Understanding Software Vulnerabilities: Injection Attacks, Race Conditions, and Format String Attacks

License

Contents

General notes about the labs

Often the lab instructions are intentionally open ended, and you will have to figure some things out for yourselves. This module is designed to be challenging, as well as fun!

However, we aim to provide a well planned and fluent experience. If you notice any mistakes in the lab instructions or you feel some important information is missing, please feel free to add a comment to the document by highlighting the text and click the comment icon ( ), and I (Cliffe) will try to address any issues. Note that your comments are public.

The labs are written to be informative and, in order to aid clarity, instructions that you should actually execute are generally written in this colour. Note that all lab content is assessable for the module, but the colour coding may help you skip to the "next thing to *do*", but make sure you dedicate time to read and understand everything. Coloured instructions in *italics* indicates you need to change the instructions based on your environment: for example, using your own IP address.

You should maintain a **lab logbook / document**, which should include your answers to the questions posed throughout the labs (in this colour).

Preparation

As with all of the labs in this module, start by loading the latest version of the LinuxZ template from the IMS system. If you have access to this lab sheet, you can read ahead while you wait for the image to load.

> To load the image: press F12 during startup (on the boot screen) to access the IMS system, then login to IMS using your university password. Load the template image: LinuxZ (load the latest version).

Once your LinuxZ image has loaded, log in using the username and password allocated to you by your tutor.

The root password -- **which should NOT be used to log in graphically** -- is "tiaspbiqe2r" (**t**his **i**s **a** **s**ecure **p**assword **b**ut **i**s **q**uite **e**asy **2** **r**emember). Again, never log in to the desktop environment using the root account -- that is bad practice, and should always be avoided.

These tasks can be completed on the LinuxZ system. Most of this lab could be completed on any openSUSE or using most other Linux distributions (although some details may change slightly).

You may need to install 32bit development packages. On openSUSE: "`sudo zypper in gcc-32bit`".

Recap: programming errors

It is very hard to write secure code. Small programming mistakes can result in software vulnerabilities with serious consequences. The two main categories of software flaws are those caused by:

- design problems: such as, a programmer not thinking through the kind of authentication required
- implementation problems: such as, a programmer accidentally introducing a bug by using an insecure library method or trying to store too much data into a variable

Common programming errors that lead to security flaws include:

- Memory errors and bounds checking, such as buffer overflows
- Using input without sanitising it: this can lead to flaws such as command injection
- Race conditions
- Misconfigured/used access controls and other security mechanisms
- Misuse of pointers and strings

This lab focuses on looking at and understanding many of these programming flaws.

Un-sanitised input and command injection

A program is vulnerable to command injection if you can change the behaviour of software by inserting commands into input that get interpreted as commands for the program to execute. The result is similar to the outcome of the previous lab, although in with command injection an attacker does not need to overflow into a variable: they simply enter data into a variable that is misused by the programmer.

Open a terminal console.

> One way to do this is to start Konsole from KDEMenu → Applications → System → Terminal → Konsole.

Move to a new directory for our code:

```
mkdir ~/code
```

```
cd ~/code
```

Create and edit a new file “injectionattack.c”:

```
vi injectionattack.c
```

Reminder: Vi is ‘modal’: it has an insert mode, where you can type text into the file, and normal mode, where what you type is interpreted as commands. Press the “i” key to enter “insert mode”. Type the below C code:

```
#include <stdio.h>
int main() {
        char name [20];
        char command [100];
        printf("What is your name?\n");
        scanf("%19[^\n]s", &name);
        sprintf(command, "echo Hello %s; echo The time is "
        "currently:; date", name);
        system(command);
}
```

Exit back to “normal mode” by pressing the Esc key. Now to exit and save the file press the “:” key, followed by “wq” (write quit), and press Enter.

> Tip: alternatively if you want to copy and paste, rather than using vi, run “cat > hello.c” and paste into the terminal, then press Ctrl-D to end the file.

Compile the program:

```
gcc injectionattack.c -g -m32 -o injection
```

Run the program:

```
./injection
```

Try entering your own name, and confirm that the program works as expected in this case.

Try entering a long input, and confirm that the program works as expected in this case.

Note that this code is vulnerable to command injection. To understand the attack, it is important to understand the way this code works. The sprintf line builds a string based on a format string: it is like printf, except that it writes to a string variable rather than to the console. It is used in this case to create the output for the user, by creating Bash commands, which the next line executes, by passing this through to the Bash shell. The final command that is stored in `command` looks like:

```
echo Hello Cliffe; echo The time is currently:; date
```

Note that the semicolon (";") is used in Bash to separate commands, so this is effectively going to run three commands via Bash:

```
echo Hello Cliffe

echo The time is currently:

date
```

Add a comment above each line of code, with an explanation of its purpose. For example, "`// this line reads 19 characters from the user, and stores them in name followed by a terminating \0 character`".

