CSci 4271W: Development of Secure Software Systems

Project 0.5 **due: Friday, February 23rd, 2024**

Ground Rules. This is partially a group project: you are recommended to do the code auditing and attack creation in groups of up to 3 students, but each student must individually write a 2–4 page report. It's OK to help other students outside your group 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 outside your group 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 Friday, February 23rd. 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 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 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 and 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 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. Also while the code auditing and attacking in this project are done in a group, Project 1 will require you to do similar tasks individually. So you should think about how to structure your collaboration so that everyone in the group can practice and improve these skills.

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. On the other hand, BCBASIC's arrays 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 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 22.04 reference systems, like the lab workstations.

Specifically, the command we used to compile the `bcbasic` 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. Though 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 first step of the project is to look for bugs in the `bcbasic` source code that might be a problem for security. You are allowed and in fact recommended to this step as a group. 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.

Compared to the simpler examples we've been seeing in labs, the vulnerabilities in this software occur in the code in a more natural way, like other kinds of bugs. This will make the process of finding them more challenging: they are subtle enough that the original programmers missed them, even though they thought about the design of the code and tested it. BCBASIC is a bit more than 1000 lines of code, so while it is not as complicated as a lot of real world programs, you will probably find it takes time to get to know the code. Even professionals would probably spend a number of hours looking at this code in order to audit it thoroughly.

The main result of the auditing in your report should be a 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 is that we only expect you to find (and later attack) one vulnerability. (However we are aware of at least three vulnerabilities in the code: one was created completely intentionally, and the other two started out as unintentional.) If you have found a vulnerability and confirmed that it can be attacked, it probably means you have found one of the intended ones, and you should concentrate your report on it. (A good, but not perfect, thing to look for early in testing is whether you can at least make the program crash, like with a segfault. Many situations that cause a segfault can be used for an attack. Whereas if you have an idea about a potential problem but can't even find a situation that leads to a segfault, it's less likely to be what we're looking for.) 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 vulnerabilities we're aware of intentionally clearly incorrect and exploitable, so looking for a vulnerability with those features would be the recommended strategy. However you can also still get full credit if you discover a 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 a working attack. Because `bcbasic` has memory-safety vulnerabilities, the goal for your attacks should be to take control of the execution of the program. We recommend you work in a group for this phase of the project as well, which should be the same group as the previous phase.

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, you will 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 aspect of the assignment will include 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 any of the known vulnerabilities in the program. When using the `bcbasic` binary we distribute (which is not PIE), there are attacks that work 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; every student must independently write a separate report. "Independently" means that after finishing the first two phases of the project, you shouldn't help the students who are your group members with the actual writing process. For instance, it's okay to keep talking about the vulnerability or the attack process if you realize you didn't understand something about them. But it wouldn't be appropriate to share a draft of a paragraph from your report, even with another member of the group, or to have another group member (or in fact any other person) proof-read your report for you.

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 main text of your 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, as well as the names and addresses of the other members of your group.

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 Friday, February 23rd, by 11:59pm Central Time. Submit the report as a PDF using the Canvas site.

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. However based on our testing of static analysis AI tools, we aren't confident that any such tools works well enough on problems like this assignment to be worth your time, because they take effort to get started using, and the results can be of limited value because of errors. The one kind of tool that a majority of students have found helpful in previous project is a fuzzer that automatically searches for crashing test cases; we will introduce one such tool, AFL++, in the 2/12 lab. However an important limitation of fuzzing to keep in mind is that it will not explain the problem in the code, just give you an example input that triggers it. Understanding the vulnerability is important part of the project, and something you will still need to do even if you have a test case. You are welcome to try using any software tool in the project; you might get lucky or inspired by some interaction with them, and familiarity with them may be helpful to you in other contexts. But our recommendation is that the task of understanding and auditing the code doesn't need complex tools beyond reading the code and testing the code with the help of a debugger.

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. 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).