radare2 리버스 엔지니어링

movptr06
해킹 보안 스터디 공부 방
리눅스 익스플로잇 개발 과정 0차시

발표자 소개

이름: 안수현

닉네임: movptr

학교: 한세사이버보안고등학교

소속: SSR, NWWTHW

오류가 있으면 언제든 말씀해 주세요!

발표 순서

1. 리눅스 익스플로잇 개발 과정 소개
2. 리버스 엔지니어링이란?
3. 어셈블리어 기초(이론)
4. 어셈블리어 기초(실습)
5. radare2 소개
6. 간단한 크랙미 예제 풀이
7. 참고 자료

리눅스 익스플로잇 개발 과정 소개

- 64비트 리눅스에서의 포너블(시스템 해킹)을 다루는 과정입니다.
- 리버스 엔지니어링, 버퍼 오버플로, 보호 기법 우회 등을 배울 것입니다.
- C언어, 파이썬, 리눅스에 관한 기초 지식이 필요합니다.
- 칼리 리눅스 환경에서 진행됩니다.

리버스 엔지니어링이란?

- 사전적 정의

리버스 엔지니어링(영어: reverse engineering, RE) 또는 역공학(逆工學)은 장치 또는 시스템의 기술적인 원리를 그 구조분석을 통해 발견하는 과정이다. 이것은 종종 대상(기계 장치, 전자 부품, 소프트웨어 프로그램 등)을 조각내서 분석하는 것을 포함한다.

https://ko.wikipedia.org › wiki › 역공학

역공학 - 위키백과, 우리 모두의 백과사전

- 쉽게 설명해서 프로그램의 소스 코드 없이 프로그램의 작동 원리를 알아내는 것을 의미한다.
- 프로그램을 분석해서 취약점을 찾아야 하는 포너블에서 중요하다.
- 방법으로는 프로그램을 실행시키지 않고 분석하는 정적 분석과 프로그램을 실행시키면서 분석하는 동적 분석이 있습니다.

어셈블리어 기초(이론)

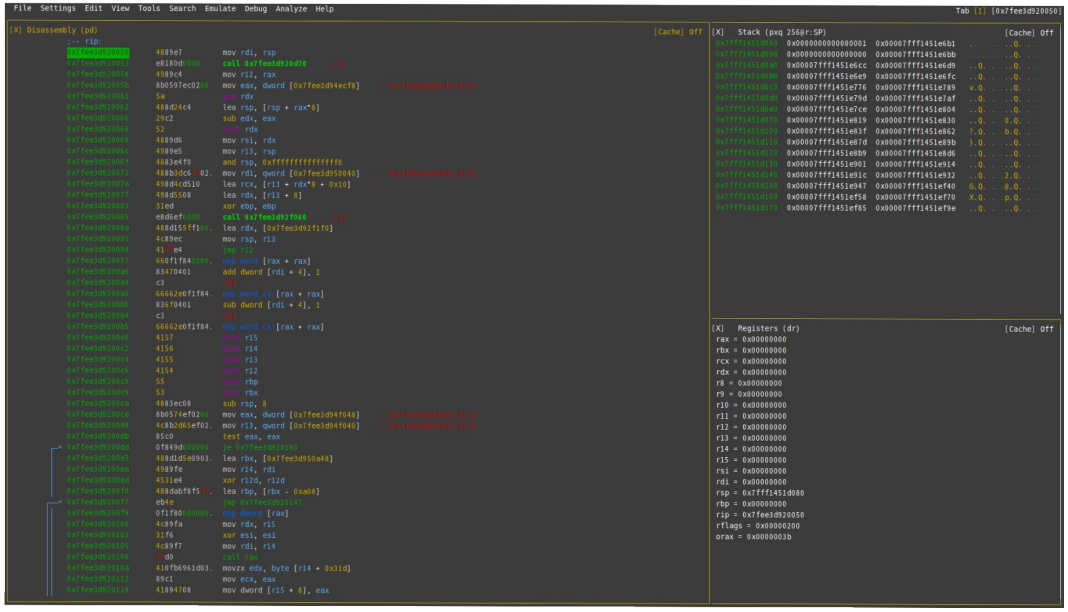
리버싱과 시스템 해킹을 공부하려면 꼭 알아야하는 어셈블리어의 기초[어셈블리어]

어셈블리어 기초(실습)

[실습편] 리버싱과 시스템 해킹을 공부하려면 꼭 알아야하는 어셈블리어의 기초 [어셈블리 언어]

radare2 소개

- radare2 는 리버스 엔지니어링을 위한 CLI 기반 프레임워크입니다.
- 칼리 리눅스에 기본적으로 설치되어 있습니다.

