Secure Programming Lab 1: Data Corruption

School of Informatics, University of Edinburgh

3pm-6pm, 7th February 2017

Exercise 0

Part 1: Effective user id

Checkpoint 1. Briefly explain what is an effective user id and why the two binary compiled from the same source code will print a different effective user id when they are executed.

Effective user id identifies the effective running user. The running binary will have the privilege as same as the user stated by the effective user id. Although the two binary is executed by the same user, they have the different effective user id settings and thus are running with different user permissions and privileges.

Checkpoint 2. Spot the difference of the two binary files, which part shows that their effective user id are different?

The execution privilege of the binary root shows an 's' instead of 'x' shows that it will run with an effective user id of the owner (and owner group) when executed.

Checkpoint 3. Briefly explain the usage of chown and chmod operation in the `Makefile`.

The chown command change the owner and group of the binary `root` to the user root The chmod command add the suid / sgid bit to the binary `root` and set the privilege to 755. This allow any person to run the binary `root` with the effective user id of root, which means everyone can run the binary `root` with the privilege of user root.

Checkpoint 4. What is a setuid / setgid bit and what are the difference of running the same binary with or without setuid / setgid bit?

It means that when a user execute that binary, the effective user id will be the user id of the owner (or owner group). Then the binary will be running with privileges as same as the file owner.

Checkpoint 5. What is the security concern if a vulnerable program is running with a setuid / setgid bit?

If a vulnerable program is owned by a super user and running with a setuid / setgid bit, the attacker can exploit the binary and run arbitary command with the privilege of the super user. This is an example of **privilege escalation**.

Checkpoint 6. Use the sticky bit technique to allow the `open` binary to read the file `secret`. State the command you have used.

```
sudo chown root:root open
sudo chmod 4755 open  / sudo chmod u+s open
```

Part 2: Simple buffer overflow

Checkpoint 1. How many user inputted characters are allowed to store in the array `password` without overflowing it?

The array `password` can accept at most 9 user inputted characters, because it still need a null character to end the string which will also occuply 1 character space.

Checkpoint 2. If the length of the user input exceed the size of `password`, where will strcpy() store the remaining characters?

The remaining character will overflow the stack and store at next memory location. In this case, it will store in the memory location of the integer correct.

Checkpoint 3. State the user input you have used and briefly describe why it is working.

Just provide an input larger than the buffer will corrupt the checking machinsm. The problem exists because the program does not check the user input size and carelessly handle the initialization of the variable `correct`. Also, the checking mechanism depends on the varialbe `correct` which is an integer. Any non-zero content of an integer is treated as true for the if statement. so overflowing the array `password` to make the variable `correct` become non-zero value will successfully corrupt the checking mechanism.

Exercise 1

Part 1: classic stack overflow

Checkpoint 1. How can you tell the program will run as setuid root, and how can you make a compiled program run as setuid root?

`ls -l` to list the file, if the program is owned by root and has an s symbol on the owner execution bit, then the program will run as setuid root.

`chown root:root <binary> && chmod +s <binary>` make a program run as setuid root

Checkpoint 2. Briefly explain the output of these tools, and how the compiler flags influence the output of `checksec.sh` (look at the `Makefile` to see how `gcc` was invoked).

- Rats tells you there is a buffer overflow vulnerability.
- Checksec tells you that there are no stack canaries and that the `nx`-bit protection (the writeable or executable bit) is disabled too.
- *RELRO* prevents overwriting library functions once the code is linked, *PIE* is a strong form of address space layout randomisation.

Checkpoint 3. Explain what your shellcode does and how you made it.

Saying you downloaded an `execve(/bin/sh,0,0)` shell code would have been enough.

The shell code try to simulate the status of register and memory when calling execve(/bin/sh,0,0)

It may use some tricks such as `xor`-ing to get a 0-byte, and pushing hex strings onto the stack for the `/bin/sh`.

Checkpoint 4. Explain how your exploit works.

