

기 술 문 서

Manual Binary Mangling with radare

정지훈
binoopang@is119.jnu.ac.kr



Abstract

Phrack 매거진에 소개된 Manual Binary Mangling with radare의 번역에 대한 최종 문서입니다. 본 문서의 내용은 phrack에 소개된 문서의 직역이 아닙니다. 여기서는 phrack의 내용에 역자의 radare 경험을 합쳐서 실제로 바이너리 조작이나 radare를 사용하여 분석, 그라픽 내용을 기술 한 것입니다.

바이너리 조작에서는 radare를 사용하여 바이너리를 먼저 분석하고, radare의 disassemble 기능과 쓰기 기능 등을 사용하여 파일을 변조하는 것입니다. 프랙 문서에서는 단순히 조작에 집중하지 않고 배경지식에 좀 더 비중을 두었습니다. 바이너리 조작의 내용은 이미 컴파일된 프로그램에 기존에 없던 새로운 코드를 삽입하는 것과, 크래킹에 대비하여 리버서에게 잘못된 정보를 주는 난 독화 부분이 있습니다. 새로운 코드를 삽입하는 문제에 있어서 코드가 어디에 삽입되어야 프로그램 충돌을 일으키지 않고 추가된 코드가 실행될 수 있는지에 대해서 먼저 알아본 후 프록시 후킹 까지 알아보며, 난 독화 부분에 있어서는 ELF입환식 파일에서 헤더를 조작하는 방법과, 라이브러리나 시스템 콜등을 난 독화 하는 내용을 다룹니다.

사실 위에서 다루는 내용들은 프로그램을 보호하기 위한 기술과 프로그램에 악성코드를 심을 때 주로 사용되는 방법일 것입니다. 하지만 문서에서는 프로그램을 공격하는 것 보다는 프로그램을 공격자로부터 어떻게 해야 보호할 수 있을지에 대해서 좀 더 조명합니다.

바이너리 조작 외에 radare를 사용한 역 참조 정보와 그래프에 대해서 기술 하였는데, 역 참조는 코드나 데이터가 어디서 사용되었는지에 대한 정보이며 이러한 정보는 바이너리를 분석하는데 매우 중요하고 유용한 정보들입니다. radare에는 이런 정보를 자동으로 구해 주며, 그래프를 지원하여 분석자가 쉽게 이해할 수 있도록 돕고 있습니다.

본 문서를 무리 없이 읽기 위해서는 radare에 대한 기초지식과 ELF 형식 파일에 대한 이해가 어느 정도 필요합니다(phrack 문서가 radare의 매뉴얼과 같은 특성을 지니지만 주요 내용은 바이너리 조작이기 때문에 radare에 대한 기본사항 까지는 다루지 않습니다.). 특히 radare의 특이한 문법은 한 번도 사용해 보지 않은 사람이 이해하기에는 무리가 있기 때문에, radare를 사용해 볼 것을 권장해 드립니다.

Content

1. radare 소개	1
1.1. 리눅스 디버거의 진화	1
1.2. radare 구성	3
1.3. 기본적인 사용 방법	8
 2. 바이너리 조작	14
2.1. 바이너리 조작의 의미	14
2.2. 바이너리 조작 환경	14
2.4. 배경 지식	15
2.5. 코드인젝션의 기초	21
2.6. 프로그램 보호를 위한 조작	27
2.7. 라이브러리 의존성 제거	33
2.8. 시스템 콜 난독 화	34
 4. 역 참조(xrefs) 정보	35
4.1. 역 참조 찾기	35
4.2. 그래프 역 참조 정보	37
 4. 결론	41
 5. 참고 문헌	42

