CSci 5271: Introduction to Computer Security

Hands-on Assignment 1 **due: September 15th - October 13th, 2017**

Ground Rules. You may choose to complete this homework in a group of up to three people; working in a group is not required, but strongly recommended. If you work in a group, only one group member should submit a solution each week, listing the names of all the group members. You'll be submitting answers once a week, by 11:55pm Central Time on Fridays running from September 15th through October 13th. Each week, one member of your group should use the appropriate Moodle activity to upload a tarred and gzipped directory containing all the files mentioned as required for that week. You may use any written source you can find to help with this assignment, on paper or the Internet, but you **must** explicitly reference any sources other than the lecture notes, assigned readings, and course staff.

Hackxploitation. This homework involves finding many different ways to exploit a poorly-written program that runs as root, to "escalate privileges" from a normal user to the super-user. The program we will exploit is the Badly Coded Text Editor, BCVI (it has that name because it is loosely modelled on the classic Unix text editor `vi`.). (The same company produced the Badly Coded Versioning System, Badly Coded Print Server, and Badly Coded File Archiver used in previous years' 5271s.) You can download the source code for BCVI from the course web page near where you got this assignment. Normal `bcvi` runs as the user who invokes it, but the security vulnerabilities are a problem because there is also a mode of running the editor as `sudobcvi`, somewhat like the common Linux program `sudoedit`, which allows users to edit files they would otherwise not have permission to edit, using a setuid-root binary. Normal `bcvi` can be used to edit any single file that you have both read and write access to (if you don't have read access, you can't open it, and if you don't have write access, you won't be able to save changes). By comparison `sudobcvi` can be used to edit text files either that you have access to though normal mechanisms, or that you have access to via a special configuration file `/etc/sudobcvi.conf`. For instance the file `/etc/issue` containing a message to be printed when users log in is owned and writeable only by root, but we have configured the VMs so that the `student` and `test` users can also edit it, which normally shouldn't be a security problem.

Because BCVI is intended to run as root, and breaking it lets you get root, we can't have you doing so directly on a CSE Labs machine. Instead, we will provide each group with a setup to run a virtual machine, and you will have root access (e.g. using the `sudo` command) inside the VM. The VMs will run on a CSE Labs cluster: we'll provide more information about running them once they're available. You can start looking for vulnerabilities in the BCVI source, and even testing many kinds of attacks, without a VM: BCVI can run in a location like your home directory, and you can test attacks that culminate in getting a shell: it will just be a shell with your own UID, rather than a root shell. You can also recompile it if you'd like, but don't run `make install` on a real machine.

BCVI has a limited set of features compared to the real `vi`, much less enhanced variants like Vim. You can only edit one file in a session; the total size and the number of lines in the

file are limited, and BCVI doesn't deal well with lines that are longer than 80 columns. In the initial command mode, you can move the cursor around the file using the arrow keys, or the letter keys `h`, `j`, `k`, and `l`. You can jump to the beginning or end of the current line with `^` or `$` respectively. You can delete the character before or after the insertion point with `X` or `x`. To add new text, you switch to inset mode with the command `i`; in insert mode, the characters you type are inserted into the file, until you type Escape to go back to command mode. There is also another set of command-line-style commands which for historical reasons are called "ex mode": they all start with a colon (`:`) and go up to the end of the line. The command `:w` saves the current file, and the commands `:q` or `:q!` quit the editor. The command `:%!` lets you transform the contents of the buffer through an external program; for instance `:%!expand` will expand tab characters into spaces, and `:%!tr a-z A-Z` will convert the file to all uppercase. Another slightly unusual feature you'll notice is that newline and tab characters are represented explicitly as reverse-video `n` and `t` characters respectively. A similar representation is also used for less common control characters. You'll probably just use BCVI with ASCII text, but in its full generality, it can edit files in the Windows-1252 character set (which effectively includes both ASCII and ISO 8859-1 as subsets), but to display everything correctly it runs best on a Unicode-capable terminal. In a pinch you can even use BCVI like a hex editor, using the `:ih` command to insert a character based on its hex code; for instance `:ihA9` inserts a copyright sign.

Another feature that any self-respecting text editor has is expandability via a Turing-complete macro language: for instance Emacs has Emacs LISP, and Vim has its own scripting language as well as supporting several common ones. Since Badly Coded was trying to keep BCVI from getting too complicated, they sacrificed some readability for compactness by allowing macros in BCVI to be written directly in machine code. You can use the command `R` to execute the code after the insertion point as a function operating on the four bytes before the insertion point (passed in `%eax`). Of course the length of the macro and the instructions it can use are limited for security reasons.

