CS155: Computer Security **Spring 2007**

Project #1

Due: Part 1: Thursday, April 12 - 1159 pm, Part 2: Thursday, April 19 - 1159 pm.

Goal

1. The goal of this assignment is to gain hands-on experience with the effect of buffer overflow, format string, and double free bugs. All work in this project must be done on a system called boxes (implemented using User-Mode Linux) available on the course web site.

2. You are given the source code for seven exploitable programs (`/tmp/target1`, ... , `/tmp/target7`). These programs are all installed as setuid root in the boxes system. Your goal is to write seven exploit programs (`exploit1`, ..., `exploit7`). Program `exploit[i]` will execute program `/tmp/target[i]` giving it certain input that should result in a root shell on the boxes system.

3. The skeletons for `exploit1`, ..., `exploit7` are provided in the `exploit/` directory. Note that the exploit programs are very short, so there is no need to write a lot of code here.

The Environment

1. You will test your exploit programs within a system called Boxes. Boxes, based on User-Mode Linux, allows you to boot a fully-functional Linux system as a userland process on another Linux machine.

2. Boxes is available from the course website. It should run on most x86 GNU/Linux machines. However, people have reported problems running it on some 2.6-series kernels. Because of this we recommend you do all your work on the CS department's myth cluster—in our testing Boxes worked fine on these machines.

   You must install Boxes on a GNU/Linux machine on your own. Refer to the end of the handout and the README file in the Boxes distribution.

3. It is recommended that you test your exploits in a virtual machine booted with a "closedbox" kernel, so that you cannot accidentally damage your host account.

4. You can use the ssh daemons running in the image to transfer files from openboxes (with hostfs access) to closedboxes.

5. It is recommended that you develop your code on the host machine, or at least keep frequent backups. The User-Mode Linux kernel is mostly stable, but can occasionally crash.

6. The last section of this handout runs you through setting up the environment on the myth cluster in Gates B08 (also available through ssh).

The Targets

1. The targets/ directory in the assignment tarball contains the source code for the targets, along with a Makefile specifying how they are to be built.

2. Your exploits should assume that the compiled target programs are installed setuid-root in /tmp – /tmp/target1, /tmp/target2, etc. Don't move them because this is where they will be when we grade your code.

The Exploits

The sploits/ directory in the assignment tarball contains skeleton source for the exploits which you are to write, along with a Makefile for building them. Also included is shellcode.h, which gives Aleph One's shellcode.

The Assignment

You are to write exploits, one per target. Each exploit, when run in the Boxes environment with its target installed setuid-root in /tmp, should yield a root shell (/bin/sh).

Hints

1. Read Aleph One's "Smashing the Stack for Fun and Profit." Carefully. Also read the two optional handouts — have a good understanding of what happens to the stack, program counter, and relevant registers before and after a function call. Read scut's "Exploiting Format String Vulnerabilities.". All the papers are linked from the course syllabus. It will be helpful to have a solid understanding of the basic buffer overflow exploits before reading the more advanced exploits.

2. gdb is your best friend in this assignment, particularly to understand what's going on. Specifically, note the "disassemble" and "stepi" commands. You may find the '`x`' command useful to examine memory (and the different ways you can print the contents such as `/a /i` after `x`). The '`info register`' command is helpful in printing out the contents of registers such as `ebp` and `esp`.

   A useful command to run gdb is to use the `-e` and `-s` command line flags; for example, the command '`gdb -e sploit3 -s /tmp/target3`' in boxes tells gdb to execute sploit3 and use the symbol file in target3. These flags let you trace the execution of the target3 after the sploit has forked off the execve process. When running gdb using these command line flags, be sure to 'run' the program before you set any breakpoints; for our purposes, entering the command 'run' naturally breaks the execution at the first `SIGTRACE` before the target is actually exec-ed, so you can set your breakpoints when gdb catches the `SIGTRACE`. Note that if you try to set break points before entering the command 'run', you'll get a segmentation fault.

   If you wish, you can instrument your code with arbitrary assembly using the __asm__() pseudofunction.

3. Make sure that your exploits work within the Boxes environment (and specifically the closed-box).

4. Start early. Theoretical knowledge of exploits does not readily translate into the ability to write working exploits. Target1 is relatively simple and the other problems are quite a bit more complicated.

Warnings

Aleph One gives code that calculates addresses on the target's stack based on addresses on the exploit's stack. Addresses on the exploit's stack can change based on how the exploit is executed (working directory, arguments, environment, etc.); in our testing, we do not guarantee to execute your exploits as bash does.

You must therefore hard-code target stack locations in your exploits. You should *not* use a function such as get_sp() in the exploits you hand in.

Deliverables

1. To encourage students to start on the project early, part 1 (due on April 12 1159 pm) consists of target1 and target2. Part 2 consists of the other 5 targets.

2. You are to provide a tarball (i.e., a .tar.gz or .tar.bz2 file) containing the source files and Makefile for building your exploits. All the exploits should build if the "make" command is issued.

3. There should be no directory structure: all files in the tarball should be in its root directory. (Run tar from inside the sploits/ directory.)

