EECS 398 Project 3: Seizing Control with Buffer Overflows

Revision History

- 1.0 (November 3, 2009) – initial release.

1 Introduction

In this project, you will exploit several classic software vulnerabilities. We will provide a series of vulnerable programs and a virtual machine environment in which you will develop your exploits.

2 Ethics

This is an attack project. As a reminder, *you must not attack any computers over which you do not have sole control*! If you have any questions about this policy, especially as it applies to this project, you must contact the course staff before proceeding. If you do not abide by this policy, *you will fail the course.* See the "Ethics, Law, and University Policies" section on the course website at `http://www.eecs.umich.edu/courses/eecs398/` for further information.

3 Groups

You must work alone for this project. You may not communicate with anyone other than the course staff in any way regarding the solutions to this project. You may confer with other students regarding setting up VirtualBox and the details of the *specification.*

4 Goals

1. Be able to identify buffer overflow vulnerabilities in native code.

2. Be able to demonstrate the severity of buffer overflows and the necessity of the standard defenses.

3. Become familiar with basic machine architecture and assembly language.

5 Targets

The target programs for this project are simple, short C programs with (mostly) clear security vulnerabilities. For each target, we have provided both source code and the compiled program. We have also provided a Makefile that compiles all the targets. The source code and Makefile are for your convenience during testing; your attacks must work against the exact binaries we have provided. The targets are divided into three groups, explained below.

5.1 Subverting control flow

The first group of targets consists of several variations on the example from discussion on Friday, October 30. These programs all take input on the standard input. They are `example1`, `example1_nostatic`, and `target1`.

For each target, your goal is merely to subvert the control flow of the program. In particular, you will cause the program to print “Your grade is perfect.”, using a function that we have included in each target solely for that purpose. It is OK if the program crashes after this happens.

5.2 Redirecting control to shellcode

The second group of targets uses two classically vulnerable C standard library functions. They are `target2` and `target3`.

In these attacks, your goal will be to redirect control to code that you will inject. In particular, you will inject our provided shellcode in order to start a root shell (i.e., a command prompt with administrator permissions), assuming that the target is installed setuid root in /tmp. (A setuid program is one that runs with the permissions of its owner, even if it is run as another user; thus, a setuid root program is one that always runs with administrative privileges.) We have included install and uninstall targets in the Makefile to perform this installation for you, and we have also provided a short x86 shellcode (see Section 7) to start the root shell.

To facilitate opening the root shell, these programs take input as a single command-line argument.

5.3 Trickier vulnerabilities

The final group of targets does not allow you to directly overwrite the return address in the process of overrunning a buffer. Instead, you must use a less direct method. (If you find an attack that does directly overwrite a return address as part of overrunning a buffer, please notify us so that we can correct the assignment (or give out extra credit if it is a clever attack and not a simple mistake in the assignment)).

Like the second group of targets, the intermediate targets take input as a single command-line argument. Also like the second group, your goal is to use the provided shellcode to open a root shell, assuming that the targets are installed setuid root in /tmp.

The intermediate targets are target4 and target5.

6 Running the virtual machine and testing your attacks

Because you need administrative privileges on your test machine, the CAEN machines are not usable for this project. We ask that you install VirtualBox (`http://www.virtualbox.org/`) on your personal machine and use the virtual machine appliance we have provided at `http://www.eecs.umich.edu/courses/eecs398/398-32bit-release.ovf` and `http://www.eecs.umich.edu/courses/eecs398/398-32bit-release.vmdk` (the download is 443 MB, so be sure to download it from on campus if possible). VirtualBox runs on Windows, Linux, Macintosh, and OpenSolaris hosts. Further guidance on setting up and using VirtualBox for this project will be released soon.

If you are unfamiliar with virtual machine software, it allows you to run a separate operating system (called the guest) as though it were a program running on your normal operating system (called the host). Thus, you will have a simulated computer on which to run your attacks. Modern advances have made virtual machines perform quite well.

If you absolutely do not have a personal computer available on which to run VirtualBox, contact the course staff ASAP for an alternative.

Inconveniently, we have “forgotten” the passwords for the virtual machine. Thus, your first task is to set up the virtual machine, gain root privileges, and reset the passwords on the `root` and `student` accounts, demonstrating how easy it is for an attacker with “physical” access to a machine to gain control over it. (Hint: try Google.)

7 Deliverables

For each target, you will provide a very short Python script that prints appropriate input to exploit the target. For each target foo, your Python script will be named attack_foo.py. We have provided a simple example in attack_template.py.

For each of the first group of targets, test the target `foo` with the command line:
`python path/to/attack_foo.py | ./foo`.

For each target where you are to open a root shell, test the target `foo` with the command line:
`/tmp/foo $(python path/to/attack_foo.py)`. (This syntax uses the output of your Python script as the first command-line argument.) Remember to run `make install` first to install the targets in /tmp.

We have provided `shellcode.py`, which defines a single variable called SHELLCODE whose value is some compiled x86 shellcode (for reference, it is the execve portion of `http://milw0rm.com/shellcode/2042`). You should import this file in your attack scripts. Successfully placing the shellcode in memory and setting the instruction pointer to the beginning of the shellcode (e.g., by returning or jumping to it) will open a shell.

8 Hints and Guidelines

You will almost certainly find it easier to do this project if you use the GDB debugger, which you should recall from EECS 280. Useful commands that you may not know are "disassemble", "info registers", "x", and "stepi". See the built-in GDB help for details, and don't be afraid to experiment!

If your exploit is crashing the target process, you can review the state of the process at the time of the crash by enabling *core dumps* with the command `ulimit -c unlimited`. This will cause a file called `core` to be created after each crash. To debug, provide the name of the core file as an additional command-line argument to GDB: `gdb ./example1 core`. Under certain conditions, an error message is also placed in the kernel log; you can view such messages with the `dmesg` command. As the output of `dmesg` is quite large, we recommend that you pipe it to the `tail` command: `dmesg | tail`.

Review the posted notes from discussion on Friday, October 30.

Read Smashing the Stack for Fun and Profit (`http://insecure.org/stf/smashstack.html`).

As a reminder, all the standard defenses against buffer overflow attacks are off. This means no ASLR (address space layout randomization), no stack canaries, no non-executable stack, disabled overflow checking in glibc, and so forth.

9 Turning it in

Submit your project via email to `swolchok@eecs.umich.edu` by 5:00 PM on November 19, 2009. Your submission should be a gzipped tar archive named proj3.uniqname1.tar.gz with all the files in one directory called proj3.uniqname1.

Late submissions are guaranteed not to receive any extra credit and may also incur substantial penalties. Start early!