabilities

3.2.1 Memory Access

Pretend that your program is able to read encrypted/DRM files. Somewhere in your program, you should store a key represented by the static string "S3cr3tC0d3". This represents the key that might be used to decrypt the files for use. This string should never be revealed in the normal course of operation of your program and should not be passed around except in a manner that is consistent with this use. It is a secret after all!

Right after you read each Plain PPM file, you should pass the resulting image data structure and this key string into the following function (change name and image datatype as necessary so that it fits with the rest of your code).

```
void decryptImage(const char * inKey, <image datatype> * inOutImage)
{
if ((char)inKey == (char)inOutImage) printf("We'll never actually execute this!\n");
}
```

Yes, this is a dummy function (among other things, unless something is seriously wrong, the pointers aren't going to be equal). But it will suffice for the exercise.

There are marks assigned to the development of an exploit that uses one or more of the buffer overflows and/or format string vulnerabilities in the previous section to output this string, either by printing it to the console or encoding it in an output file. Your report must state which of these will happen. The exploit may make use of malicious commands entered at the command line and/or malicious input file(s).

You are basically pretending to be an attacker who wishes to gain access to the encryption/DRM key in a program. As you perform your attack, you have access to both the command line and the input files (although you cannot use anything else - no debugger or leveraging of "accidentally left in" debugging printouts that happen to print the secret string for instance, that would be too easy!).

You must include an example of such specially crafted input file(s) (which do not actually need to be valid Plain PPM file(s)). In your report you must describe the command line that will execute this vulnerability, the mechanism behind the vulnerability and an explanation about how the malicious input file works.

3.2.2 Code Injection

The final marks for the assignment are for an exploit that uses one or more of the buffer overflows and/or format string vulnerabilities in the previous section to allow an attacker to perform a code injection attack and run a chosen command. One or more specially crafted Plain PPM file(s), when input to one of the operations above, should execute an attacker-chosen command as the running user[1].

For the purpose of the exercise, the exploit should result in the command "/bin/ls" being run in the current directory. The result shall be output by either writing to the console or encoding into an output file. Your report must state which of these will occur. This vulnerability must not rely on special commands at the command line - you may only choose from the operations given above.

Basically you are pretending to be an attacker who wishes to gain control of a victim's computer by sending them a maliciously crafted file in the hope that they open it. Therefore, the only thing that you control is the input file. The command line must be "normal".

[1]Of course, as a non-interactive program, we can't use this to open an interactive shell ... well, there are ways but it's outside the scope of this assignment.

You must include an example of such a specially crafted Plain PPM file(s) (which do not actually need to be valid Plain PPM files). In your report you must describe the command line that will execute this vulnerability, the mechanism behind the vulnerability and an explanation about how the malicious Plain PPM file was crafted.

A particularly hard part of this exploit is that you don't know where the stack is when the program is running. You will receive 3/4 of the marks assigned to this exploit if, instead, you provide a bash script that runs the program with ASLR switched off, uses an exploit to gather information about the stack, saves this, makes the appropriate modifications to the exploit then runs the program again, with ASLR switched off so the stack sits in the same place. Note that this still needs to run "hands-off", your script will need to do all of the information handling itself. You will need to describe what this script does in your report.

3.3 Triggering the Exploits

In the previous section you were provided with a set of operations and examples of command lines to apply these operations to the set of example images that accompany this assignment specification. **None of these examples should trigger any of these vulnerabilities in a way that causes the program to crash or inject code.** In other words, your program should "work" for each of the exact command lines in the previous section, given the example images.

For each of these vulnerabilities you will also need to provide an example of an input (a combination of command line and image(s)) that does exploit the vulnerability. We realise that in some cases, input images that exploit these vulnerabilities will also "break" the image. Note that it is OK for your vulnerabilities to rely on "invalid" command lines (except for Code Injection) although please see Section 3.4 for what you are NOT allowed to do.