If we run the program in GDB we can figure out how to get control of the instruction pointer by trying to print various length strings. If we print 160 bytes the last four bytes form the new address.

```
(gdb) run `perl -e 'print "A"x156, "\x01\x02\x03\x04"'`

Starting program: /home/user/Exercise-1.1/./noticeboard `perl -e 'print "A"x156, "\x01\x02\x

Program received signal SIGSEGV, Segmentation fault.
0x04030201 in ?? ()
```

From here we just need to jump to some shell code. My prefered way of doing this is to use an environment variable (they're stored on the stack).

From here you can write a program to leak the environment variable address:

```
#include<stdio.h>
#include<stdlib.h>
int main(void){printf("%08p\n", getenv("SHELLCODE"); return 0;}
```

If we run the program with its full path, and the filename is the same length as the vulnerable program it'll give us the address we want. Place that at the end of your buffer overflow and you're done.

Checkpoint 5. Provide your patch to fix the notice board program (use `diff -c <oldprogram> <newprogram>`).

Switching `strcpy` for `strncpy` and using a **length of 139 then setting the 140th byte to 0** will fix the bug.

Part 2: another vulnerability

Checkpoint 1. Identify the security flaw in the new version of `noticeboard.c`; explain what it allows and demonstrate an exploit that compromises the standard system security.

We still have a buffer overflow, and the path of the noticeboard file we append to is what we will overflow into.

Checkpoint 2 (optional). Briefly, explain how your root shell exploit works.

Many ways to do this: you could create an init script or service to launch a reverse shell. Read the -i options of bash for more details. Or you could corrupt authentication related file, like passwd / shadow / sudoers / etc. Alternately because the program is vulnerable to return oriented programming you could use construct a return oriented shellcode.

Checkpoint 3. Give a patch which fixes the second version of `noticeboard.c`.

Again, don't use `strcpy`! Always use the bounded (or safe) alternatives.

Exercise 2

Checkpoint 1. Explain the format of the messages sent by the client.

length message

Checkpoint 2. Provide a program (or shell script) which crashes the server remotely.

provide negative length

Checkpoint 3. Give a patch to fix the problem(s).

You could check for a negative number, or you could use proper Java bounded strings.

You can also provide a better exception handling for array index out of bound.

Exercise 3 (Advanced)

Checkpoint 1. How do you get the address of a `/bin/sh` string, and if you can't find one in memory how to inject one?

You can include it in environment variable, or make use of the one provided as parameter for printf function call.

Use rabin2 -z ret2libc to print all string address in the memory

You can also use objdump -s ret2libc to print out the memory content in different region of the memory. Try -j for a specific region.

You can also search for /bin/sh in some shared library.

Checkpoint 2. Where does the `system` function exist in libc? Where is it loaded in your program?

Use p system in gdb to locate the address after it has been loaded in the program. You will need to add a breakpoint in the program to do so.

Checkpoint 3. How do you call `system` as you return from the overflown function with your string as its argument?

Replace the return address with the address of `system`, then provide the `/bin/sh` string address after it as argument for the system function call.

Checkpoint 4. A program which crashes may leave a log file somewhere. You should also ensure your program exits cleanly. How do you do this?

Provide a function call to exit succesfully, for example `exit(0)`.

Exercise 4 (Optional)

Checkpoint 1. Identify the security flaw in the code, and provide the relevant CVE number.

CVE-2012-2110

It is casting an unsigned long into an signed int as part of getting a length: so if the top bit of the int is set it will become a negative number.

Checkpoint 2. Briefly summarise the problem and explain and why it is a security flaw.

It is a heap overflow (leading to arbitrary code execution), ultimately caused by an improper type cast.

A full explanation can be found by the discoverer, Travis Ormandy, on the Full Disclosure mailing list: http://seclists.org/fulldisclosure/2012/Apr/210

Checkpoint 3. Give a recommendation for a way to repair the problem.

Upgrade the version of OpenSSL, as recommended by OpenSSL at the end of the disclosure notice.

Checkpoint 4 (very optional). Build a *proof-of-concept* to demonstrate the security flaw and explain how it might be exploited; check that your repair (or the current released version) prevents your attack.

There is some code to start from in Travis's disclosure: http://seclists.org/fulldisclosure/2012/Apr/210