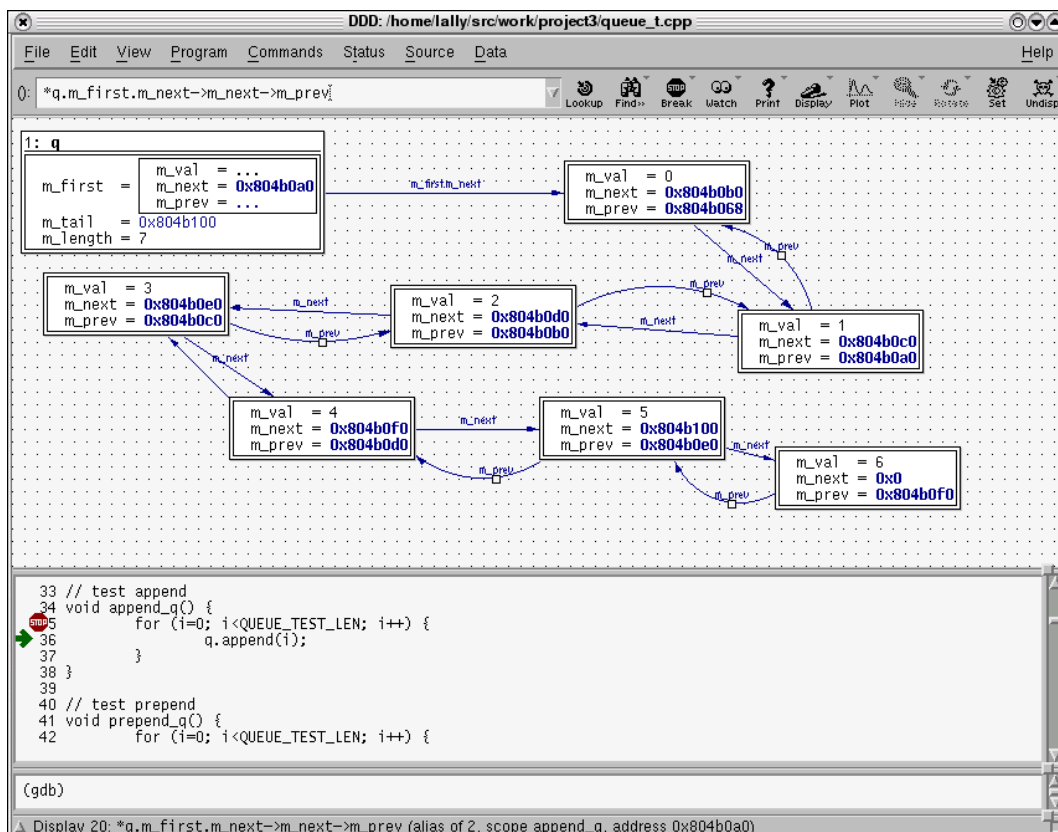
1. radare 소개

1.1. 리눅스 디버거의 진화

리눅스 디버거하면 보통 GDB를 떠올리지만 생각보다 많은 디버거가 존재합니다. 대부분의 디버깅은 GDB로 가능하지만 GDB가 콘솔기반의 디버거이기 때문에, 좀 더 편리하게 사용하기 위해서, 혹은 특정 부분에 특화된 디버거가 나타났습니다.

1.1.1. GDB와 D.D.D

리눅스에는 GDB라는 매우 훌륭한 디버거가 존재합니다. GDB는 콘솔기반의 디버거로서 매우 유연하고, 강력한 기능을 제공합니다. GDB는 백엔드(back end)로서의 역할을 수행할 수 있기 때문에 꽤 오래전 사용자가 편하게 사용할 수 있도록 그래픽 유저 인터페이스로 제작된 D.D.D라는 툴이 제공 되어졌었습니다.