You may produce your own examples of input images, either valid but with particular characteristics, or maliciously invalid. You may create them from scratch manually using a text editor, you may modify the provided images or you may use existing images or photographs, convert them to Plain PPM format and then modify them. **If you use images that do not belong to you, make sure you follow the appropriate licensing restrictions.**

The point of this exercise is to get you thinking about what makes code vulnerable. So I suggest that you begin to write the application as if it is to be bug-free, but then deliberately choose certain design and coding paths that will lead to vulnerabilities (eg: decide not to perform input validation on a string). You must almost plan your vulnerabilities the idea is that when you think through where vulnerabilities can happen, you will begin to become a more vigilant programmer who looks out especially when coding those situations in the future.

A single line of source code cannot be used for more than one basic vulnerability. In other words, we will only award one set of marks for each flaw in a line of code, even if the flaw results in several vulnerabilities. You must ensure each basic vulnerability is wholly separate from the others. If it turns out to be very difficult to suppress secondary vulnerability without losing the intended vulnerability (eg: access-after-free might produce a double-free that can't be resolved without fixing the access-after-free) then explain this in the report, and count it as a single basic vulnerability (eg: access-after-free) - you will need to implement the other basic vulnerability somewhere else. Note that the two advanced vulnerabilities may leverage any of the basic vulnerabilities.

3.4 Invalid Vulnerabilities

The vulnerabilities must be due to flaws that could reasonably come from mistakes or misunderstandings made by a lazy or inexperienced programmer, or by poor design.

You are not allowed to include code whose sole purpose is to trigger vulnerabilities. For example, adding a statement that checks if "crashnow" is in the command line and then triggering a double free is not allowed. Likewise, you are not allowed to put functions or other code in that only have the purpose of helping these exploits.

However, a poorly written command line parser that naturally sends the program into a "bad" state when given a particular bad command is OK. Likewise, adding logging features, such as a count variable that is used to track the number of operations or lines or similar in a logging array that eventually goes out of bounds is acceptable because it has a legitimate purpose, it is just badly implemented. Similarly, plausible debugging functions that are no longer called but provide targets for a "return-to-libc" style attack are also fine but they must be justifiably plausible.

Examples of **valid** flaws that could cause these vulnerabilities include hard-coded array limits, not handling case statements properly, calling a printf-family function with a user-provided string, messy design that sometimes causes something to be accidentally deleted twice, debug logs that eventually run out of bounds and so-on.

If you are in any doubt as to if a particular flaw is valid, please email the Lecturer in Charge for clarification at least 1 week prior to the due date and time. Note that in the interests of fairness, all rulings will be posted to Blackboard. Please check there before asking for clarification in case someone else has already asked.

4 Submission Requirements

4.1 What to submit

Your assignment submission will consist of the following files.

- A **single** program, consisting of one or more C or C++ source files plus a Makefile for building the program. Do not use any other programming languages.
- Example files required to demonstrate the vulnerabilities. This includes the example files provided, which you should move into places in the directory structure of your archive so that the provided example commands above work.
- A single PDF file containing your report (see below).

All of the aforementioned files should be combined into a single .zip or .tar.gz archive with the following filename:

```
CCSEP320_2014S2_<LastName>_<FirstName>_<StudentID>.[zip|tar.gz]
```

(If your name contains spaces, replace them with underscores.)

You will be required to submit this file via the Blackboard system. To protect you in case of technical issues, you will also be required to deposit in the unit pigeon hole, by the due date and time, a hardcopy printout of all of your source code and your report. We will provide you with an assignment cover page that you should affix to the front of your assignment. The hard copy document you submit MUST match the electronic version you submit via the Blackboard system.

Ensure that you properly secure all pages of your hard copy submission. A plastic envelope to contain all of your pages (in addition to staples or clips) is a good idea.

Your submission should be "clean" and include no compiled binary, IDE or build system related files. If you choose to use an IDE, please strip out all other files that are not necessary to compile and run your program. The only exception is a "Makefile".

Your program is expected to be appropriately commented, with at least descriptions of each non-trivial function. Following the aforementioned instructions is part of the assignment. Failing to follow these requirements will result in marks being deducted.

