CS 642 Homework #4

Due Date: 11:59 p.m. on Tuesday, May 1, 2007

Warning!

In this assignment, you will construct and launch attacks against a vulnerable computer on the CS network. The network administrators are aware of this homework exercise and expect attacks against the machine. However, you may not use the techniques in this homework to attack any other machine in the CS network or greater Internet, and you may not use root-level access on the target machine to launch further attacks. Use your head: there are University and criminal repercussions for such activity.

This homework assumes a not-insignificant level of experience (or at least comfort) with a UNIX/Linux shell. Furthemore, the assignment emphasizes research and experimentation.

Start work on this assignment early!

Hello wily hacker.

This assignment guides you through the steps an attacker may use when launching an attack across a network. Your end goal: retrieve the password file `/etc/shadow` from a specific machine located somewhere in the CS building. The password file is readable only by root, so you must first gain root access to the machine. This is a multi-stage attack that requires significant effort on your part. To reach the password file, you will need to remotely scan the machine, run a remote buffer overrun exploit, run a local format string exploit, and break out of a chroot jail.

You may have difficulty successfully completing some stages of the attack. We will provide alternate instructions that will allow you bypass a stage that you cannot complete so that you can continue with the assignment. The alternate instructions also allow you to solve the stages in various orders, although if you solve all stages you will have a complete attack. The alternate routes are worth 0 (zero) points. If you follow only the alternate instructions and do not solve a stage, you will not receive points for that stage. You will still receive points for any questions that you answer in your writeup.

This is a lengthy homework problem that requires you to use tools with which you may be unfamiliar. Google for things that you do not understand. You are allowed to use online

resources without point deduction, although your final homework writeup must cite the URLs that you used. All the programs that you will run are either already installed on CS lab machines or are available from `~mihai/public/642`.

The target machine is at IP address 128.105.48.223.The machine is running Ubuntu Linux with a 2.6.17 kernel.

Network access to `128.105.48.223` is blocked from the outside Internet. You can only reach the victim machine from inside the CS networks. If you work from home or any non-CS computer lab, you must first ssh to a computer in a CS lab and then run attacks against `128.105.48.223` from the lab machine.

The attack will work as follows:

1. Remotely portscan the machine to find a vulnerable network service.
2. Construct a buffer overflow exploit and run it against a vulnerable HTTP server to gain user-level access to the remote machine. The server runs in a chroot jail, so you will have very limited filesystem access. This stage has bypass instructions if you cannot build a successful overflow exploit.
3. Gain root access by constructing a format string attack against a vulnerable setuid-root program running in the chroot jail. This stage has bypass instructions if you cannot build a successful format string attack.
4. As root, break out of the chroot jail to gain full filesystem access. Copy the password file `/etc/shadow` to your CSL lab account.

You will hand in a tarball or ZIP file containing:

1. Your buffer overflow exploit.
2. Your format string exploit.
3. The source code for your jailbreak program.
4. The `/etc/shadow` file retrieved from `128.105.48.223`.
5. A typed write-up with your solutions to all questions Q1 through Q12.

Email your tarball or ZIP file to mihai@cs.wisc.edu before 11:59 p.m. on the due date.

Please direct questions about this assignment to Mihai Christodorescu at mihai@cs.wisc.edu. In particular, contact me for the following difficulties:

- `128.105.48.223` seems to have crashed.
- You follow the directions listed here, but things do not work as described.
- You think your exploit is valid, but it still does not work.

Let's get started.

Stage 1: Exploration

Portscan `128.105.48.223` to learn what services are running. Use the scan program nmap. Feel free to try the various scan types provided by nmap, but limit your scans to only `128.105.48.223`.

Code Example 1: nmap scan

```
$ nmap 128.105.48.223

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Interesting ports on 128.105.48.223:

[nmap results follow]
```

Q1: What services are running on `128.105.48.223`?

Q2: What do open, filtered, and closed ports mean?

Q3: How does nmap identify whether a port is open, filtered, or closed?

There is a simple http server running on `128.105.48.223`. This server contains a buffer overflow vulnerability that will be our point of entry to the system in Stage 2. You can actually verify that the server correctly delivers files by opening http://128.105.48.223/index.html in your favorite web browser.