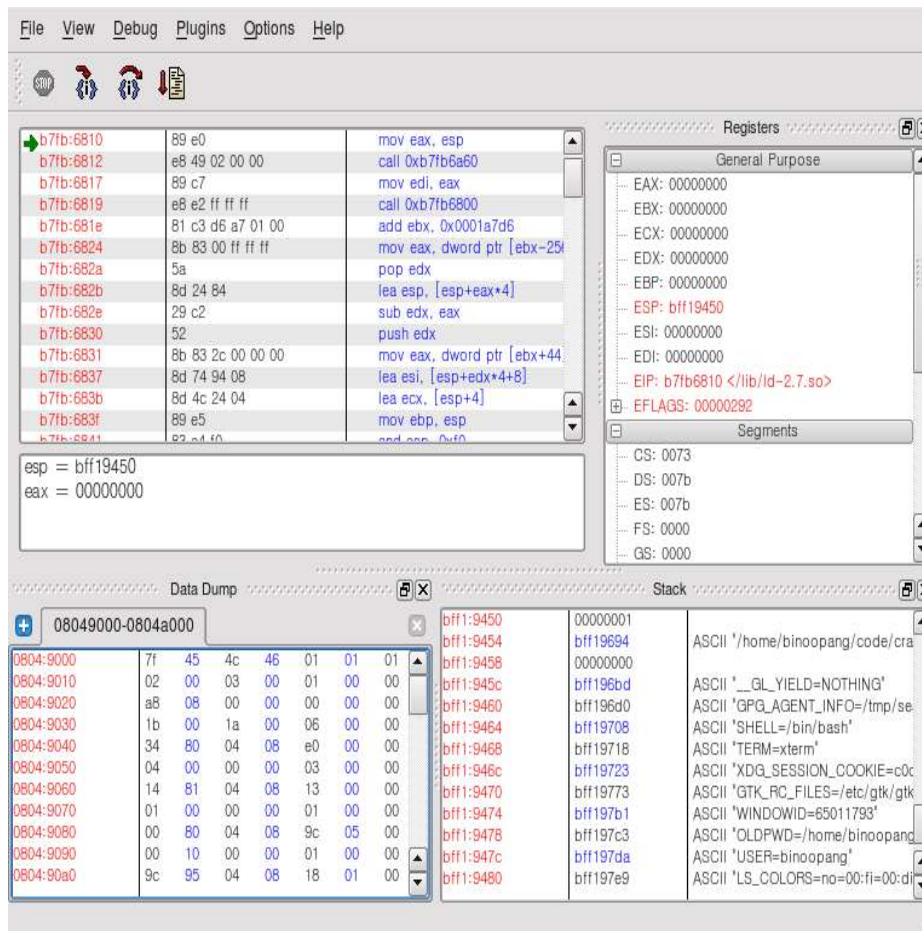


[그림 1] DDD

DDD는 GDB의 프론트엔드입니다. 따라서 실제 디버깅은 GDB에서 이루어지고, DDD는 이를 해석해서 사용자가 보기 쉽게 출력하여 줍니다. 즉 사용성을 편리하게 하기 위해서 제공되었지만, 큰 호응을 얻지 못하였습니다.

1.1.2. EDB

EDB는 Evan's Debugger입니다. Evan은 개발자 이름이고, Evan은 디스어셈블러와 디버거를 모두 만들어냈습니다. 이 디버거는 윈도우 디버거인 올리디버거 콘셉트로 만들어진 툴입니다.



[그림 2] Evan's Debugger

윈도우의 올리디버거와 사용하는 방법에 있어서 동일하고 도움 될 만한 플러그인을 여러 개 제공하기 때문에 매우 디버깅이 편리해 집니다.

1.1.3. radare

radare는 처음 HEX editor로 출발한 프로그램입니다. 지금도 HEX editor로서의 기능을 충실하게 가지고 있지만, 기본적인 에디터 기능보다는 프로그램 분석을 위한 도구로서 사용이 됩니다. 디버거, 디스어셈블러, 어셈블러와 같은 기능을 가지고 있고, 여기에 분석에 도움이 될 만한 작은 유틸리티들을 제공하고 있습니다.

1.2. radare 구성

1.2.1. 구성파일 목록

radare는 다음과 같은 실행 가능한 파일로 구성되어 있습니다.

```
[비누/usr/local/radare/bin]$ ls -al
합계 1836
drwxr-xr-x 2 root root 4096 2009-08-05 04:01 .
drwxr-xr-x 7 root root 4096 2009-08-05 03:07 ..
-rwxr-xr-x 1 root root 239386 2009-08-05 04:01 rabin
-rwxr-xr-x 1 root root 1322350 2009-08-05 04:01 radare
-rwxr-xr-x 1 root root 43824 2009-08-05 04:01 radiff
-rwxr-xr-x 1 root root 51272 2009-08-05 04:01 rahash
-rwxr-xr-x 1 root root 28312 2009-08-05 04:01 rasc
-rwxr-xr-x 1 root root 107433 2009-08-05 04:01 rasm
-rwxr-xr-x 1 root root 14097 2009-08-05 04:01 rax
-rwxr-xr-x 1 root root 960 2009-08-05 04:01 rfile
-rwxr-xr-x 1 root root 9994 2009-08-05 04:01 rsc
-rwxr-xr-x 1 root root 18740 2009-08-05 04:01 xrefs
[비누/usr/local/radare/bin]$
```

[그림 3] radare를 구성하는 파일

radare는 위의 파일들로 구성되어 있고, 각 파일은 특정 역할에 특화되어 있습니다. 위 파일 중 radare가 가장 중요한 메인 프로그램이고, 나머지 파일은 radare에서 보조로 사용하거나 직접 실행해서 특정 기능을 사용할 수 있습니다.

다음은 주요 유틸리티에 대한 소개입니다.

1.2.2. rabin

rabin은 바이너리파일 정보를 추출하는 도구입니다.

```
[비누/usr/local/radare/bin]$ rabin -l ./rabin
[Information]
class=ELF32
encoding=2's complement, little endian
os=linux
machine=Intel 80386
arch=intel
type=EXEC (Executable file)
stripped=No
static=No
baddr=0x08048000
[비누/usr/local/radare/bin]$
```

[그림 4] ELF 파일 정보

readelf와 비슷한 동작을 하는 도구입니다. 간략한 파일 정보 외에도 라이브러리 의존성이나 엔트리 포인트와 같이 디버깅에 도움이 되는 정보들을 보여줍니다.

