Bug Hunting Using Fuzzing and Static Analysis

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.

Using the VM download script (as described in the previous lab), download and **start** these VMs:

- Kali Linux (Bridged and Host Only)
  username:root password:toor
- Windows XP (any)

Most of this lab could be completed on any Linux distribution with CppCheck, gcc-32bit, gdb, and Spike (although some details may change slightly depending on the distribution used). Spike can be relatively hard to install, so using Kali Linux is the simplest approach.

Introduction to bug hunting

There are many kinds of programming and design mistakes that can lead to serious security flaws. Auditing software for flaws is an important component of secure software development and testing. After software has been created (and during the development process), programs can be tested *with or without access to the source code* to look for vulnerabilities. If a security researcher can find a flaw that the software authors do not know about, this can have serious security ramifications. If they then weaponise the attack, and create an exploit to attack, this is known as a ***zero-day exploit***, if there is currently no fix available to defend against the attack.

Zero-day exploits can be used for malicious purposes, such as building worms or in targeted attacks, and can be sold on the blackmarket. White hat hackers (the good guys, like you!) will generally follow *responsible disclosure*, and notify the vendor and give them time to fix the problem before going public with the information, after a fair amount of time. Many of the larger vendors offer rewards for reporting vulnerabilities.

The most popular approaches to bug hunting

When you have access to the source code you can perform:

- *manual code review*, to look for security mistakes
- *static analysis*, which is when you use software to automatically analyse the code

If you do not have access to the source code, you can use:

- *reverse engineering* to transform the the compiled code (program) to get a (harder to read) version of the source code — binary static analysis is possible analysing a program without the original source code
- *fuzzing* to do blackbox testing to find faults by feeding in variations of unexpected input into a program in an attempt to uncover unexpected behaviour

Manual code review

Being able to spot errors in code is an important skill for developers and security professionals.

Spot the error:

```
#include <stdio.h>
```

```
#include <string.h>
#include <stdlib.h>
#define MAXPASS 7
int main()
{
    char correct_pass[7] = "SecreT\0";
    char input_pass[7];
    do{
        scanf("%s", &input_pass);
        // uncomment the next line to see what is
        // happening...
        //printf("You entered %s, pass is %s\n", input_pass, correct_pass);
        if(strcmp(input_pass, correct_pass) == 0) {
            printf("Access granted\n");
        } else {
            printf("Access denied\n");
        }
    } while(strcmp(input_pass, correct_pass) != 0);
}
```

Can you spot the error, without compiling and running the code?

On Kali Linux VM

Save, compile, debug, and test this program.

Try uncommenting the printf line, and see what happens when various inputs are used.

Is this program secure? Why not?

Run and manually exploit this code.

Update the code to prevent the attack.

Spot the error:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXPASS 7
int main()
{
    char *correct_pass, *input_pass;
    correct_pass = malloc(sizeof(char)*MAXPASS);
    input_pass = malloc(sizeof(char)*MAXPASS);
    strcpy(correct_pass, "SecreT\0");

    printf("Please enter the password: ");
    scanf("%7s", input_pass);
    // uncomment to see what is happening...
    printf("You entered %s, pass is %s\n", input_pass, correct_pass);
    if(strcmp(input_pass, correct_pass) == 0) {
        printf("Access granted\n");
    } else {
        printf("Access denied\n");
    }
}
```

Can you spot the error, without compiling and running the code?

> Hint: defensive programming and memory allocation.

Update the code to prevent the problem.

Static analysis

On Kali Linux VM

Download and install cppcheck, a FOSS static analysis tool:

```
apt-get install cppcheck
```

Run cppcheck against the previous examples:

```
./cppcheck [directory containing code]
```

Did it detect each of them? How accurate are the descriptions?

Save this code as "integer_overflow.c":