Stage 2: Gain Remote User Access

Your portscan should have shown that `128.105.48.223` is running an HTTP server. This server contains a buffer overflow vulnerability in its input processing that you will exploit to gain access to the machine. In this stage, you will build up a buffer that will trigger the vulnerability and will eventually send the buffer to `128.105.48.223`. The attack will convert the web server to a shell process, allowing you to execute arbitrary commands on the victim machine.

You may wish to review buffer overflow vulnerabilities before continuing.

The source code for the web server running on `128.105.48.223` is in the file `httpd.c` located at `~mihai/public/642/httpd.c`. Review the source code to answer the next two questions.

Q4: Explain the buffer overflow bug in `httpd.c`. How serious is this bug?

Q5: Propose a source code patch that will remove the vulnerability.

Now, you will construct an exploit for this vulnerability. The compiled `httpd` running on `128.105.48.223` is located at `~mihai/public/642/httpd`. Grab a copy so that you can work on the exploit locally. You will attack `128.105.48.223` only when your exploit first works against a local `httpd`.

This `httpd` is very simple: it reads an HTTP request from standard input and writes its output to standard output. It takes a single command-line parameter, "`-r`," that specifies the WWW root directory. Try it:

Code Example 2: Sample execution of the httpd program

```
$ echo "GET .cshrc" | ./httpd -r ~/
HTTP/1.0 200 OK
Content-type: text/html
Content-length: 3329
Connection: close

#
# This file (.cshrc) is read by each new ...
#
# There are comments to explain most of what ...

[output continues]
```

After you construct a buffer that you believe exploits the overflow vulnerability, you can test the buffer by simply feeding it to `httpd` as input via a UNIX pipe or redirected input.

You can build a buffer by writing a text file that specifies each byte of your buffer as a two-character hexadecimal duplet. For example, the byte sequence 0x3f5c88 would be listed in the text file as "`3f 5c 88`". The program `~mihai/public/642/make_bin` will read the text file as input and output a file with the binary representation of the hex duplets. The file `overflow.bin` would then be the input to `httpd`.

Code Example 3: Use of the `make_bin` hex generator

```
$ make_bin < overflow.hex > overflow.bin
$ ./httpd < overflow.bin
```

Alternatively, you can set up a pipe chain:

Code Example 4: Use of `make_bin` in a pipe chain

```
$ cat overflow.hex | make_bin | ./httpd
```

We have provided an example file `~mihai/public/642/segfault.hex` that will cause `httpd` to segfault.

Your task is to create an `overflow.hex` file that exploits the buffer overflow vulnerability to cause httpd to execute a shell process. Your overflow will inject shellcode into the program. Shellcode contains instructions that may not have appeared in the original program. We have written the shellcode for you. The file `~mihai/public/642/shellcode.s` contains the x86 assembly-language instructions (in AT&T syntax) making up the shellcode, and `~mihai/public/642/shellcode.hex` gives the hexadecimal representation of the shellcode.

Q6: Analyze the assembly code in `~mihai/public/642/shellcode.s`. What is this code doing?

Although you may take any approach you like to build your exploit buffer, we suggest the following:

1. Figure out how to manipulate the return address on the stack. You may want to start with the example `segfault.hex`. Find the place in the buffer that overwrites the return address either by computing the size of the stack frame or by simply manipulating values in the `segfault.hex` buffer to see where in memory the values are written. Figure out how to specify an address of your own choosing.

2. Figure out how to include shellcode in the buffer. You can use the hexadecimal-encoded shellcode at `~mihai/public/642/shellcode.hex`. You just need to figure out where to put the shellcode in your buffer.

3. Identify the address where the shellcode will appear in memory after the buffer overrun. Make your buffer overwrite the return address with this value. As a result, when the function returns, the program's execution will jump into the shellcode.

Special note. Even a correct exploit will segfault when tested on the CSL lab machines. A correct exploit overwrites the return address to specify the value in the stack where the shellcode resides. The CSL machines have a security restriction in place that will not allow instructions on the stack to execute. When the program reads the overwritten return address and attempts to execute code in the stack, it will segfault. The `128.105.48.223` machine does not have this restriction.

