CSci 5271: Introduction to Computer Security

Hands-on Assignment 1 **due: September 23rd and October 7th, 2016**

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 round, listing the names of all the group members. You'll be submitting answers in two rounds: by 11:55pm Central Time on Friday September 23rd and October 7th. When submitting, 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 round. 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 File Archiver," or BCZIP (loosely modeled on the Unix `tar` program and the cross-platform `zip`). (The same company produced the Badly Coded Versioning System and the Badly Coded Print Server used in previous years' 5271s.) You will be able to download the (round 1) source code for BCZIP from the course web page, at `http://cs5271.site/bczip1.tar.gz`, and (eventually) the round 2 source code from `http://cs5271.site/bczip2.tar.gz`. For its intended mode of use, BCZIP should be installed in a system-wide directory, and the vulnerabilities will be a problem when it is run as root. BCZIP can create (compress) archive files, when invoked as `bczip c`, and it can extract files from (decompress) archive files when invoked as `bczip x`. One would hope that an archiving system would be secure even when a privileged user is decompressing an untrusted archive, so to test this we've also installed another program, `bcunzip-as-root`, which is installed setuid root. This program will take an arbitrary BCZIP archive as an argument, and decompress it into a temporary directory while running as root. (This setuid decompression service might seem a bit artificial, but you can think of it as simulating a situation where a normal user decompresses files after downloading them from shady websites. Many of the techniques you use against `root` here could analogously be used to take over a user's account, steal credit card information, etc.)

Because BCZIP 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 BCZIP source, and even testing many kinds of attacks, without a VM: BCZIP 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.

BCZIP has a limited set of features compared to a practical archive program. It only supports a few kinds of compression, and they don't compress very well. Also, it doesn't know

how to store or create directories, so it works best for just archiving files from or extracting back into the current directory. A BCZIP archive consists of a sequence of "members", each of which is either a normal file or a symbolic link. Normal files are stored by breaking them up into 4096-byte chunks, and compressing each chunk with the most effective supported method; if no compression mechanism is effective for a chunk, BCZIP will just store it uncompressed ("raw"). BCZIP files have a simple binary format, in which types of members and blocks are introduced by particular bytes, and strings and block data are stored preceded by their length in bytes represented as a 32-bit big-endian integer. To create an archive, you run `bczip` with a first argument of `c` (for "create"), a second argument that is the name of the archive to create (conventionally we use the file extension `.bcz`), and any remaining arguments are members to go in the archive. To reverse the process and decompress all the files from an archive, run with a first argument of `x` ("extract") and a second argument of the archive file.

BCZIP also has a few other features that can be used to affect the decompress process with environment variables (which are automatically passed through `bcunzip-as-root`). One danger of extracting from an archive is that you might accidentally overwrite an important file with one from the archive with the same name. But if you set the environment variable `BACKUP_FORMAT` to a string containing `%s`, then BCZIP will rename files it would otherwise overwrite to a new name formed from the format by putting the old name in for the `%s`. For instance you might use `BACKUP_FORMAT=%s.bak` to make backup copies ending in `.bak`. Though BCZIP archives do not store ownership and permission information for their contents, these can be controlled with the `BCZIP_OWNER` and `BCZIP_MODE` environment variables: the latter should be an octal value prefixed with 0.

BCZIP 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 controls the archive file to become root. The main part of the assignment is for you to find four ways to get a running command shell with UID 0 as a result of sloppy coding and/or design in BCZIP. (We haven't intended to put any vulnerabilities in `bcunzip-as-root`, but if you find any you can exploit them too.) 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.

Another thing to consider in planning your exploits is where the BCZIP archive files will come from. For some vulnerabilities, you may be able to exploit them using a file produced by the normal `bczip c`. But for other attacks, you may want to give an abnormal file with `bczip c` would not have created, but causes `bczip x` to do something it's not supposed to.

Patching 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 "penetrate-and-patch" format. There will be two rounds of this format. In round one you'll be responsible for finding two vulnerabilities in BCZIP, and producing exploits for

them; these exploits will be due Friday, September 23rd. Then, by the following Monday, we'll post a new version of BCZIP with one or more of these 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.) The more obvious or easy-to-exploit bugs in BCZIP are likely to be fixed after round one, 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 round 1 scripts `exploit1.sh` and `exploit2.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 BCZIP. 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.

(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 `bczip.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`. Also, this file shuold contain the names of your group members.

In choosing which vulnerabilities to patch each round, 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 in the next round**. 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. If there could be any doubt about whether two of your exploits are 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 BCZIP: 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.

Submission Process There will be two rounds of submissions. For each round, one person from your group will submit for the whole team. In the course of the assignment there are a total of 100 regular points. Specifically the points are split up as follows:

- Round 1 due Friday September 23rd
  Deliverables:
  - exploit1.sh (8 points)
  - readme1.txt (8 points)
  - exploit2.sh (8 points)
  - readme2.txt (8 points)

  The above files should be inside a tar file named after your group (e.g. group1.tar) and submitted to Moodle.

- Round 2 due Friday October 7th
  Deliverables:
  - exploit3.sh (14 points)
  - readme3.txt (14 points)
  - exploit4.sh (14 points)
  - readme4.txt (14 points)
  - design.txt (12 points)

Notice, for the second round, you should also submit a document titled *design.txt*. In this file you should choose three secure design principles (for instance, among the ones discussed in lecture) which are most blatantly violated bv BCZIP. For these design principles, discuss how BCZIP violates them and how you would change the design of BCZIP to mitigate these vulnerabilities. Include pseudocode or working C to illustrate your changes.
The above files should be inside a tar file named after your group (e.g. group1.tar) and

submitted to Moodle.

Grading 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 your script 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.

Nishad's Pro Tips

- As mentioned, certain vulnerabilities will be patched after Round 1. Therefore, it's recommended to find the easiest vulnerabilities first.
- The codebase may seem daunting at first, so break it up. Make a quick scan – are there any dangerous functions or operations being executed? If so, trace its calls backwards until you can find a point of entry where you can exploit it to your advantage.
- Look over the various techniques you've learned in lecture. If you understand them, look for parts in the code where such a technique could be used (e.g. if you know how to perform a buffer overflow, it will be easier to find where that might work)
- Are there any files or programs on the OS that could be valuable if you could gain access to them? How can you manipulate BCZIP into accessing those resources?
- Since you have the source code for BCZIP and are able to install it on your VM, you can make a copy for yourself that includes debug print statements or any other edits to the code that may be helpful. If you do make any changes, verify that your exploit still works with a fresh install of BCZIP before submitting your exploit.

Happy Hacking!