Note that the user can type up to 19 characters, and they will be included in the above command sequence.

Can you think of a way of subverting the behaviour of this program, so that it prints the contents of /etc/passwd?

Confirm the command injection vulnerability...

Run the program:

```
./injection
```

Challenge: Enter an input that makes the program print out /etc/passwd

Solution:

SPOILER ALERT! SPOILER ALERT! SPOILER ALERT!

Since the semicolon can be used to separate commands, you can simply include a semicolon in your input, which starts a new Bash command.

You can exploit this vulnerability as follows:

```
./injection

;cat /etc/passwd
```

This makes the final command run by Bash:

```
echo Hello ;cat /etc/passwd; echo The time is currently:;
date
```

SPOILER ALERT! SPOILER ALERT! SPOILER ALERT!

Make sure you understand the answers to these questions:

- What is the maximum amount of damage that this attack against this program could theoretically cause?
- Would the security threat be higher if the program was: setuid? What about a server that people connected to over a network?

Note that this attack is the same way that SQL injection works, the difference is that here we have commands sent to bash, rather than to a Web server database. Command injection attacks can apply to any type of programming, web, application, system, or otherwise, whenever input from a user is used as part of an interpreted command.

The solution is that all data that comes from an untrusted source must be validated and sanitised before use.

- What is an untrusted source? Examples?
- What sources would you trust enough that you wouldn't check it before processing?

Validation

Validation involves checking that data is in the format you expect.

Create and edit a new file “injectionattack_validated.c”:

```
vi injectionattack_validated.c
```

Enter the below C code:

```
#define IS_VALID 1
#define NOT_VALID 0
#include <stdio.h>
#include <string.h>

int main() {
        char name [20];
        char command [100];
        printf("What is your name?\n");
        scanf("%19[^\n]s", &name);
        if(validate(name) == IS_VALID) {

sprintf(command, "echo Hello %s; echo The time is currently:;"
                  "date", name);
                system(command);
        } else {
                printf("Invalid input!\n");
        }
}

int validate(char* input) {
        int i;
        for(i=0; i < strlen(input); i++) {
                if(!isalpha(input[i])) {
                        return NOT_VALID;
                }
        }
        return IS_VALID;
}
```

The above code adds validation to the previous code, by examining each character in the input, and if any character is not a letter, it refuses to run the command.

Add a comment above each new line of code, with an explanation of its purpose.

Compile and run the program.

Attempt your command injection, and confirm that the attack no longer works.

Creating code patches using diff

The common Unix commands `diff` and `patch` can be used to apply delta changes to a file. A delta describes the changes made, rather than an entire file.

The `diff` command outputs a delta (or "diff"), that can be applied by someone else to a suitably similar file, using `patch`.

Create a patch:

```
diff -u injectionattack.c injectionattack_validated.c >
injectionattack.addvalidation.patch
```

This has directed the output from diff into a new file, which adds validation to the original code.

View the diff:

```
less injectionattack.addvalidation.patch
```

Make a copy of injectionattack.c called injectionattack_update.c:

```
cp injectionattack.c injectionattack_update.c
```

Apply your patch to the copy:

```
patch injectionattack_update.c <
injectionattack.addvalidation.patch
```

View the updated version:

```
less injectionattack_update.c
```

You have successfully generated a patch, and used it to update an old version of the code to apply a set of changes.

Sanitisation

Sanitisation involves *removing* any potentially dangerous formatting / content from a variable. This can 'fix' the input, to make it safe for use.

Make a copy of injectionattack_validated.c called injectionattack_sanitied.c:

```
cp injectionattack_validated.c injectionattack_sanitise.c
```

Edit injectionattack_sanitied.c to sanitise the input, so that it is safe for use, and prints the name input, even if the user enters invalid characters, such as ";".

For example, if the user enters ";/etc/password" this could be changed to "etcpasswd" or "--etc-password".

Hint: replacing any invalid character with "**-**" is probably the easiest solution.

Compile, run, and test your program.

Once you are satisfied with your solution, create a patch:

```
diff -u injectionattack_validated.c
injectionattack_sanitised.c >
injectionattack.addsanitisation.patch
```

Share your patch with a classmate, and apply a different patch from another class mate. Does it fix the security problems? Try to break the new version of the program, look for other possible errors.

Blacklisting vs whitelisting values

Try entering this as input to you and your classmate's new version of the program:

```
./injectionattack_sanitised
```

```
& cat /etc/passwd
```

If this attack succeeds, even after you updated the code to defend against the previous ";" attack, then this means you likely based your solution on blacklisting certain values, rather than checking against a whitelist of acceptable values.

The safest approach is to remove everything you don't actively expect, rather than removing what you know you don't want. (The same goes for validation, which should check for a set of valid characters, rather than a set of invalid characters.)