To test your exploit on a CSL lab machine, you must use a debugger such as `gdb`. At a return instruction, the return address about to be used will be at the top of the stack. To see if your exploit will work, run `httpd` inside the debugger. Insert a breakpoint at the return instruction that uses the clobbered return address. Run the program, with the exploit buffer as input, up to the breakpoint. Check that the return address targets the shellcode in your buffer.

. . . . . .

Once you have an exploit that you believe may be successful, run it against the remotely executing `httpd` on `128.105.48.223`. The weird syntax of the command that sends the exploit to `128.105.48.223` will keep the connection open after sending your exploit code:

Code Example 5: Buffer-overflow attack

```
$ make_bin < overflow.hex > overflow.bin
$ (cat overflow.bin ; cat) | nc 128.105.48.223 80
```

If your exploit is successful, you will receive a bash-shell prompt on the remote machine. To logout of `128.105.48.223` after a successful attack, type "`exit`" to close the bash prompt and then type "`^d`" (control-d) to close the locally-executing `cat` process. ***Include the `overflow.hex` file in your tarball or ZIP file that you hand in.***

Alternative instructions for Stage 2: If you are unable to construct a buffer that successfully triggers the overflow vulnerability to give you a shell on `128.105.48.223`, you can connect to `128.105.48.223` via ssh to continue the homework assignment. Your level of access will be equivalent to the access gained via a successful exploit.

Ssh to `128.105.48.223` with username `student` and password `cs642student`.
This is worth 0 (zero) points.

. . . . . .

After connecting to `128.105.48.223` via a successful overflow or via the alternate instructions, type "`whoami`" and note that you do not have root access. Look around the filesystem. It looks a bit strange. There are very few commands in `/bin` or `/usr/bin`. There isn't even a kernel. There is an `/etc/shadow`, but this happens to be the wrong one. The reason the filesystem looks so empty is that you exploited a process executing inside a chroot jail, discussed further in Stage 4. To see the full filesystem, you need to break out of the jail. Breaking out requires root privilege, which you will gain next in Stage 3.

Some hints for Stage 2:

1. The stack stabilizing macro in `httpd.c` is not relevant to the overflow vulnerability and is simply present to make exploit construction easier. You can completely ignore this macro.

2. If you cannot find the overflow in `httpd.c`, try compiling the program. `gcc` may help you out.

3. Test your exploit against the precompiled `httpd` that we provide. If you test against an `httpd` that you compile yourself, offsets in the program may change. An successful exploit against a recompiled `httpd` may not be successful against our `httpd` running on `128.105.48.223`.

4. The x86 processor is little endian. This affects how you specify values that overwrite the return address.

Stage 3: Gain Remote Root Access

To escalate privilege from user to root, you will exploit a vulnerability in a setuid-root program. Regardless of which user installed the program, most programs run with only the privilege of the user executing the program. Setuid programs are different: they run with the privilege of the owner of the program. If the owner is root, then they execute with root privilege. One such program is su: the UNIX switch user command. This program must run as root because it must be able to switch to any user on the system, including root. After verifying a password and switching to a specified user, su then executes a shell process with the privilege of that user. The machine `128.105.48.223` has a vulnerable su program accessible in the chroot jail.

This program is vulnerable to a format string attack. Programs written for UNIX or Linux often use output functions that make use of a format string; printf is a good example. If an attacker can specify the format string passed to a function such as printf, then they can provide a string that causes printf to write arbitrary values to arbitrary memory locations.

The source code for su is at `~mihai/public/642/su.c` and the binary program is at `~mihai/public/642/su`.

Q7: Identify and explain the format string bug in `su.c`.

Q8: Propose a source code patch that will remove the vulnerability.

You will now create a format string attack against su. This attack is different from the buffer overflow exploit: you will not insert shellcode. Instead, you will use the format string to overwrite just a return address. When the return address is used, the program's execution will bypass the code of `su.c` that performs safety checks such as password verification and will execute a shell with root privilege.

Format string attacks are exceptionally complicated, so we have written almost the entire format string for you. Your task is to specify what address should overwrite the existing return address. You must analyze the code of su to find an existing instruction to which the return instruction should jump.

