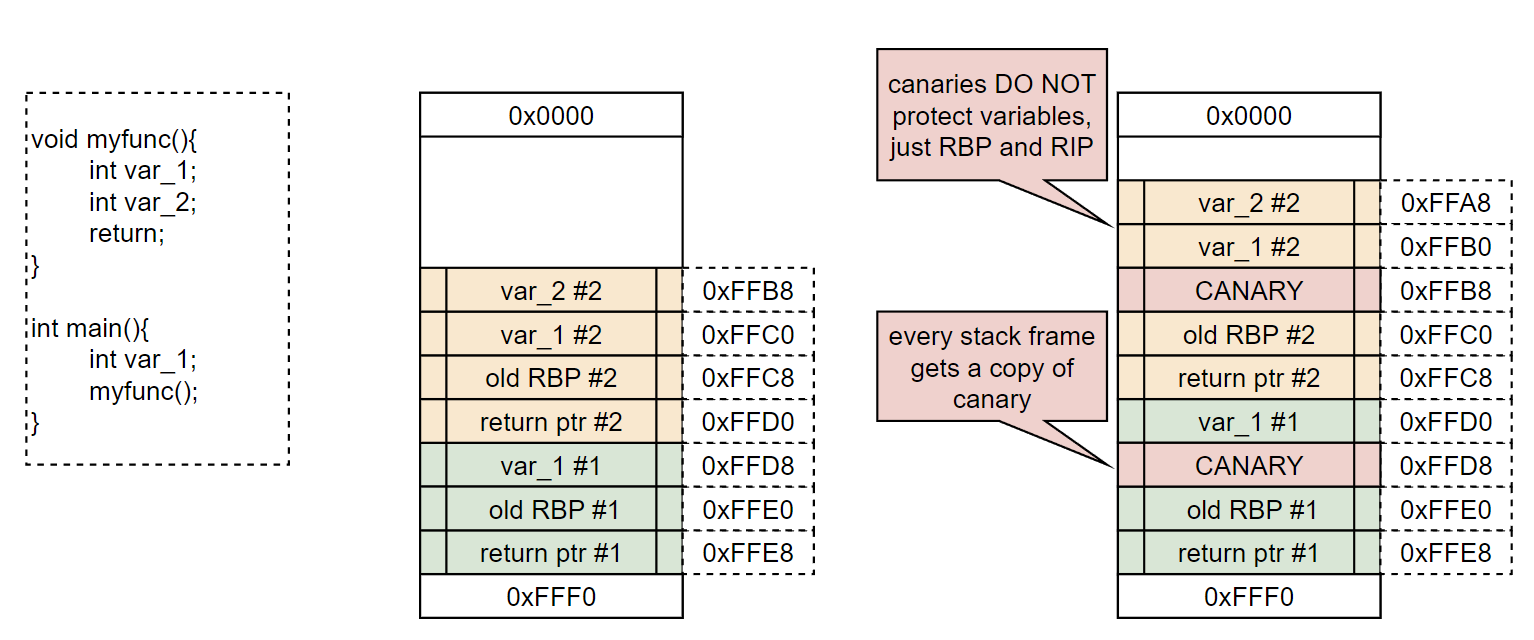
Lab 5

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:

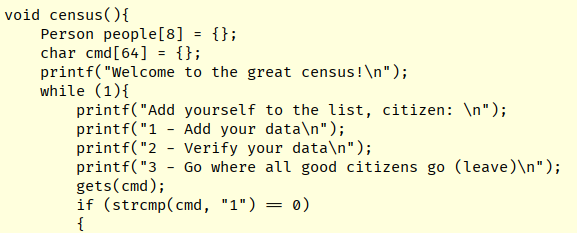
* Leak them
* Find a precise overwrite that will not corrupt canary

Lab

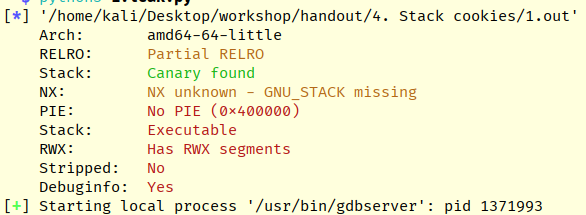
This lab will walk you through leaking and precise overwriting of stack cookies. We will showcase how data can be leaked and introduce few new ideas regarding C strings. You will then be tasked with finishing abuse of the binary.