Why is checking against a whitelist the safer approach?

More command injections

Create and edit a new file "injectionattack_validated.c":

```
vi injectionattack_bash_escaped.c
```

Enter the below C code:

```
#include <stdio.h>
int main() {
        char name [20];
        char command [100];
        printf("What is your name?\n");
        scanf("%19[^\n]s", &name);
        sprintf(command, "echo 'Hello %s'; echo The time is currently:;
date",
          name);
        system(command);
}
```

Compile and run this code.

Test a simple injection attack against this code.

Does it succeed? Why not?

Answer: this is due to the single inverted commas: the contents of which are considered arguments to the echo command. So after an attempt at injection the command becomes:

> echo ‘Hello ;cat /etc/passwd’; echo The time is currently:; date

Which is effectively going to run three commands via Bash:

```
echo ‘Hello ;cat /etc/passwd’

echo The time is currently:

date
```

Challenge: perform command injection against this version.

> Hint: think about what the resulting command will look like to bash. Also think about how you can use single quotes (“‘”).

Is the sprintf function call safe? (Hint: man sprintf)

Race conditions

If you rely on something on the system to stay in the same state between two lines of code, you probably have a race condition. It is hard not to be dependent on the sequence or timing of other events, but getting this wrong can have security implications.

Time of check to time of use, *TOCTTOU (“tock too”),* flaws can cause security vulnerabilities. TOCTTOU occurs when something changes between when a condition is checked, and the resulting action happens. For example, if a program checks that a file exists (or someone has permissions to it), then goes on to do something. This could introduce a security flaw, since an attacker could make carefully timed changes to the system between these two events.

Create and edit a new file “race_condition.c”:

```
vi race_condition.c
```

Enter the below C code:

```
#include<stdio.h>
#include<sys/stat.h>
int main()
{
        char tmpname [20] = "/tmp/not_so_random";
        char command [100];

        struct stat buf;
        int ok = stat(tmpname, &buf);

        sleep(5);

        if(ok == 0) {
                printf("Temporary file already exists, "
                  "we need a new name...");
        } else {
                printf("File does not exist, writing...");
                sprintf(command, "echo Hello > %s", tmpname);
                system(command);
        }
}
```

This code checks that a temporary file named `/tmp/not_so_random` does not exist before using the file… but what if something happens between the check and the following code?

The 5 second delay in the code between the check and subsequent action is an exaggeration (the actual delay would be *much* shorter), but it helps us explore the flaw.

What if a file was put there in between, and it was a symlink?!

A *symlink* (AKA symbolic link) is a file that just points to another file, operations on symlinks are typically carried out on the target.

> Remember, a hardlink is when two file names share the same inode, and are therefore referring to the same file.
>
> Tip: “`man ln`”

Create a file that belongs to “student”, and that only student can access:

```
cp /etc/passwd /tmp/students-file

chmod 600 /tmp/students-file

sudo chown student /tmp/students-file
```

We now have a file /tmp/students-file that we cannot access ourselves, and want to make the program write to it.

Compile the program:

```
gcc race_condition.c -o race_condition
```

Cache the student’s password for sudo (will prompt for password, and remember that you have authenticated):

```
sudo -u student test
```

> Hint: you know root’s password, so you can reset student’s password first, if you don’t already know it.

This one-line Bash command starts the program as the victim (imagine the victim is running the program) and does the switcheroo between the program checking if the file exists then writing to it:

```
sudo -u student ./race_condition & sleep 1; ln -s /tmp/
students-file /tmp/not_so_random
```

It starts the program, then after student has been running the program for 1 second (after it has checked the file), it creates a symlink in the file's place, pointing to the target.

Check the students private file:

```
sudo less /tmp/students-file
```

The target file is modified! The very subtle timing flaw in the program means that if an attacker can create a file at just the right time, it can make the program act in unexpected ways. Image if the attacker used this attack to overwrite important system files, by subverting the behaviour of a program that root was running.

Preventing race conditions by using exception handling

One way of reducing these types of errors is to use exception handling rather than checking separately. For example, use an "open" function call, with a flag to tell the function to return an error if the file exists.

Modify race_condition.c so that it opens the file using a flag that tells it to fail if the file exists.

> Hint: man open, and write to the file using C library calls rather than the "system()" function call

Compile, run, and test that this fixes the race condition.

Look up and read about other types of race conditions such as those caused by multiple threads sharing the same data.

Format string attacks

Some functions such as `printf()` receive a format string, followed by multiple variables to display, as specified in the format string.

For example, to print the a message that includes a string (text) and an integer (whole number), the C code would be:

```
printf("Hello %s, you entered %d", string, number);
```

When `printf()` is run, it reads the format string, then looks to the stack for the data to replace the format string values. The “%s” is replaced by a string from the stack (`input`), and the “%d” is replaced by an integer (`number`).