Because BCVI is a full-screen text editor, it relies on the Curses (specifically, `ncurses`) library to manage putting text on the screen. In its source code you'll see calls to Curses functions like `getch` and `mvaddch` that you may not have seen in other C code. The VMs have the Curses manual pages installed, so you can just say `man getch` to read about Curses functions. You can also read `ncurses` documentation on the web[1]. Because of the way a Curses program takes over the whole screen, it is inconvenient to run it directly under a debugger. Instead it will work better to run BCVI in one window, GDB in another, and use gdb's `attach` command. This is possible on the VMs, though it is disabled for security reasons (unless GDB is running as root) on many other Linux machines. You can get a similar effect, in a somewhat more cumbersome way, using `gdbserver`.

Another side-effect of BCVI being a Curses application is that if BCVI crashes in an uncontrolled way, it may leave your terminal in an unusual state that makes other programs work strangely. For instance, BCVI disables the automatic insertion of carriage returns on terminal output, so in the output of normal batch programs like `ls`, you will see each new line

[1] `http://invisible-island.net/ncurses/man/ncurses.3x.html`

starting at the column position after the end of the previous line, instead of at the left side of the terminal. You can use the `stty` command to manually control the terminal settings of your terminal if this happens. Specifically the command `stty sane` is a convenient one to remember to return the terminal to its default settings. You might also find manipulating other terminal characteristics with `stty` to be useful in carrying out attacks.

BCVI is intentionally sloppy code; please never copy or use this code anywhere else! It is so sloppy that when run as root, it is full of ways that allow someone who sends the right commands to become root. The main part of the assignment is for you to find four or more ways to get a running command shell with UID 0 as a result of sloppy coding and/or design in BCVI. Another way of classifying the vulnerabilities is that some of them are logic errors or problems with the program's interaction with the operating system (for instance these would arise in just the same way if the program were written in Java), while others are related to the unsafe low-level nature of C which lead to control-flow hijacking.

There's another aspect of BCVI's implementation that merits a bit more discussion. To efficiently store the contents of the file being editing even as you make changes to it, BCVI uses a data structure called a gap buffer. The bytes from the file are stored inside an array that's larger, and they are split into two contiguous pieces: the bytes from the first part of the file start at the beginning, and the bytes from the rest of the file go to the end of the array, but in between these two sections there is a "gap" of unused space. The location of the gap corresponds to where the insertion point (cursor) is in the file. When you move the cursor, bytes are copied across the gap. The main advantage of the gap is that it's efficient to insert or delete characters at the insertion point, since only one or the other half has to be modified. An equivalent way of looking at this is that the bytes are represented as two stacks, which grow in opposite directions, sort of like when going through a large pile of papers you can stack up the papers you've already looked at upside down. Inserting a character is like pushing it on one of the stacks, deleting one is like popping from one of the stacks, and moving the cursor corresponds to popping a character from one stack and the pushing it onto the other. To make it easy to access the text in the buffer by lines, BCVI also keeps a second gap buffer containing pointers that point to the newlines in the main character gap buffer; the gap in this buffer also moves in sync with the one in the main buffer when the cursor moves between lines. Some of the code that manipulates the gap buffer pointers may look a bit intimidating, since it has a lot of special cases and fencepost conditions. That complexity might cause it to contain bugs, but if you find it hard to understand, rest assured that BCVI also contains many vulnerabilities in other parts of the code as well.

To give you a feel for how security vulnerabilities evolve over time, and to provide a reason not to put all the work off until the last minute, we run the assignment in a weekly "penetrate-and-patch" format. Each week you'll be responsible for finding one vulnerability in BCVI, and producing an exploit for it; this exploit is due on a Friday. Then, by the following Monday, we'll post a new version of BCVI with one or more previous security vulnerabilities fixed ("patched"), and the cycle will repeat. (Note that in addition to the usual rules about partial credit for late submissions, you will get zero credit for an exploit if you submit it after we release a patch that fixes the same vulnerability. Just another reason

to submit on time.) Over time the more obvious or easy-to-exploit bugs in BCVI will get fixed, so you will have to find more subtle bugs and more sophisticated exploits.

For each hole you find, you should submit:

(a) A UNIX shell script (for the `/bin/bash` shell) that exploits this hole to open a root shell. In fact more specifically, just so there's no confusion about what's a root shell, we've created a new program named `/bin/rootshell` specifically for your exploit to invoke. If you invoke `rootshell` as root, it will give you a root shell as the name suggests; otherwise it will print a dismissive message.

Name your script `exploit.sh`. We will test your exploit scripts by running them as an ordinary user named `test`, starting from that user's home directory `/home/test`, with a fresh install of BCVI. So your scripts will need to create any supporting file or directory structures they need in order to work, and they need to run completely automatically with no user interaction. On the same CSE Labs machines with the VMs we will also provide you with a tool `test-exploit` you should use to test your exploit scripts.

