CSci 4271W: Development of Secure Software Systems

Project 0.5 **due: Wednesday, April 5th, 2023**

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. This project will be due on Wednesday, April 5th. The submissions will be online, using the course Canvas page, and the deadline time will be 11:59pm Central Time. 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 BASIC interpreter, or `bcbasic` for short. It implements a simple programming language, a non-standard dialect of the BASIC language family. BASIC was invented in the 1960s as one of the first languages designed to be easy for beginners to use, and it was very common on early microcomputers in the 70s and 80s. The management's future plans for BCBASIC involve using it as an extension language for other applications, similarly to how Microsoft Office uses Visual Basic for Applications, Emacs uses Emacs Lisp, web browsers use JavaScript, and so on. However for now it has just been implemented as a standalone interpreter program which reads an executes a program file supplied as a command-line argument.

Similarly to the use of JavaScript in web browsers, the design goal is that BCBASIC should be a safe programming language: memory is managed automatically, and even a maliciously-written program can't escape its sandbox or cause the interpreter to crash. Your job for this assignment will be to check how well the Badly Coded developers have done at this task. You will audit the code to look for problems, test whether any potential vulnerabilities can be exploited, and write up your results in a form that the developers can use to improve the software in the future.

This project is numbered 0.5 because it is intended in part as a warm-up for project 1. Project 1 will also have a buggy program to audit and attack, but project's 1 has more independent pieces, multiple binary input formats, and requires a wider variety of attack techniques. Project 1 will also have extra requirements such as threat modeling and repairing vulnerabilities. This project concentrates on just auditing and attack construction in a more circumscribed context, since we've observed these can be the most challenging new skills in security.

BCBASIC is a limited dialect of BASIC, mixing some features of historical BASICs with some more modern decisions. It only supports two general data types: integers (64 bit, signed), and arrays of integers. (Strings also exist, but they can only be printed.) BCBASIC has a reasonable set of arithmetic operators, but its syntax doesn't allow complex expressions.

For instance a single line of code can only do one arithmetic operation. Most places in the syntax support only a "simple expression" that can be an integer constant (decimal only) or an integer variable. BCBASIC's arrays do have some nice features (like Perl and Python): they automatically grow as needed, and can be indexed backwards with negative indexes.

One aspect of BCBASIC that is archaic now is its control flow: BCBASIC has only "unstructured" control-flow, in contrast to the structured control flow of if statements and loops standard in almost all modern languages. The only control flow operations in BCBASIC are unconditional and conditional versions of the "goto" statement, which can transfer control to any numbered line. In fact it might be that the only language you've studied with control flow like BCBASIC's is assembly language. There are also no subroutines or functions. You do not have to write line numbers explicitly, but a good practice is to write line numbers on all the lines that will be targets of goto, and to choose a sparse numbering so that you don't have to renumber when if you insert new statements later. We've included some example BCBASIC programs you can read to get a feel for how the language works.

When you get to the point of trying out attacks, please use our supplied binary for doing so. 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. Though you should be able to understand vulnerabilities at the source code level, you will have the easiest time convincing us of the vulnerabilities you have found if they work the same way as our CSE Labs Ubuntu reference systems.

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

```
gcc -no-pie -fno-stack-protector -Wall -g -Og bcbasic.c -o bcbasic
```

Your Job. For this project, you will help the Badly Coded developers with assessing and improving the security of `bcbasic`. Specifically, you will audit the code to look for memory safety vulnerabilities. Then you will test how the vulnerabilit(y/ies) you have found can be exploited. Those you'll do these tasks in front of the computer, the end result will be a written report describing your findings. The next sections describe these tasks in more detail.

Code Auditing. The next step of the project is to look for bugs in the `bcbasic` source code that might be a problem for security. The threat model for this assignment is intentionally circumscribed. The potentially malicious input is the BCBASIC program supplied to the interpreter (in fact BCBASIC programs can't themselves take other input). The attacker's goal is to take over the interpreter program with a control-flow hijacking attack. In the testing context, this hijacking doesn't make any further damage possible because the interpreter program doesn't have special privileges. But you should imagine that BCBASIC is being prepared to be used in a context like JavaScript in a web browser, where the program will be untrusted and control-flow hijacking would allow other malicious actions.

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. Some kinds of vulnerabilities are more applicable to this program than others, and some kinds of risky operations might be more prevalent in certain areas of the code. While it's helpful to understand at a high level what all the code in the program does, you can probably make your auditing more productive by prioritizing particular sections of the code and vulnerability classes. A portion of your report should be a description of the processes and techniques you used in your auditing, but the details of the process aren't as important as the results.

The main result of the auditing in your report should be description of the vulnerabilit(y/ies) 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.

One of the ways we have made this project simpler than the next one is that we have one included one intentionally placed vulnerability in the `bcbasic` source code (though of course there may be other bugs that exist unintentionally). Given this, you only need to find (and later attack) one vulnerability for the project If you have found a vulnerability and confirmed that it can be attacked, it probably means you have found the right one, and you should concentrate your report on it. You are allowed to describe multiple possible vulnerabilities if you have found them. And if you have multiple leads but are not confident in any of them, it could be a strategic choice to mention several to increase the chances that one of them is the right one, since you'll be able accrue partial credit across multiple descriptions. But it is a better strategy, if you can do it, to concentrate on vulnerabilities that you have confirmed are clearly exploitable security problems that you can see lead to control-flow hijacking. You can get full credit by describing just one such vulnerability. Our intent was to make the one vulnerability we inserted intentionally clearly incorrect and exploitable, so looking for a vulnerability with those features would be the recommended strategy. However you also still get full credit if you different (unintended) vulnerability which is also exploitable.

Creating Attack(s). The second task for your submission will be to demonstrate the vulnerabilit(y/ies) you discovered by constructing working attack(s). Because `bcbasic` 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 `bcbasic` code just for the purposes of your attacks. The function named `shellcode_target` exists in the code of `bcbasic`, 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 untrusted input to BCBASIC is a BCBASIC program, your attack will likely take the form of a (perhaps partially illegal or nonsensical) BCBASIC program. We would recommend including the full text of the attack program as a figure in your report and explaining what it does by referring to line numbers.

You may recall experiencing in lab that low-level attacks that depend on the memory layout of a program can behave differently based on the program's memory layout. You may find this applies to your attacks against BCBASIC, as well. Since these challenges aren't the main focus of the project, it is enough if you can demonstrate your attack working for a single memory layout of the program, such as with ASLR completely disabled or when the program is running under GDB, as long as you can explain why this is not a fundamental limitation. For instance you can explain how you chose parts of the attack based on the memory layout, even if this process wasn't automated. However if you're feeling ambitious to create a very flexible attack, it is possible to build a reliable attack against the vulnerability we introduced into the program. When using the `bcbasic` binary we distribute, the attack works outside of GDB, and when the stack and heap locations are randomized with ASLR.

Written Report. Your results in the project will be submitted in the form of a written report including 2-4 pages of text. The report should be 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 2-4 page length expectation refers to the text of you report. Your report should probably also include at least one figure, but you should put any figures at the end, after the main text, so that the length of your main text is clear at a glance. (This will require readers to flip back and forth a bit, but we're willing to accept this slight hassle for uniformity.) There is no maximum length limitation for the pages used by figures, but don't include more extra information than would be useful to readers.

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).

The report submission for this project will be due on Wednesday, April 5th, by 11:59pm Central Time. Submit the report as a PDF using the Canvas site.

Other Suggestions. 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).