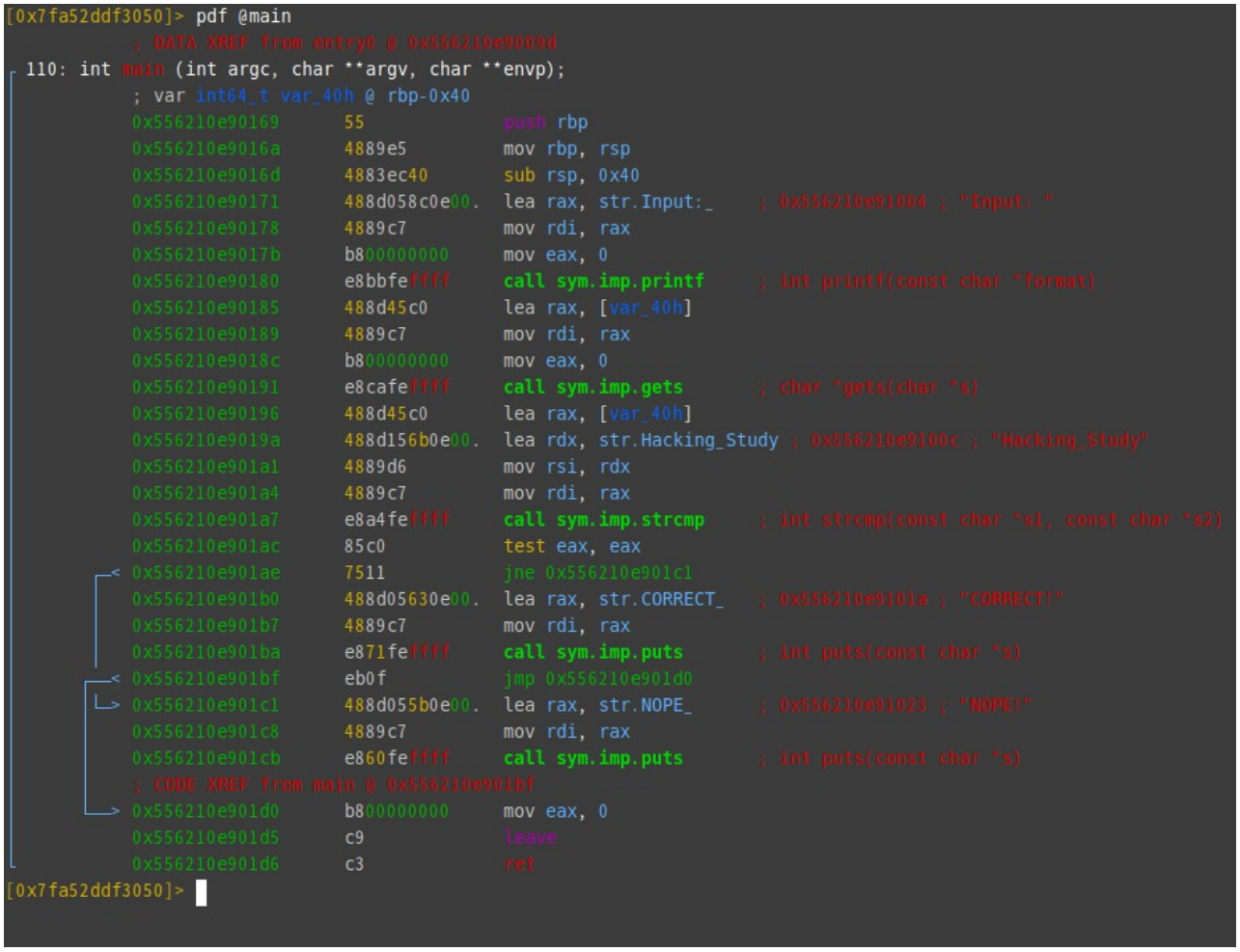


간단한 크랙미 예제 풀이

- r2 -d <파일 이름> 으로 파일을 디버깅 모드로 열 수 있습니다.
- 여기서는 crackme 파일을 열어 줍니다.
- aa 명령어로 분석을 실행합니다.
- afl 명령어로 함수 목록을 확인할 수 있습니다.
- 쉘 명령어도 실행시킬 수 있습니다.

```
┌──(vagrant㉿kali)-[/vagrant/Study/00]
└─$ r2 -d crackme
[0x7fa52ddf3050]> aa
[x] Analyze all flags starting with sym. and entry0 (aa)
[0x7fa52ddf3050]> afl
0x556210e90080    1 43           entry0
0x556210e92fe0    1 4124         reloc.__libc_start_main
0x556210e900b0    4 41   -> 34   sym.deregister_tm_clones
0x556210e900e0    4 57   -> 51   sym.register_tm_clones
0x556210e90120    5 57   -> 54   sym.__do_global_dtors_aux
0x556210e90070    1 6            sym.imp.__cxa_finalize
0x556210e90160    1 9            entry.init0
0x556210e90000    3 23           sym._init
0x556210e90240    1 1            sym.__libc_csu_fini
0x556210e90244    1 9            sym._fini
0x556210e901e0    4 93           sym.__libc_csu_init
0x556210e90169    4 110          main
0x556210e90030    1 6            sym.imp.puts
0x556210e90040    1 6            sym.imp.printf
0x556210e8f000    8 376  -> 455  loc.imp._ITM_deregisterTMCloneTable
0x556210e90050    1 6            sym.imp.strcmp
0x556210e90060    1 6            sym.imp.gets
[0x7fa52ddf3050]>
```

