# BASH - Basic Asynchronous Shell

Author:
Matteo Bordignon, 13/05/2024

## Background

Memory corruption 1 occurs in a computer program when the contents of a memory location are modified due to programmatic behavior that exceeds the intention of the original programmer or program/language constructs.
A buffer overflow 2 condition exists when a program attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer. Writing outside the bounds of a block of allocated memory can corrupt data, crash the program, or cause the execution of malicious code.
Programming languages commonly associated with buffer overflows include C and C++, which provide no built-in protection against accessing or overwriting data in any part of memory and do not automatically check that data written to an array is within the boundaries of that array 3.
Mitigation against this attacks include:

- **Stack canary** 4
- **Position Independent Executables(PIE) and Address Space Layout Randomization (ASLR)** 5 - 6
- **Non-eXecutable (NX) stack** 7

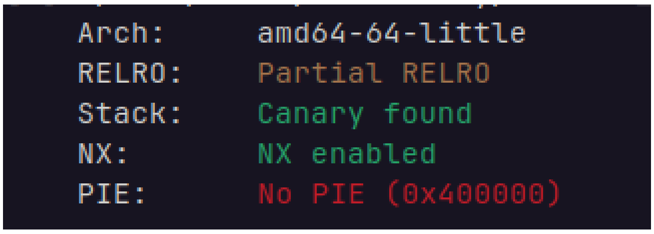
## Vulnerability

The program has a memory corruption vulnerability, more precisely it is vulnerable to buffer overflow (stack overflow). It is possible to bypass the canary protection and then overwrite the return address to call the `win` function.

## Solution

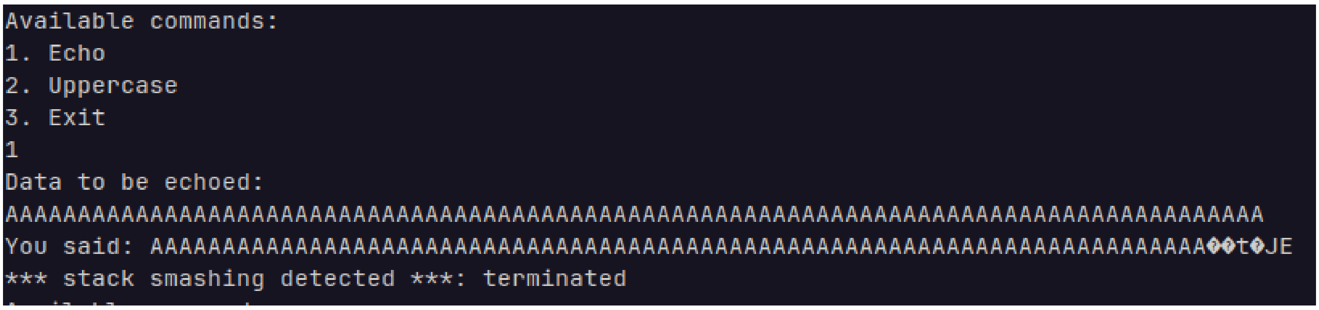
This challenge is a ret2win CTF. The goal is to make the program execute the `win` function which returns us the flag, overwriting the return address with the address of the `win` function.

Firstly I used the command `checksec` to check which countermeasures are applied on the provided binary file.



```
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

The canary and NX are enabled but PIE is disabled. This means that the program uses absolute addresses.
I executed the program to observe its behavior:



```
Available commands:
1. Echo
2. Uppercase
3. Exit
1
Data to be echoed:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
You said: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA��t�JE
*** stack smashing detected ***: terminated
```

Using the first option (echo) and providing a sufficient high number of characters I noticed that the program echoed strange bytes at the end of the string and then prints *stack smashing detected*. This means that probably the provided string as exceeded the buffer and has corrupted the canary.

Observing the source code I saw a potential bug:

*echo()*

```c
void echo() {
    char input[64];

    puts("Data to be echoed: ");
    read(0, input, 73);
    printf("You said: %.79s\n", input);
}
```

the `read` function reads the first 73 bytes (chars) on the standard input and puts them in the input variable which allocates space for just 64 chars. Then the `printf` prints the first 79 chars.

To bypass canary protection it is needed the canary offset, to find it, I followed the procedure showed in the last laboratory. The canary offset is 72 bytes and it is the same for the whole program since the input buffer are all the same size.

Using these information it is possible to write a python script [8] which solves the challenge. As seen before the `printf` function prints just 79 bytes which are not enough to leak the whole canary since the offset is 72 bytes and the canary is 8 bytes long, the last byte would not be leaked. So the last byte must be brute-forced using the fork oracle [4] technique given that the program fork itself when the user select either to echo or uppercase something. These are the steps that the python script performs:

- leak the first 7 bytes of the canary sending a payload composed of 72 bytes + `\n` to the `echo` function, the new line char overwrites the first byte of the canary which is the null byte ( `\x00` end of string), this allows the canary leakage.
- Now the script loops through all the possible 256 byte values and sends a payload composed of 72 bytes + candidate canary + 8 bytes (to overwrite the saved base pointer) + the address of the `win` function (found using the command: `readelf -s bin | grep win`), to the `toUpper` function. In this way when the right canary is provided, the payload will overwrite the return pointer of `toUpper` function with the address of `win` .
- Now the script has found the last byte of the canary but the program has not returned the flag since to obtain it the `win` function needs to be called from the parent process. To do that the scripts send the same payload as before using the canary just found to the main function and then press 3 to exit. Since the return address of the main function has been overwritten, the `win` function is called and it prints the flag.

```
Canary found: 0xde84aefc38d14f00
Congratulations! Here is the flag: UniTN{brute_forcing_canary_is_possible_but_not_easy}
```

# References

1. Memory curruption wikipedia
2. Buffer Overflow OWASP
3. Buffer Overflow wikipedia
4. Stack canary
5. Position Indipendent Executable (PIE)
6. Address Space Layout Randomization
7. NX stack
8. Full python code