CSci 4271W: Development of Secure Software Systems

Programming Project **due: April 30th, 2021**

Ground Rules. This is an individual assignment that each student should complete on their own. It's OK to help other students with understanding the concepts behind what we're doing in the project, or to help with technical difficulties, especially if you do so in venues like Piazza and office hours where the course staff are also present. But don't spoil the assignment for other students by telling them the locations of vulnerabilities or details of how to exploit them: everyone should have the experience of figuring those out for themselves. The assignment is due online on Friday, April 30th. The submission will be accessible from the course Canvas page, and the deadline time will be 11:59pm Central Time. You also have the opportunity to make an early submission on or before Friday, April 23rd, for feedback on writing and presentation (but such an early submission will not be graded). You may use external written sources to help with this assignment, such as books or web pages, but don't get interactive help with the assignment from outside sources. You **must** explicitly reference any external sources (i.e., other than the lecture notes, class readings, and course staff) from which you get substantial ideas about your solution.

The Program. The latest product of Badly Coded, Inc., that you are working with is named the Badly Coded Image Viewer, or `bcimgview` for short. It provides the functionality of viewing bitmap images in a GUI or converting them to a different format, but it is distinguished from the many pre-existing image viewer programs by working solely on several image file formats designed by Badly Coded and produced by other Badly Coded products such as the Badly Coded scanner interface and the Badly Coded plotting tool (not yet released).

Bcimgview runs on Linux (and in the future perhaps other compatible Unix variants), and is distributed as a x86-64 binary that links with the GTK 3 family of GUI libraries. It has functionality to recognize images in any of three different Badly Coded formats (distinguished by the first few bytes of the file called a "magic number"), and to parse and decompress the images into an internal format that includes a struct containing metadata and a packed array of pixels, each represented by one-byte red, green, and blue samples (sometimes called 24-bit per pixel true/direct color). For display, this internal format is further converted into a GDK Pixbuf object which can be displayed by the GTK library in a GtkImage widget. The program can take the name of an image file as a command line argument, or the GUI interface has a button to bring up a standard GTK file chooser dialog box to select another image to view. When the command-line option `-c` is supplied, the bcimgview binary will instead operate in a batch conversion mode, converting an input image in one of the Badly Coded formats into an output file in the PPM (Portable PixMap) format. (PPM is supported by several Linux applications including the GIMP image editor and the `eog` image viewer.) Both the GUI display and command-line conversion modes will also print a log message in a configurable format about the image being displayed.

Bcimgview supports three bitmap image formats:

- BC-Raw, with the file extension `.bcraw`, is an uncompressed true color format similar to bcimgview's internal representation.

- BC-Progressive, extension `.bcprog`, is intended for transmitting images over low-bandwidth network connections, like web sites in the 1990s. Colors are represented with single bytes from a 6x6x6 color cube, and scanlines are sent in an interleaved order.
- BC-Flat, extension `.bcflat`, is a losslessly compressed true color format that separately encodes the red, green, and blue channels. The differences between adjacent samples in a scan line are computed and then one or more differences are encoded with a fixed dictionary of variable-length binary codewords.

The source code for `bcimgview`, some sample images in the supported formats, and a pre-compiled Linux x86-64 binary of the most recent release are available to download from the course's public web page. The program should work on most recent Linux systems, though the supported configuration is to run it on CSE Labs Ubuntu 20.04 installations like VOLE and the lab workstations.

If you want to use the GUI interface and see what the sample pictures look like, you'll need to run `bcimgview` together with the "X11" graphical environment that is common on Unix systems. This will happen automatically if you run it on VOLE or when physically logged in to a lab workstation. If you connect to a CSE Labs machine via SSH from another Unix machine with a graphical interface, you can use the SSH feature called X forwarding to display the GUI on your local computer. You can also use X forwarding to a Mac or Windows computer, but you'd also need to run a separate X server on your local computer to make this work.

Please use our supplied binary for the purpose of trying out attacks. This binary has been compiled in a way to disable defenses against certain attacks, and using the exact same binary makes the results more consistent. You may not be able to receive full credit if you describe an attack that doesn't work with our supplied binary on a CSE Labs Ubuntu machine.

Specifically, the command we used to compile the `bcimgview` binary was:

```
gcc -no-pie -fno-stack-protector -Og -g -Wall \
    $(pkg-config --cflags gtk+-3.0) \
    bcimgview.c -o bcimgview \
    -lgtk-3 -lgobject-2.0 -lglib-2.0 -lgdk_pixbuf-2.0
```

(The backlashes at the ends of lines represent line continuation; you won't need them if you copy the whole command onto one line.)

Your Job. For this project, you will take on a series of tasks helping the Badly Coded developers with assessing and improving the security of `bcimgview`. Specifically, you'll perform threat modeling, audit the code for vulnerabilities, and test how the vulnerabilities can be exploited; your submission will be a written report describing the results of these activities. The next sections describe these tasks in more detail.

Threat Modeling. For threat modeling, you will analyze the architecture of the `bcimgview` program, such as by describing data-flow and trust relationships, to determine which security threats might be a risk against the program. Your threat modeling should include at least one data-flow diagram showing the architecture and a text description of the parts of the architecture, the flows between them, and which threats are possible given the data flows.

Your diagram should include more detail than having just one box representing the `bcimgview` program. Instead you should show some of the internal functional structure of the program: which parts of the functionality involve computation on different kinds of data? The purpose of this level of detail is to guide thinking your about threats.

It is fine to include threats in your threat modeling for which you are unsure how serious a problem they are, such depending on the context in which the program is used or what data might be under the control of an adversary. It's better to be aware of a threat that turns out to not be a serious problem to reduce the risk of missing one that does. But you may wish to highlight the risks that seem most serious.