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.

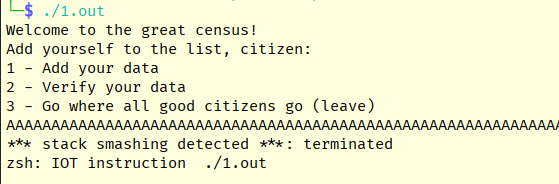
As we can see, this binary again seems to have a buffer overflow vulnerability:



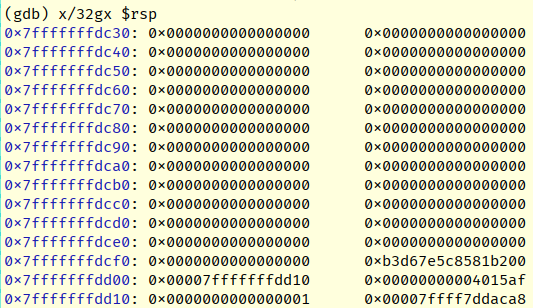
However, this time canary cookies are present:



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



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:



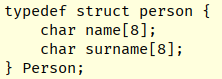
This is the stack cookie, notice the first byte being set to zero, effectively making it 7-byte long.

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)

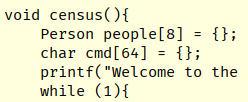
This means, it will try to access the address where array starts PLUS x times the size of one element. There is no built-in bound checking in C, if you want to read past array, you will.

In our binary, a following custom type is defined:

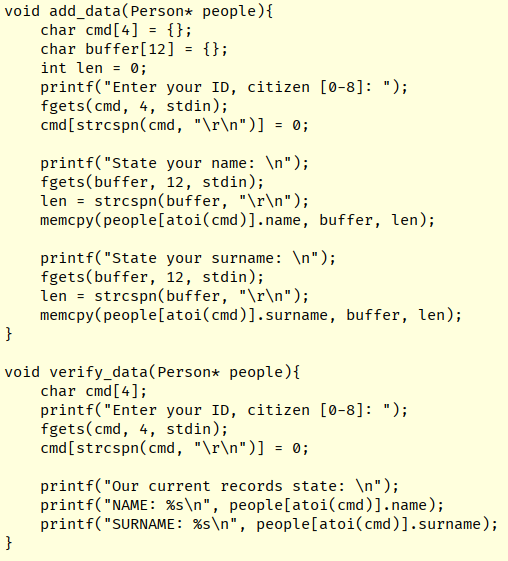


Esentially this means that our Person takes exactly 16 bytes of space, with name and surname each being 8 bytes long.

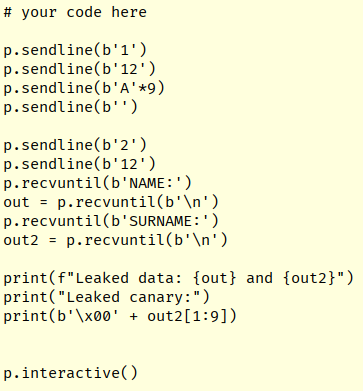
Further, an array of people 8 is declared.



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.



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:



Tasks

1. Read and try to understand the binary's source
2. Read the stack canary leak PoC that's in '1.leak.py'
   1. 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:
   1. with precise overwrite of RIP
   2. with plain gets() overflow and leaked cookie being aligned correctly
   3. pick one (or both if time permits) and do your best to finish the exploit