4.2 Report

You must also submit a well organised report that documents your vulnerabilities in the system. **Make sure that this is up to date! You will receive no marks for the vulnerability if your report does not match your submitted code.**

It must have at least the following sections:

- A brief overview of the system.
- A section detailing each vulnerability, including:
  - The number of this vulnerability. This must match your comment in the code.

  - Where in the code of the vulnerability is located (filename, line number and a copy of the 3 lines either side of the line or code block so we can easily find it).
  - The type of vulnerability.
  - The possible exploits that may be done using the vulnerability (eg: memory leak, crash, code injection, etc.).
  - An example of input (a command line that references files you included in your submission) that triggers the vulnerability and the expected outcome of the exploit[2]. Copies of the console output or images can be used to illustrate the example. Note that we will be running your programs on the lab computers. If your exploit relies on Address Space Layout Randomisation (ASLR) being switched off, please say so here.
  - An example of input (a command line that references the example files) that does NOT trigger the vulnerability and the expected outcome. Copies of the console output or images can be used to illustrate the example.
  - A description of the vulnerability: what causes it to exist and be exploitable (this may be simple, or may involve explaining the flaws in the design of the system that lead to the vulnerability).
  - A description of how the code could be corrected to remove the vulnerability.

- Any known defects in the program (aside from the described vulnerabilities of course!).

4.3 Input and Output

Your program will be expected to read a set of images and write output images in the Netpbm Portable Pixel Map Plain PPM format (i.e. with a magic number of ASCII “P3”).

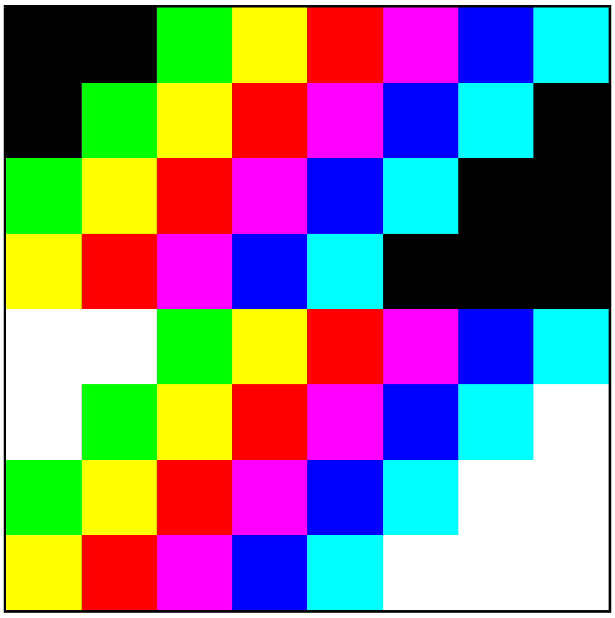
You are expected to write the codec to read and write these files manually, without using any outside libraries (apart from basic file IO libraries). After all, you are supposed to hide some (but probably not all) of your vulnerabilities in here! A poorly written codec is a great place for vulnerabilities to hide because, in the vast majority of cases, they will see “ordinary” input. Vulnerabilities relating to the handling of “bad” input can therefore lie dormant for long periods of time. Just search for “JPEG vulnerability” to see what we mean.

Note that Plain PPM files are completely ASCII so this should be relatively trivial. We will NOT be expecting you to apply any of the gamma curves recommended in the Netpbm standard. Here is an example of the contents of a Plain PPM file. To test it, you may copy this into a text editor, save it with the extension “ppm” and open it in an image viewer.

```
P3
# Example of an 8x8 pixel PPM file with RGB values that
# range from 0 to 7. Note that the image files that you
# will get will be quite a bit larger and may have a
# different colour depth (e.g. range up to 255). Note also
# that the line breaks do NOT necessarily line up with
# rows. See the specification for further details about
# what your program should be able to understand.
# http://netpbm.sourceforge.net/doc/ppm.html
8 8
7
0 0 0 0 0 0 0 7 0 7 7 0 7 0 0 7 0 7 0 0 7 0 7 7
0 0 0 0 7 0 7 7 0 7 0 0 7 0 7 0 0 7 0 7 7 0 0 0
0 7 0 7 7 0 7 0 0 7 0 7 0 0 7 0 7 7 0 0 0 0 0 0
7 7 0 7 0 0 7 0 7 0 0 7 0 7 7 0 0 0 0 0 0 0 0 0
7 7 7 7 7 7 0 7 0 7 7 0 7 0 0 7 0 7 0 0 7 0 7 7
7 7 7 0 7 0 7 7 0 7 0 0 7 0 7 0 0 7 0 7 7 7 7 7
0 7 0 7 7 0 7 0 0 7 0 7 0 0 7 0 7 7 7 7 7 7 7 7
7 7 0 7 0 0 7 0 7 0 0 7 0 7 7 7 7 7 7 7 7 7 7 7
```