An overview of format string specifiers:

- %d: signed integer (**d**ecimal)
- %f: double (real **f**loating point number)
- %x: (he**x**idecimal -- “%08x” can be used to print 8 digits/bytes of memory)
- %s: (**s**tring)
- %n: number of bytes written by printf (writes to memory)

The correct way of printing only the string, is:

```
printf("%s", string);
```

The lazy and vulnerable (**bad**) way would be to use code like this:

```
printf(string);
```

If a lazy programmer passes user input *as the format string*, an attacker can use clever tricks to view the contents of the stack, or even write to memory! This can lead to serious security vulnerabilities.

Create and edit a new file “format_string_attack.c”:

```
vi format_string_attack.c
```

Enter the below C code:

```
#include <stdio.h>
int main() {
        int allowed_access = 0;
        int secret_number = 42;
        char name [100];
        printf("What is your name?\n");
        fgets(name, sizeof(name), stdin);
        printf("You entered:\n");
        printf(name); // oops!
        if(strcmp(name, "secret\n") == 0) {
                allowed_access = 1;
        }
        if(allowed_access != 0) {
                printf("\nThe secret number is %d\n", secret_number);
        } else {
                printf("\nNot telling you the secret\n");
        }
}
```

Compile this program:

```
    gcc format_string_attack.c -g -m32 -o format_string_attack
```

Try running the program, and enter the name “Bob”.

Run it again and enter “secret”. Note that the program is designed so that only someone called “secret” will be told the secret number. Ignoring the obviously weak authentication, lets focus on how we can exploit the line in bold — “`print(name);`” — using a format string attack.

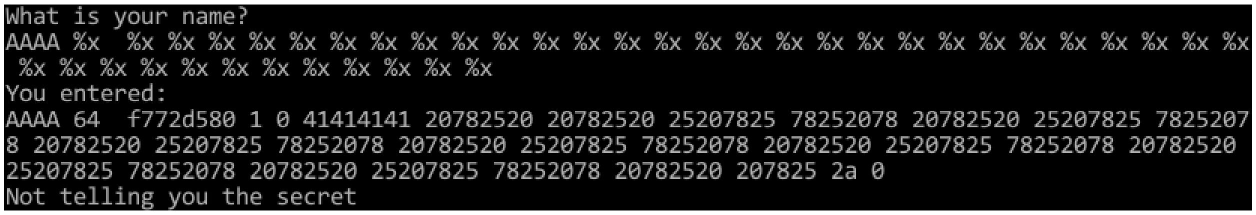
So what happens if the user enters a format string specifier?

Try running the program, and enter the name “**%d**”.

This results in a call to `printf()` which prompts it to read a number off the stack. Unfortunately there is no number passed as a parameter to `printf()`; however, `printf()` does not know this, and diligently reads from the stack, and displays a mysterious number.

Try running the program, and enter the name:

“AAAA %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x”. (%x appears 31 times).



```
What is your name?
AAAA %x  %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x
 %x %x %x %x %x %x %x %x %x %x %x %x
You entered:
AAAA 64  f772d580 1 0 41414141 20782520 20782520 25207825 78252078 20782520 25207825 7825207
8 20782520 25207825 78252078 20782520 25207825 78252078 20782520 25207825 78252078 20782520
25207825 78252078 20782520 25207825 78252078 20782520 207825 2a 0
Not telling you the secret
```

The result is that `printf()` works its way back down the stack, reading values and printing them in hex form. Note that the “41414141” represents the first “AAAA” from the input, which shows that we are successfully tricking the program into displaying the stack. The values before the 0x41414141 are not intended for `printf()`; but `printf()` works its way through whatever is on the stack, as though they were parameters passed to `printf()`.

So what does this mean?: an attacker can read values off the stack. In this case the output includes the `secret_number`, 0x2a which equals decimal 42! Following that is the value 0, which specifies that we don't have permission to access that information.

It is also possible to directly access specific memory areas and write to the stack. However, this is outside the scope of this lab.

Here are some good online tutorials:

> [Avoiding security holes when developing an application - Part 4: format strings]
>
> [Format String Exploitation Tutorial]

What format string specifier can be used to write arbitrary data?

Extra challenge (optional): exploit the format string vulnerability in the above program to overwrite the value of `allowed_access` to gain access to the `secret_number` printing behaviour.

Conclusion

At this point you have:

- Exploited command injection errors
- Written C code for doing sanitisation
- Used diff and patch to create and apply delta changes
- Exploited code with time of check to time of use race conditions
- Coded a solution based on exception handling
- Exploited format string vulnerabilities to read data from the stack
- You may have completed the additional challenge of exploiting the format string vulnerability further, by writing to a variable stored on the stack (if so, congratulations!)

Well done!