Secure Programming Lab 2: Data Corruption

School of Informatics, University of Edinburgh

10am-1pm, 6th February 2017

Exercise 0

Part 1: Hash puzzle

Checkpoint 1. The program has an obvious memory corruption vulnerability. Briefly describe the vulnerability, including how it can be attacked and possible consequence of the attack.

- Scanf reads in 29 characters to an array which holds only 16.
- Therefore can control all of hash memory apart from the last 3 bytes which will be 00, da, ac.
- Need to find a random string that hashes to something ending in these bytes.
- Input is password, a bunch of nil to fill up to 16 bytes, then the password hash characters. (No mark if limited the password to length 15 only)
- The input cannot contain whitespace character.
- By this exploit, an attacker can get authenticated with random password of length 1 to 15 that has a hash value end with 00, da, ac

Checkpoint 2. Create an exploit script that takes the path of the program as its first argument and make the program prints the message Correct Password!

See exploit.sh

Checkpoint 3. Provide a patch file to fix the vulnerable program (use diff -c <oldprogram> <newprogram>).

Check the size of input or change strcpy to strncpy. Remember to leave 1 character for ending null character. Also remember that strncpy will not add null character automatically if source is longer than destination buffer. You need to add it yourself.

Part 2: Address puzzle

Checkpoint 1. What command can help you locate the address of a function?
objdump -d ./noticeboard

Checkpoint 2. What is the address of the correct() function?

Depends

Checkpoint 3. Force a segmentation fault to the program, and state the size of buffer between the local variable password and the return address.

input more than 23 characters as input

Checkpoint 4. After the above steps, you should able to force the program to call the correct() function, but you may see an extra segmentation fault afterwards, why is it?

Because the program fail to jump back to normal running procedure and continue executing through the memory and soon hit an illegal address or invalid assembly code in the memory.

Checkpoint 5. Fix your input to avoid the segmentation fault. (Hints: You may need the address of the exit() function) Add in the address of the exit function after the input to false the program exit after jumping to the correct() function.

Exercise 1

Part 1: classic stack overflow

Checkpoint 1. 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 2. 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\x03\x04";'`
Program received signal SIGSEGV, Segmentation fault.
0x04030201 in ?? ()

From here we just need to jump to some shell code. My preferred 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:

```c
#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 3.** 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

Passwords for exploits

- Exercise 0.1: smashing hash
- Exercise 1.1: strcpy does not check buffer size
- Exercise 1.2: nc -1 1337 | bash
- Exercise 2: len<0
- Exercise 3: solar operator