```
#include <stdio.h>
int main()
{
    int num;
    printf("Please enter the a number: ");
    scanf("%i", &num);
    printf("You entered %d\n", num);
    return 0;
}
```

Compile the program as "intover".

Run it and make it crash:

```
perl -e 'print "5"x2100000' | ./intover
```

Note that Perl is used here to simply feed in lots of "5"s (2100000 of them).

Use cppcheck on this code (integer_overflow.c), and use its suggestion to fix and then test the error again.

Was cppcheck's suggestion helpful?

Use cppcheck to analyse the code in the previous examples.

Which of the above flaws was cppcheck able to detect?

The gcc compiler has extra static analysis compiler checks it can do. Options such as "`-Wextra`" can be used to check for certain types of errors.

Compile the above examples with the "`-Wextra`" flag.

What kinds of security errors does gcc detect with the code examples given?

It does detect a previously mentioned library call, can you find any other examples?

Run cppcheck against examples from the previous labs.

Does it detect all the flaws? Does it detect command injection attacks?

Run cppcheck against its own code.

CppCheck also has a graphical interface, which you may like to try. However, this will involve downloading the source code and compiling.

Fuzzing

Fuzzing involves sending deliberately unexpected data as input to a program in an attempt at discovering programming flaws. Fuzzers are often used by security researchers to find new vulnerabilities in software.

Three popular network fuzzers are Spike, Sulley, and Peach. If you are testing Web apps or GUI programs, other specialised fuzzers exist, such as the Web app fuzzer in Burp Suite.

Spike is a popular fuzzer written in C. It works by reading in a scripted template file describing the normal protocol (what is normally expected as input), and it automatically feeds in data based on its collection of strings that are likely to crash the program by deviating from what is expected. For example, it may try invalid characters/symbols, and longer than expected inputs.

On your Kali Linux VM:

Start Wireshark, and monitor the traffic on the "lo" interface to view the stream of traffic.

Next, lets see Spike in action…

Run Ncat as a network service on port 4444:

```
ncat -vlk -p 4444
```

Open a new terminal tab (Ctrl-T).

Create and edit a file named "my_first_spike.spk". Enter this text:

```
s_string("Hello, world!");
```

This spike script simply sends the greeting string to the server.

To summarise:

- `s_string()` sends a string

Spike has a number of interpreters for interacting with programs.

Since we are using Ncat to simulate a network service, we can use one of Spike's most common interpreters, "generic_send_tcp".

The usage for generic_send_tcp is:

```
Usage: ./generic_send_tcp host port spike_script SKIPVAR
SKIPSTR
```

Run our spike script:

```
generic_send_tcp IP-address 4444 my_first_spike.spk 0 0
```

View the Ncat output to confirm that Spike sends out the expected text.

Use Wireshark, and follow the TCP stream, to view the behaviour of the fuzzer.

Note that it tries a number of different network connection attempts, sending this same payload data (with different TCP options).

At this point the fuzzer is not really doing anything unexpected.

Edit "my_first_spike.spk". Change the first line to:

```
s_string_variable("Hello, world!");
```

Try running this new version of the script:

```
generic_send_tcp localhost 4444 my_first_spike.spk 0 0
```

Watch the Ncat output in the previous tab.

As we can see the spike script now involves some visible "fuzziness". The first time the text "Hello, world!" is sent, and subsequently other malformed versions are sent.

In this case we are not going to find anything that breaks our `Ncat` server, so lets move on to fuzzing some vulnerable software.

(Still on Kali)

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