After you identify such an address, generate the format string using the command `make_string` located at `~mihai/public/642/make_string`. This example will create a format string that will overwrite the return address with the address of the main function:

Code Example 6: Creating a format string

```
$ make_string 0x8048970 > format.bin
```

Test the format string against a local copy of `~mihai/public/642/su`:

Code Example 7: Performing the format-string attack

```
$ ./su `cat format.bin`
```

A successful exploit will start a new shell process. Your local copy of su is not setuid, so the shell will not be root-level.

Q9: Look at the format string produced by `make_string`. How does this attack work? What is the meaning of the bizarre format string? Why does this string cause a return address to be overwritten?

Special note. Even a correct exploit will segfault when tested on the CSL lab machines. The `128.105.48.223` machine does not have this restriction. It is sufficient for you to identify the right instruction to which the return instruction should jump. You can test your exploit using a debugger such as `gdb`. Run `su` inside the debugger and set a breakpoint at the instruction you identified. If the exploit is correct, the execution will stop at that instruction.

. . . . . .

Once you have a format string that executes a shell locally, attack the remote su process. Connect back to `128.105.48.223` via the buffer overflow or via the alternate instructions. Move into the directory `/home/student` and make a new directory for yourself. Upload your format string to your new directory. You can use `scp` and the `student` account to upload your format string file, without losing any points. Run the attack against su. A successful attack will give you a bash prompt with root privileges.

Code Example 8: Format-string attack

```
$ /bin/su `cat /home/student/myname/format.bin`

[some weird output here]

bash-3.00# whoami
root
```

Clean up after yourself by removing the directory and any files that you added. ***If your attack is successful, include your format string in the tarball or ZIP file that you hand in.***

Alternative instructions for Stage 3: If you are unable to construct a format string that successfully triggers the vulnerability in `su`, you can gain root access by executing `/bin/make-me-root`.

This is worth 0 (zero) points.

You are still inside the chroot jail and have limited filesystem access, but you now have the privileges that you need to break out of the jail.

Some hints for Stage 3:

- The stack stabilizing macro in `su.c` is not relevant to the format string attack and is present only to make exploit construction easier. You can completely ignore this macro.

Stage 4: Chroot Jailbreak

All the processes that you have exploited on `128.105.48.223` are executing inside a chroot jail. A chroot jail is a security feature that limits the filesystem access given to particular processes. If the process is exploited, as you have done in Stages 2 and 3, an attacker has limited access to the underlying system.

The Linux kernel has a data structure for each process executing on the system. This data structure includes a field that specifies the process's root directory. Most processes will have a root that matches the actual filesystem root. A process inside a chroot jail, however, has a different root directory. Only files reachable from the process's root directory can be accessed by the process. The rest of the filesystem is unreachable. If an attacker wants full access to the filesystem, they must break out of the chroot jail by resetting the root directory to the actual filesystem root.

Your task in Stage 4 is to write a program that will break you out of the chroot jail. If you use a textbook or online resource to help you break out of the jail, make sure you cite the source in your writeup.

After writing and compiling your program, connect to `128.105.48.223` and get root access. Create a subdirectory for yourself in `/home/student` and upload your jailbreak program to your new directory. You can use `scp` and the `student` account to upload your format string file, without losing any points. Run your program to break out of the chroot jail. If your

break was successful, you will be able to access the entire filesystem of `128.105.48.223`. ***Include the source code for your jailbreak program in your tarball or ZIP file that you hand in.***

Q10: Explain how to break out of a chroot jail.

Q11: The chroot jail on `128.105.48.223` was insecure and allowed you to break out. How could this jail have been made secure?

Q12: You attacked the programs `/bin/httpd` and `/bin/su` inside the chroot jail. What is the complete path to these programs from the actual filesystem root directory?

Stage 5: Root-Level Activity

You can now run arbitrary commands on `128.105.48.223` with root privilege. Please do not do anything destructive. I will be severely unhappy if I need to reinstall the operating system midway through this assignment.

In particular, you now have full access to the filesystem. Copy the real `/etc/shadow` to your CSL account. ***Include `/etc/shadow` in your tarball or ZIP file that you hand in.***