정적 분석 방법



- pdf @<함수 이름> 명령어로 함수를 디스어셈블할 수 있습니다.
- pd @<주소> 명령어로 주소에 있는 코드를 디스어셈블할 수 있습니다.
- 여기까지만 봐도 “Hacking_Study” 라는 정답을 찾을 수 있습니다.
- 정적 분석 할 때에는 -d 옵션이 없어도 됩니다.

발표할 때 간단하게 설명하고 넘어가서 추가한 내용입니다.

어셈블리어 코드 해석

```
[0x7fa52ddf3050]> pdf @main
            ; DATA XREF from entry0 @ 0x5562
┌ 110: int main (int argc, char **argv, char
            ; var int64_t var_40h @ rbp-0x40
            0x556210e90169      55
            0x556210e9019a      488d156b0e00.
            0x556210e901d0      b800000000
            0x556210e901d5      c9
            0x556210e901d6      c3
[0x7fa52ddf3050]>
```

```
mov rbp, rsp
sub rsp, 0x40
lea rax, str.Input:_     ; 0x556210e91004 ; "Input: "
mov rdi, rax
mov eax, 0
call sym.imp.printf      ; int printf(const char *format)
lea rax, [var_40h]
mov rdi, rax
mov eax, 0
call sym.imp.gets        ; char *gets(char *s)
lea rax, [var_40h]
lea rdx, str.Hacking_Study ; 0x556210e9100c ; "Hacking_Study"
mov rsi, rdx
mov rdi, rax
call sym.imp.strcmp      ; int strcmp(const char *s1, const
test eax, eax
jne 0x556210e901c1
lea rax, str.CORRECT_    ; 0x556210e9101a ; "CORRECT!"
mov rdi, rax
call sym.imp.puts        ; int puts(const char *s)
jmp 0x556210e901d0
lea rax, str.NOPE_       ; 0x556210e91023 ; "NOPE!"
mov rdi, rax
call sym.imp.puts        ; int puts(const char *s)
e901bf
mov eax, 0
leave
ret
```

함수 프롤로그로 함수 반환 이후 기존에 rbp 를 복원하기 위해 스택에 저장해 두고 있습니다. (sfp 값 저장)
또한 지역 변수를 저장하기 위해 rsp 를 감소시키고 있습니다.
(메모리상에서 스택은 깎아 나가는 형태로 구현됩니다.)

```
printf("Input: ");
gets(buf);
```

함수를 호출할 때에는 64비트 리눅스의 호출 규칙에 따라
rdi, rsi, rdx, rcx, r8, r9 레지스터에 순서대로
왼쪽의 인자부터 입력합니다.
남은 인자는 오른쪽에서부터 스택에 push 합니다.

```
if(!strcmp(buf, "Hacking_Study")) {
        puts("CORRECT!");
} else {
        puts("NOPE!");
}
```

strcmp 함수 호출의 결과를 이용한 조건문입니다.
test(and 연산을 실행만 하고 저장하지 않습니다.) 명령어로
rax(반환값이 저장되는 레지스터입니다.) 값이 0 인지
판단하여 ZF 플래그를 정합니다.
(0인 경우 ZF 플래그는 1입니다.)
jne(ZF 플래그가 0 이면, 점프합니다.)

함수 에필로그로 leave(mov rsp, rbp; pop rbp 와 동일) 명령어로 sfp 값을 읽어 rbp 를 호출 이전으로 복원합니다.
그리고 ret 명령어로 스택에서 ret 값을 읽어 호출 이전의 주소로 점프합니다.