```
vi another_vulnerable_service.c
```

Enter the below C code:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>

int main()
{
  // variables
  int sock_fd, connection_fd;
  const int buffer2_len = 50;
  const int buffer1_len = 1000;
  char buffer2[buffer2_len];
  char buffer1[buffer1_len];
  struct sockaddr_in addr;

  // socket details
  addr.sin_family = AF_INET;
  addr.sin_addr.s_addr = htons(INADDR_ANY);
  addr.sin_port = htons(5555);

  // create socket
  sock_fd = socket(AF_INET, SOCK_STREAM, 0);

  // set socket details
  bind(sock_fd, (struct sockaddr *) &addr, sizeof(addr));

  // start listening for connections
  listen(sock_fd, 10);
  // keep listening for new connections
  while(1) {
    // accept connection
    connection_fd = accept(sock_fd, NULL, NULL);
    // get name
    bzero(buffer1, buffer1_len);
    sprintf(buffer1, "What is your name?\r\n");
    write(connection_fd, buffer1, strlen(buffer1)+1);
```

```
    bzero(buffer1, buffer1_len);
    read(connection_fd, buffer1, buffer1_len);
    printf("Connection from %s", buffer1);
    // reply
    bzero(buffer2, buffer2_len);
    snprintf(buffer2, buffer2_len, "Hello, %s\r\n", buffer1);
    write(connection_fd, buffer2, strlen(buffer2)+1);
    // if name starts with Cliffe, ask for another string, and print back
    if(strncmp(buffer1, "Cliffe", 6) == 0) {
      printf("(Access granted)\n");
      bzero(buffer1, buffer1_len);
      sprintf(buffer1, "Access granted... Please enter a string:\n");
      write(connection_fd, buffer1, strlen(buffer1)+1);

      bzero(buffer1, buffer1_len);
      read(connection_fd, buffer1, buffer1_len);
      printf("Received string: %s\n", buffer1);

      strcpy(buffer2, buffer1);
      sprintf(buffer1, "You entered: %s\r\n", buffer2);
      write(connection_fd, buffer1, strlen(buffer1)+1);
    }

    close(connection_fd);
  }
}
```

Compile this program to “another_vulnerable_service”:

```
gcc -g -m32 -fno-stack-protector -z norelro
another_vulnerable_service.c -o another_vulnerable_service
```

Start our vulnerable service (listening on port 5555).

```
./another_vulnerable_service
```

Connecting to your vulnerable service:

Open a new terminal tab (Ctrl-Shift-T).

```
ncat -v 0.0.0.0 5555
```

Type some text and press Enter. This sends the text to the server, which responds with the output from your program.

Create and edit a file named "my_name_spike.spk". Enter this text:

```
printf("Fuzzing…\n");
s_readline();
s_string_variable("Bob");
s_string("\r\n");
spike_send();

s_readline();
s_readline();

s_string_variable("A string!");
s_string("\r\n");
s_readline();
spike_send();
```

Note that:

- `printf()` writes to the local console
- `s_readline()` reads and displays a line of response
- `s_string_variable()` writes a string that is fuzzed
- `spike_send()` sends what is in the buffer now
- The best way to know what commands are available is to look at the source code of spike.h (if you like, you can download the source code from here)

Try running the spike script to test the vulnerable program:

```
generic_send_tcp localhost 5555 my_name_spike.spk 0 0
```

The script will try to run through a large list of fuzzed inputs, in an attempt to crash the program.

You may find that the fuzzer does not find a way of crashing the program.

Edit the spike script, so that the fuzzer gets past the first condition within the code.

> Hint: edit the spike script so it passes the string comparison.

Try running the spike script again:

```
generic_send_tcp localhost 5555 my_name_spike.spk 0 0
```

If you have created an effective spike, the program will eventually crash after some fuzzing. When the program crashes, the fuzzing will stop, and the number of attempts will be shown. This can be used to determine the string that was used to crash the program.

View the traffic in Wireshark, and determine the last input that the fuzzer sent.

Restart the service (in the previous tab):

```
./another_vulnerable_service
```

Connect manually with Ncat:

```
ncat -v 0.0.0.0 555
```

Enter the input (possibly multiple lines) that the fuzzer used to crash the program.

Make sure you can reproduce the crash.

Save the crash inducing input (multiple lines) into a text file named "**fuzzed**".

