Secure Programming Solutions 1: Data Corruption

School of Informatics, University of Edinburgh

2pm-5.30pm, 31st January 2014

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?

`chown root:root program && chmod +s program`

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.

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. 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.

Use `nc` to write a 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.

Exercise 3 (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