동적 분석 방법

```
[0x7fa52ddf3050]> db 0x556210e901a7
[0x7fa52ddf3050]> dc
Input: hello
hit breakpoint at: 0x556210e901a7
[0x556210e901a7]> dr
rax = 0x7fff7932a7c0
rbx = 0x556210e901e0
rcx = 0x7fa52ddd19a0
rdx = 0x556210e9100c
r8 = 0x7fff7932a7c0
r9 = 0x00000000
r10 = 0x0000005d
r11 = 0x00000246
r12 = 0x556210e90080
r13 = 0x00000000
r14 = 0x00000000
r15 = 0x00000000
rsi = 0x556210e9100c
rdi = 0x7fff7932a7c0
rsp = 0x7fff7932a7c0
rbp = 0x7fff7932a800
rip = 0x556210e901a7
rflags = 0x00000202
orax = 0xffffffffffffffff
[0x556210e901a7]> pxr @0x7fff7932a7c0
0x7fff7932a7c0 0x0000006f6c6c6568   hello... @ rsp 478560413032 ascii ('h')
0x7fff7932a7c8 0x0000556210e90225   %...bU.. /vagrant/Study/00/crackme .text sym.__libc_csu_init program R X 'add rbx, 1' 'crackme'
0x7fff7932a7d0 ..[ null bytes ]..   00000000
0x7fff7932a7d8 0x0000556210e901e0   ....bU.. /vagrant/Study/00/crackme .text __libc_csu_init,rbx sym.__libc_csu_init program R X 'push r15' 'crackme'
0x7fff7932a7e0 ..[ null bytes ]..   00000000
0x7fff7932a7e8 0x0000556210e90080   ....bU.. /vagrant/Study/00/crackme .text entry0,section..text,_start,r12 entry0 program R X 'xor ebp, ebp' 'crackme'
0x7fff7932a7f0 0x00007fff7932a8f0   ..2y.... [stack] stack R W 0x1
0x7fff7932a7f8 ..[ null bytes ]..   00000000
0x7fff7932a808 0x00007fa52dc2a7fd   ...-....
0x7fff7932a810 0x00007fff7932a8f8   ..2y.... [stack] stack R W 0x7fff7932b6b1
0x7fff7932a818 0x0000000017933e000  ..3y.... 6328410112
0x7fff7932a820 0x0000556210e90169   i...bU.. /vagrant/Study/00/crackme .text main,main main program R X 'push rbp' 'crackme'
0x7fff7932a828 0x00007fff7932ab69   i.2y.... [stack] stack R W 0x39f0b9ddc593e44e
0x7fff7932a830 0x0000556210e901e0   ....bU.. /vagrant/Study/00/crackme .text __libc_csu_init,rbx sym.__libc_csu_init program R X 'push r15' 'crackme'
0x7fff7932a838 0x8cd5ba5926a4bccd   ...&Y...
0x7fff7932a840 0x0000556210e90080   ....bU.. /vagrant/Study/00/crackme .text entry0,section..text,_start,r12 entry0 program R X 'xor ebp, ebp' 'crackme'
0x7fff7932a848 ..[ null bytes ]..   00000000
0x7fff7932a860 0x732b483c7684bccd   ...v<H+s
0x7fff7932a868 0x739fe1dc69c8bccd   ...i...s
0x7fff7932a870 ..[ null bytes ]..   00000000
0x7fff7932a888 0x0000000000000001   ........ 1
0x7fff7932a890 0x00007fff7932a8f8   ..2y.... [stack] stack R W 0x7fff7932b6b1
0x7fff7932a898 0x00007fff7932a908   ..2y.... [stack] stack R W 0x7fff7932b6bb
0x7fff7932a8a0 0x00007fa52de24220    B.-.... /usr/lib/x86_64-linux-gnu/ld-2.33.so library R W 0x556210e8f000
0x7fff7932a8a8 ..[ null bytes ]..   00000000
0x7fff7932a8b8 0x0000556210e90080   ....bU.. /vagrant/Study/00/crackme .text entry0,section..text,_start,r12 entry0 program R X 'xor ebp, ebp' 'crackme'
[0x556210e901a7]> ps @0x556210e9100c
Hacking_Study
[0x556210e901a7]>
```

- db <주소> 명령어로 중단점을 설정할 수 있습니다.
- strcmp 함수를 호출하는 위치에 중단점을 설정하고 실행하였습니다.
- 주소는 실행할 때마다 달라질 수 있습니다.
- dr 명령어로 레지스터들의 값을 확인할 수 있습니다.
- pxr @<주소> 명령어로 특정 주소의 메모리 값들 확인할 수 있습니다.
- ps @<주소> 명령어로 특정 주소의 문자열을 확인할 수 있습니다.
- dc 명령어로 프로그램을 실행하거나 이어서 실행할 수 있습니다.

참고하면 좋은 자료

- 매번 까먹는 Radare2 사용법 정리
- x64 Assembly 시작하기
- radare2 익스플로잇 작성
- gdb 로 디버깅할때 오류나면 참고하세요.
  (발표 이후에 추가되었습니다.)

감사합니다