Run the program in a debugger, feeding in the fuzzed input:

```
gdb ./another_vulnerable_service
```

```
run < fuzzed
```

```
backtrace
```

Display the value stored in EIP.

Explain the value and its significance.

Resume the fuzzing at the point that it stopped for the crash, by changing the last two arguments (previously you used "0 0"). The first number represents the fuzzing variable (such as string_variable) to skip to, and the second number is the fuzzing iteration to skip to for that variable (for example, "AAAAAAA").

In order to fuzz more complex programs, the spike script file needs to have added complexity to send certain parts of the input as expected, and focus on fuzzing the inputs that are most likely to trigger flaws:

- Altering input lengths and command options
- Altering integers, to test for boundary conditions and outliers
- Command injections, invalid characters and so on

Fuzzers are very good at finding shallow bugs. However, it can be quite hard for a fuzzer to find deep bugs (behind a number of conditional code statements), since the fuzzer needs to try to get good code coverage, to test the behaviour of all the possible control flows. However, complete coverage can be impractical or impossible for complex programs.

One of the most effective ways to fuzz a network program is to use a sniffer to record examples of normal traffic (and/or study protocol RFC specifications), and then create a fuzzer that follows the normal expected flow of communication, with fuzzed input for the situations that are most likely to be vulnerable.

Fix the security error(s) in another_vulnerable_service.

Also, consider fixing the error handling: for example, gracefully exit with an error message if the port is already in use. And if you want more practice writing C (and with multithreading), update the program to accept multiple connections at once (search for example code online).

Fuzzing an FTP server

On your Windows VM:

Download a vulnerable FTP server: FreeFloat FTP.

> http://www.exploit-db.com/search/?action=search&filter_description=freefloat

Save the program, and start it in a Windows VM.

On your Kali Linux VM:

Run an FTP spike script:

```
generic_send_tcp Win_IP_address 21 /usr/share/spike/audits/
FTPD/ftpd1.spk 0 0
```

Does it crash the program? What input crashed the program?

Hint: using Wireshark may be the easiest way.

Have a look through the spike script file, and try to understand how it works.

Another fuzzer is BED (Bruteforce Exploit Detector). Try running BED against FreeFloat FTP:

Restart the FTP server on your Windows VM, if it has crashed.

```
bed.pl -s FTP -t Win_IP_address -u anonymous -v anonymous
```

Does it crash the program? What input crashed the program?

Metasploit has an FTP fuzzing module, try using it:

Restart the FTP server on your Windows VM, if it has crashed.

```
msfconsole
```

```
msf > use auxiliary/fuzzers/ftp/ftp_pre_post
```

```
msf auxiliary(ftp_pre_post) > set RHOSTS Win_IP_address
```

```
msf auxiliary(ftp_pre_post) > exploit
```

Does it crash the program? What input crashed the program?

Have a look at the Metasploit module, and try to understand how it works.

```
less /usr/share/metasploit-framework/modules/auxiliary/
fuzzers/ftp/ftp_pre_post.rb
```

or **view the code online**

If you are intending to choose the fuzzing option for your assignment, you may find the following resources particularly helpful.

Further resources for learning about fuzzing:

A tutorial for Spike:

- Part 1: http://resources.infosecinstitute.com/intro-to-fuzzing/
- Part 2: http://resources.infosecinstitute.com/fuzzer-automation-with-spike/

A good overview of the most popular fuzzers available: http://www.blackhat.com/presentations/bh-usa-07/Amini_and_Portnoy/Whitepaper/bh-usa-07-amini_and_portnoy-WP.pdf

A list of fuzzers: http://pentest.cryptocity.net/fuzzing/

Fuzzing vs static analysis Defcon presentation slides: http://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-west.pdf

Conclusion

At this point you have:

- Manually audited C code for harder-to-spot errors
- Used static analysis tools to detect and help fix security issues
- Used various fuzzers to test network programs for serious security flaws

Well done!