4. Along with your exploits, you must include file called ID which contains, on a single line, the following: your SUID number; your Leland username; and your name, in the format last name, comma, first name. An example:

   ```
   $ cat ./ID
   3133757 hermann Buhl, Hermann
   $
   ```

   If you did the project with a partner, then both of you will submit only one solution and the ID file will have two lines giving the relevant information.

   You may want to include a README file with comments about your experiences or suggestions for improving the assignment.

5. Instructions for submitting the tarball will be posted on the course website. Again, make sure that you test your exploits within the Boxes environment.

Late Policy

Every student in the class is given a total of 72 late hours that can be applied to the projects and homeworks. These late hours must be taken in chunks of 24 hours (essentially 3 late days) — for example, submitting a homework 3 hours later than it's due counts as 24 late hours used. After all your late hours are used up, the assignment score gets halved with every 24 hours the assignment is late — for example, someone submitting a project 47 hours late after having used all her late days will get $(1/2)^2 = 1/4$ of the grade. If a project has more than one part, each part is considered a separate assignment for late days — for example, if you submit part 1 72 hours late and part 2 24 hours late, then part 1 gets full credit and part 2 gets 50credit.

Note that no late hours can be used on the last programming project.

How to set up the Environment

Below are instructions for setting up the environment on the myth(1-29) machines. You can probably use any machine running Linux but, again, many people have reported problems running Boxes on some machines, and myth works fine in our testing. Note: You have been assigned an extra 500 mb of quota on your Leland account.

1. Log into the myth machine of your choice, and get Boxes and the programming project.

   ```
   wget http://crypto.stanford.edu/cs155/cs155-pp1.tar.gz
   wget http://crypto.stanford.edu/cs155/boxes-040405.tar.bz2
   ```

2. `cd` into `/tmp`, create a directory for yourself, and extract the files.

   ```
   myth10:/tmp/eujin>tar jxvf boxes-040405.tar.bz2 ; tar xxvf cs155-pp1.tar.gz
   ```

   Next, make sure that no one else can read your directory in `/tmp` by setting the permissions using '`chmod 700 /tmp/eujin`'.

3. Since we extracted the files into `/tmp/eujin`, we need to set up the Boxes enviroment variables as follows

   ```
   myth10:/tmp/eujin> setenv BOXESDIR /tmp/eujin/boxes
   myth10:/tmp/eujin> setenv BOXESHOME /tmp/eujin
   myth10:/tmp/eujin> setenv PATH /tmp/eujin/boxes:$PATH
   ```

   This assumes you're using the default tcsh shell. For convenience, instead of repeatedly typing in the commands, you may wish to place the three setenv commands into a file and '`source`' the file whenever you log into a machine.

4. Now we want to start openboxes with a virtual network. Don't forget that we're testing on a closedbox so you want to test your exploits on that before submitting. The difference between an open and closed box is that you can't mount the hostfs in a closed box and you can't ssh into a closed box. Start the switch daemon (behaves like a network switch).

```
myth10:/tmp/eujin> xterm -e boxes/string &
```

Then run boxes, specifying the copy-on-write disk and the virtual private network.

```
myth10:/tmp/eujin> boxes/openbox cow1 10.64.64.64
```

5. There are two accounts on these boxes, root and user. The passwords are the same as the usernames. To get your project files onto boxes, log in as root and mount myth10's local filesystem in boxes with the following command

   ```
   box:~# mount none -t hostfs /mnt -o /tmp/eujin/pp1
   ```

   Now you can access the project files in `/mnt`. Copy the sploits dir to the user's home directory (and make sure to set the ownership so that user can access them '`chown -R user:user sploits`'), and target dir to root's home directory. Make the targets and copy the targets to `/tmp` together with the corresponding .c files. Using the following commands, set up the permissions so that the targets are owned by root, are setuid root, and the .c files are publicly readable.

   ```
   box:~# chown root:root target? ; chmod 4755 target? ; chmod a+r target?.c
   ```

6. Everytime you reboot boxes, you'll have to set up the targets in boxes' `/tmp` because it'll be wiped clean. Also make sure to **periodically transfer your work out to your home directory in AFS space** because the myth machines' `/tmp` (containing your copy of boxes and the project) gets wiped out every few days. To save your instance of boxes, you can copy `cow1` to your AFS space and then when you set boxes up again, you can just copy cow1 back to `/tmp` and start boxes with that image. You may also want to keep everything in your home directory and only create symbolic links in `/tmp`.

7. To create more virtual terminals, while root in boxes, do

   ```
   box:~# TERM=vt100 ; vi /etc/inittab
   ```

   and uncomment out lines:

   ```
   ##2:23:respawn:/sbin/getty 38400 vc/2
   ##3:23:respawn:/sbin/getty 38400 vc/3
   ```

   to spawn two extra console windows on the next reboot of boxes. The only two editors on the boxes environment are `nano` and `vi`. If you prefer other editors, write the code outside of boxes and then use 'nano' to paste the code into a text file in a boxes console.

8. For more information, read `boxes/FAQ` and `boxes/README`.

9. The easiest way to transfer your work from an openbox to a closedbox is to reuse the same cow when starting up a closedbox.