One benefit of good threat modeling is to inform the auditing and bug-fixing processes, but you aren't restricted to doing the project completely in this order. If you find a vulnerability in a later stage, you should go back and check that it corresponds to a threat you had identified, or add a new threat if one corresponding to the vulnerability was missing.

Code Auditing. The next step of the project is to look for bugs in the `bcimgview` source code that might be a problem for security. Your auditing should be informed in part by your threat model: for instance, which parts of the code are more likely to be vulnerable because they are exposed to untrusted inputs? You should also think about the different kinds of memory-safety vulnerabilities we discussed in class, which are the focus of this project and an important danger for a program written in C. For a vulnerability to be exploitable, there needs to be the combination of a bug with a situation where the bug can be triggered or controlled by an attacker-controlled value.

You should briefly describe the process you used for auditing, but the main result of the auditing in your report should be description of the vulnerabilities you found. For each vulnerability, describe what the original programming mistake was, how it leads to an unsafe situation, and how that unsafe situation might be controlled by an adversary. You don't need to give every detail of a possible attack here (that's the next section), but describe the adversarial control in a general way to explain why this is a security problem and not just a non-security bug.

You can also include in the results of your audit other places in the code that looked like they might be dangerous from a security perspective, but where you aren't sure that they are vulnerable. Usually this will either be because something else in the code currently prevents an attack, so you are confident it is not currently vulnerable, or because the conditions are so complex that it is not clear whether an attack is possible. These other problem areas might still be useful suggestions for the developers to improve, even if they are lower priority than bugs that are known to be exploitable right now.

The `bcimgview` code contains at least four different security vulnerabilities that we placed intentionally, and others might exist unintentionally. For full credit, your auditing results should describe at least three of these vulnerabilities.

Creating Attacks. The last hands-on task will be to demonstrate vulnerabilities by constructing working attacks. Because `bcimgview` has memory-safety vulnerabilities, the goal for your attacks should be to take control of the execution of the program.

To make things more uniform and so that you don't have to write your own shellcode, we have implemented a special function in the `bcimgview` code just for the purposes of your attacks. The function named `shellcode_target` exists in the code of `bcimgview`, but it is never called during normal execution. The only this function can execute is if an attack changes the execution of the program to go to a location chosen by the attacker, and that what you should do to demonstrate that your attack technique is working. (Generally if you have an attack that works to call `shellcode_target`, you could also modify it to execute any other code of the attacker's choosing, like injected shellcode or a ROP chain; we're just not asking you to do that later stage of the attack.)

For this project, we won't be using any machine-checked way of verifying that your attacks work. Instead, your need to describe the attacks in enough detail in the text of your report to convince a reader of the report that you carried out the attack successfully. So this part of the assignment includes writing clear and accurate technical description in addition to discovering the attack in the first place. Your description of the attack should also show that you correctly understand why it works; it shouldn't sound like something you discovered accidentally without understanding it. (Of course it's OK if you originally find an attack accidentally, as long as you understand it eventually.)

Because the binary-format image files are one of the main untrusted inputs to `bcimgview`, it is likely that at least some of your attacks will involve maliciously-crafted images files which are illegal or unexpected by the code in some way. One technique we'd recommend for explaining an attack like this is to show the relevant part of the attack input in a binary-oriented format like a hex dump, and to highlight the parts of the file contents that are involved in the attack.

As with the auditing step, you will need to demonstrate successful attacks against at least three different vulnerabilities for full credit; there are at least four usable vulnerabilities in the code.

Written Report. Your submission, due Friday April 30th, will be a written report describing your threat modeling, code auditing, and attacks. Your report should be 4-6 pages long, formatted for US-standard "Letter" paper (8.5 by 11 inches) with one-inch margins. The main text of your report should use a Times, Times Roman, or Computer Modern Roman font, 10 points high, and double spaced. (By comparison, these instructions use single-spaced 12 point Computer Modern Roman on letter paper with one-inch margins, so your document should take up the same area of the page, but should have a smaller font with more space between the lines.) The expectation of 4-6 pages refers to the text of you report. Your report should probably also include some figures, but you should put them at the end, after the 4-6 pages of main text, and they may take up more pages. Your report should be labeled with your name and UMN email address.

Writing is part of the purpose of this assignment and about half of what you will be graded on, so be sure to allow time for quality writing, including revising, checking spelling and grammar, and so on. You should write in a relatively formal style like a report you were writing in business, but your priority should be explaining your technical points clearly. Don't make jokes or opinionated comments about the topic, and your approach doesn't need to be primarily arguing for any particular position. Instead your approach should be to inform readers about the security of the software in an objective-sounding way, providing the facts they need to do their job (e.g., to fix bugs or withdraw the product until it can be improved).

Other Suggestions. Binary formatted files like the image files in this project can't be easily edited with a normal text editor. To create binary files for testing or attacking bcimgview, you will instead either want to write small programs that generate a binary file (such as using binary I/O in C, pack in Perl, or similar features in other languages), or using an editor designed for binary files. Editors designed for binary files are conventionally called "hex editors", because they are based around displaying each byte of the file in hexadecimal, like an interactive version of the hex dump program hd. If you have a favorite programmer-oriented text editor, it may already have a hex editing mode available. If you're using the CSE Labs machines, ghex is the name of a simple standalone hex editor that's available.

Check out the class's Piazza page for more Q&A and suggestions about the project. If you can ask a question without spoilers, please ask in a public post (it can still be anonymous if you'd prefer) so that everyone can contribute answers and benefit from them. If you are worried the question might be a spoiler, use a private post (we might request that you make it public later if we think it's not a spoiler).