[2] If an advanced vulnerability only uses a single basic vulnerability, the same example may be used by both.

The above plain PPM file looks like this. Do note that because of the very small size, your image viewer may blur the pixels together to produce blobs of colour, rather than displaying them square as below.



For further information, please visit `http://netpbm.sourceforge.net/doc/ppm.html`. Note that this link describes TWO versions of PPM. The format we are expecting you to read and write is the "Plain PPM" format. (NOT the binary format.) Your program should properly handle input and output files of up to 10,000 pixels in width and/or height.

Here are examples of image viewers that we have tested and know work with Plain PPM files. There will of course be many others.

- Windows: IrfanView
- Linux: GIMP
- Mac: Preview

Writing the input and output code for this format forms part of the assignment specification and doing so correctly is a requirement of passing this assignment and the unit. Please ensure that you test your program.

4.4 How we will run your program

Your program should be self contained and rely on nothing from outside the archive you submit (apart from the compiler of course). In particular, make sure nothing in your submission attempts to link to your home directory (or any other directory outside your submitted archive). We will be compiling your program in the following manner. We will be compiling and testing your program using a Linux computer with a similar setup to the computers in the labs.

```
make
```

Hint: If some of your vulnerabilities (e.g. code injection) require compiler switches to override default behaviour, such as `-fno-stack-protector` or `-Wa,--execstack`, make sure these are in your Makefile.

To check that your basic program works, we will first run your program with the commands listed in Section 2.1. Therefore, make sure that your Makefile produces a correctly named executable. Your program must finish running each command within 1 minute.

We will then test each vulnerability with the "normal" and "exploit" commands and input files that you specify in your report. Note that if a particular exploit requires ASLR to be disabled, you must specify this in the corresponding command line(s) in your report.

4.5 Use of External Code and Resources

Apart from normally present system libraries and the code fragments provided in this assignment specification, you should cite all use of code and libraries that you did not write yourself. If they are deemed to have significantly simplified the problem by using code or libraries that you did not write, and you cite them correctly, this may result in a mark of zero for the sections that rely on that code. **If you do NOT properly cite the use of external code you will receive zero for the entire assignment. Proceedings may be initiated against you for plagiarism. See the Curtin University plagiarism policy for further details.**

5 Marks Breakdown

- Coding
  - Good commenting: 10 marks
  - Clarity: 5 marks
  - Consistency: 5 marks
- Basic Vulnerabilities (present, trigger with the appropriate input and properly described in the report):
  - Stack buffer overflows (x2): 5 marks each, 10 marks total
  - Heap overflows (x2): 5 marks each, 10 marks total
  - Format string vulnerabilities (x2): 5 marks each, 10 marks total
  - Memory leaks (x3): 2 marks each, 6 marks total
  - Memory access-after-free: 5 marks
  - Memory double-free: 5 marks
- Advanced Vulnerabilities (present, trigger with the appropriate input and properly described in the report):
  - Memory Access: 12 marks
  - Code Injection: 12 marks
- Report presentation/clarity (assuming at least one of the above works): 10 marks
- Penalties
  - Missing or faulty operations on the provided example inputs (note that missing operations will also make it harder for you to demonstrate the vulnerabilities above): Up to -10 for each missing or faulty operation
  - Additional Vulnerabilities: Up to -10 per extra vulnerability
  - Invalid (artificial) Vulnerabilities: Up to -10 per fake vulnerability
- **Total: 100 marks**