

CSci 4271W: Development of Secure Software Systems

Project 1

due: March 29th and April 19th, 2024

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. There will be two rounds of submission for the project: the first on Friday, March 29th, and the other on Friday, April 19th. Both submissions will be online, accessible from the course Canvas page, and the deadline time will be 11:59pm Central Time. (If you are prone to putting things off until the last minute, be aware that Canvas is a computer program that interprets deadlines strictly, and that 11:59pm is 60 seconds *before* midnight.) 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 human sources. You can also use software tools, including AI systems, to try to help with understanding the code, finding bugs, or assisting with the writing process. 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, and there will be a section of the submission to describe your use of tools.

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. This internal format includes a struct containing metadata and an 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 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 and to PPM.
- 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 22.04 installations like VOLE-FX3 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 -lm
```

(The backslashes 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`. For the first submission, 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. For the second submission, due three weeks later, you'll revise your report in response to our feedback, and provide fixes for the bugs in the program that made the attacks possible. 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.

Your report should first 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 use tools to assist you in the auditing process, but you need to correctly understand the vulnerabilities you have found, which cannot be automated.

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. If your submission describes more than three vulnerabilities, your grade will be primarily based in the three best vulnerabilities, but you can also get credit for information you provide about further vulnerabilities to compensate for areas where you lost points on the best three.

Creating Attacks. The last task for your first submission 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 way 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 is 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 initial submission, due Friday March 29th, will be a written report describing your threat modeling, code auditing, and attacks. Your report should be 4–5 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 4–5 pages refers to the text of your report. Your report should probably also include some figures, but you should put them at the end, after the 4–5 pages of main text, and they will likely take up more pages. Your report should be labeled with your name and UMN email address at the top of the first page.

Writing is a major 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).

Fixing The Code. For your second submission, due April 19th, one of your two tasks will be to fix the security bugs you found in `bcingview`. You should do this by modifying the source code and recompiling it, and verifying that the normal functionality of the program still works correctly but that areas of the code that used to allow attacks are now safe. (Note that when you recompile the program, address like the address of `shellcode_target` will probably change. But that isn't a substantial security improvement on its own because an attack could be easily modified to use the new address.) Between the initial and second submissions, we'll post some more information on Piazza about the bugs that have been found in `bcingview`, so you can fix them even if you missed finding them the first time.

The goal of your fixes should be to make the code secure. Your changes should be sufficient so that the security bugs that previously existed are completely eliminated, but it is not important to make non-security improvements to the code: you should think of this as software maintenance, not rewriting. In some cases, a change that involves refactoring the code might be easier to see the security of than a small change with a subtle security argument. You should weigh this sort of tradeoff based on the assumption that your grade will be primarily based on whether the result of your fix is secure. In other words, it is in your interest to make the simplest change whose security you are confident of.

Your second (i.e., revised) written report, described in more detail below, should describe the ideas behind your fixes: how the new code is different from the old code and how it avoids a security problem. Your general goal in making a security fix is to keep the normal behavior of the program the same, but sometimes people might differ as to what behavior is "normal," or some change to functionality might be unavoidable. In these latter cases you should justify the decisions you made.

The grading expectations around fixes are similar to those around vulnerabilities and attacks. There are at least four vulnerabilities available to fix, but you only need to have three good fixes for full credit, and your score will primarily be based on the three best fixes you submit. Any further fixes you include can improve you grade as partial credit if they make up for deficiencies in your three best fixes.

Your change should also maintain at least as high a level of readability in terms of things like variable names and comments as the code you are modifying. You don't have to put as much information into comments as you put in your written report, but if there is some important information in a comment that would be helpful to future code maintainers, that would be a good addition to your fix. If the original code had incorrect information in a comment that was related to the vulnerability, you should correct the comment so that it doesn't mislead future developers.

You will submit your code changes in the form of a patch, a text file that shows just

the differences between the old version of the code and your fixed version. You may have seen patches when working with Git or another version control system, but it's up to you whether or not you use version control in this project. You can create your patch by using the command `diff -up` on a Linux machine; the arguments to the command are the old and new source code files. The primary purpose of the patch is to be readable by another developer, so you should check that the patch doesn't contain extraneous changes like changing whitespace or indentation. (If your patch looks like every line in the file has been changed, you have definitely done something wrong in generating it.)

Revised Report. The other part of your second submission will be a revised version of the report you submitted in the first round. We will have given you feedback on your initial submission, covering both the technical aspects of your report and the writing. The revision is your chance to improve these aspects of your report. For instance, this might include describing additional vulnerabilities that you missed the first time, improving the vulnerability descriptions or threat model discussion, or just making your writing clearer and grammatically correct. You should also devote some space in your revised report to describe how you fixed the vulnerabilities you identified (as mentioned above). This could either be a part of the discussion of each vulnerability, or a separate section.

A portion of your grade for the second submission will be for improvements to parts of the report you had first drafted in earlier submissions. These won't replace the corresponding parts of your score from previous submissions, but they are a chance to improve your overall score because your revised version will presumably be better than the original.

Because some of the changes in your revision might be best addressed by adding more material to your report, the length for your revised report can increase from 4–5 pages to 4–6 pages.

We will also provide a script that provides a feature analogous to `diff`, but for PDF documents, for you to create a PDF that highlights the changes between your original and revised versions. Like with the source-code patch submission, please look at this output and make sure it matches the changes you believe that you made. There will be three separate Canvas assignment entries for the second round: one for the patch file, one for your revised PDF, and one for the differences PDF.

Tool usage. You are allowed to use a variety of software and/or AI tools to assist you in the project. One way you might use tools is to help you audit the code, such as static bug-finding tools, fuzzers or other testing tools, or asking questions about the code of an AI chat system. The kind of tool that students have found most helpful in previous semesters is a fuzzer like AFL, for finding crashing inputs that can be leads towards vulnerabilities. However you will generally still need to understand what is going wrong in a crash in order to exploit it, so the fuzzer is just the start of a longer process. Based on our testing of such tools, we aren't confident that static analysis or AI tools work well enough on problems like this assignment to be worth your time. However you are welcome to try using any such tools; you might get lucky or inspired by some interaction with them, and familiarity with them may be helpful to you in other contexts.

You can also use software and/or AI tools to assist with the writing aspects of the project. At a minimum, you should probably be using a spell checker to catch typos, but there are lots of more sophisticated tools that can help catch other writing mistakes like grammar or style, or even claim to rewrite your text for you. However these tools are still far from

perfect: so we wouldn't recommend applying any AI writing tools blindly. It is probably better to consider such tools as providing suggestions, and to make the final decision yourself about whether to accept the suggestion.

As a final appendix to your report (a separate last page), write between one sentence and one page describing the software and/or AI tools that assisted you in any of the phases of the project. If you chose not to use any special tools, this could be a single sentence like "The only tools I used were GDB and the spellchecker in Google Docs." But we would like to hear more details if you found other tools to be useful. We won't reduce the credit you get for the body of the report if you say, to take an extreme example, that you were able to get a large language model to do the whole project for you by using the right prompts. But we are interested in these information to inform how we adjust the project for future versions of the course.

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. (Vim, Emacs, and VSCode all do.) If you're using the CSE Labs machines, `ghex` is our recommendation of a simple standalone hex editor that's easy to start using.

Check out the class's Piazza page for more Q&A and suggestions about the project. In particular it's the best place to ask questions. 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).