(b) A text file that explains how the exploit works, named `readme.txt`. The text file `readme.txt` should identify what mistakes in the source code `bcvi.c` make the exploit possible, explain how you constructed your inputs, and explain step-by-step what happens when an ordinary user runs `exploit.sh`.

In choosing which vulnerabilities to patch each week, we will start by looking at which vulnerabilities were most commonly exploited, so there is a good chance that your old vulnerabilities will no longer work at all after the patch. However even if an old vulnerability happens to still work, you still need to submit an exploit for a new vulnerability each week. How can we judge whether two scripts, `exploit1` and `exploit2`, exploit different vulnerabilities? Imagine that you are a lazy programmer for Badly Coded, Inc., and someone shows you `exploit1`: a patch is in order! If there's a plausible patch the lazy programmer might write which would protect against `exploit1`, but still leave the program vulnerable to `exploit2`, then the two scripts count as exploiting different vulnerabilities. In particular, if you have an exploit that works against an old BCVI version, and the vulnerable code is changed to stop some attacks, but then you find an attack that works against the new "fixed" version, that also counts as a new exploit. If there could be any doubt about whether two of your exploits too similar in this way, for instance if they rely on the same or overlapping line(s) of code, you should argue for why they are distinct in your `readme` files. If you're not sure about whether two exploits are distinct, please ask us before turning the second one in. (Or of course you could also keep looking for more vulnerabilities: there are enough that are clearly distinct if you can find them.)

Because we won't be patching the vulnerabilities all at once, you have some flexibility in when you spend your time on this project: you might be able to save time later by finding a lot of different vulnerabilities early on. Since we'll be patching roughly in order of increasing difficulty, you'll want to use your simplest exploits first. Of course you always run a risk that

vulnerabilities will be patched if you save them: and even if the vulnerability still exists in a newer version, other changes to the program might mean that the exploit needs to be a bit different.

You'll probably want some of your exploits to be control-flow hijacking attacks as discussed in lecture. The classic tutorial on building such attacks is is $\aleph_1$'s "Smashing the stack for fun and profit," which can be downloaded from `http://www.insecure.org/stf/smashstack.txt`. Though it's detailed, it will still take some work to apply this tutorial to BCVI: for instance to find out the locations of things you'd like to overwrite, you'll need to do something like use GDB, add `printf` statements, or examine the assembly-language code.

In the course of the assignment there are a total of 100 regular points available of the first four weeks, and then extra credit points in the fifth and final week. Specifically the points are split up as follows:

- Week 1: 10 points, due Friday September 15th.

  In the first week of the assignment, you should familiarize yourself with reading the BCVI source code and writing automatic exploit scripts. But to make the searching a bit easier in the first week, we've included a vulnerability that's particularly easy to exploit: it's really just a regular feature of BCVI that would be easy to misuse to get a root shell. So once you find it, it shouldn't take much work to figure out how to attack it.

- Week 2: 20 points, due Friday September 22nd.

  In the first patch, the security experts at Badly Coded, Inc., will definitely fix the simple mentioned in the first week. But other than what was patched, you'll have your choice among the remaining vulnerabilities in the second week; probably some relatively simple exploits will still be possible.

- Week 3: 30 points, due Friday September 29th.

  By the third week the remaining vulnerabilities and attack techniques will become more subtle, which is part of why we increase the points available.

- Week 4: 40 points, due Friday October 6th.

  The fourth week exploit will again be challenging, so it's again worth 30 points.

  But even if we fixed all of the sloppy coding mistakes in BCVI, the *design* of the system leaves it vulnerable to some kinds of attacks. So taking the white-hat perspective, for the remaining 10 points, in a file called `design.txt` you should choose two or three secure design principles (for instance, among the ones discussed in lecture) which are most blatantly violated by BCVI. For these design principles, discuss how BCVI violates them and how you would change the design of BCVI to mitigate these vulnerabilities. If you feel it will be helpful, you can include pseudocode or working C to illustrate your changes.

- Week 5: $10 \cdot n$ points extra credit, due Friday October 13th.

  After the fourth week we will definitely have fixed all the easily exploitable bugs in BCVI, and we might even re-enable some of the security hardening mechanisms we'd earlier removed. But it's still not really secure. So if you're enjoying finding and exploiting bugs, you can keep going. You'll earn 10 points of extra credit for each additional unique exploit you find, limited only by the total number of remaining security bugs in BCVI. In addition to the extra credit, the team(s) that find the largest total number of bugs may also receive some special in-class recognition.

A portion of your grade for each exploit will depend on the quality of your explanation, to make sure you really understand what's going on. But an exploit that does not run `/bin/rootshell` as root when invoked by `test-exploit` is not an exploit as far as we're concerned. A non-working exploit will be eligible for at most 3 points of partial credit. Make sure to test your exploits carefully.

Happy Hacking!