1.2.3. radiff

radiff는 리눅스의 diff와 비슷한 프로그램이지만 diff보다 다양한 알고리즘이 적용되어져 있는 도구입니다. 기본적인 diff로서의 기능을 수행할 수도 있습니다.

```
[비누~/code/radare/patch]$ cat a.txt b.txt
binoopang
binoopane
[비누~/code/radare/patch]$ radiff -d ./a.txt ./b.txt
0x00000008 67 | 65 0x00000008
[비누~/code/radare/patch]$
```

[그림 5] radiff를 사용한 diffing

위와 같이 radiff를 사용해서 파일의 차이점을 파악할 수 있습니다.

1.2.4. rahash

rahash는 hash 알고리즘을 사용할 수 있게 해 주는 도구입니다. 특정 문자열을 hash하거나 파일을 hash하는 게 가능합니다.

```
[비누~/code/radare/patch]$ #특정 문자열 hash
[비누~/code/radare/patch]$ rahash -a md5 -s "binoopang"
89cb4f53f0e0a6f874c7d0f79f6665fa
[비누~/code/radare/patch]$
[비누~/code/radare/patch]$ #특정 파일 hash
[비누~/code/radare/patch]$ rahash -a md5 ./aUrootW?
409589fc9944ef28020c871489db4a6e
[비누~/code/radare/patch]$
```

[그림 6] rahash

첫 번째는 문자열을 hash한 것이고, 두 번째는 특정 파일을 hash 한 것입니다. 위의 경우 알고리즘으로 md5를 사용하였는데 이외에 다양한 알고리즘이 제공되고 한꺼번에 여러 알고리즘을 사용할 수도 있습니다.

1.2.5. rasc

rasc는 셸코드를 작성할 수 있도록 도와주는 도구입니다. 일종의 메타스플로잇과 비슷한 도구입니다. 여러 아키텍처와 기능을 제공합니다. 다음은 사용가능한 셸코드 목록입니다.

```

[비누~/code/radare/patch]$ rasc -L
arm.linux.binsh      47  Runs /bin/sh
arm.linux.suidsh     67  Setuid and runs /bin/sh
arm.linux.bind       203 Binds /bin/sh to a tcp port
armle.osx.reverse    151 iPhone reverse connect shell to HOST and PORT
dual.linux.binsh     99  x86/ppc MacOSX /bin/sh shellcode
dual.osx.binsh       121 Runs /bin/sh (works also on x86) (dual)
mips.linux.binsh     87  Runs /bin/sh (tested on loongson2f).
ppc.osx.adduser      219 Adds a root user named 'root' with no pass.
ppc.osx.binsh        72  Executes /bin/sh
ppc.osx.binsh0       72  Executes /bin/sh (with zeroes)
ppc.osx.bind4444     224 Binds a shell at port 4444
ppc.osx.reboot       28  Reboots the box
ppc.bsd.binsh        119 Runs /bin/sh
sparc.linux.binsh    216 Runs /bin/sh on sparc/linux
sparc.linux.bind4444 232 Binds a shell at TCP port 4444
sparc.linux.binsh2   36  Runs /bin/sh on sparc/linux (coder)
sparc.linux.bind1124 188 Listen shell at 1124
sparc.linux.connect  252 Connects to 10.12.34.3 : 1124
x64.linux.binsh      46  Runs /bin/sh on 64 bits
x86.bsd.binsh        46  Executes /bin/sh
x86.bsd.binsh2       23  Executes /bin/sh
x86.bsd.suidsh       31  Setuid(0) and runs /bin/sh
x86.bsd.bind4444     104 Binds a shell at port 4444
x86.bsdlinux.binsh   38  Dual linux/bsd shellcode runs /bin/sh
x86.freebsd.reboot   7   Reboots target box
x86.freebsd.reverse  126 Reboots target box
x86.linux.adduser     88  Adds user 'x' with password 'y'
x86.linux.bind4444    109 Binds a shell at TCP port 4444
x86.linux.binsh       24  Executes /bin/sh
x86.linux.binsh1     31  Executes /bin/sh
x86.linux.binsh2     36  Executes /bin/sh
x86.linux.binsh3     50  Executes /bin/sh or CMD
x86.linux.udp4444     125 Binds a shell at UDP port 4444
x86.netbsd.binsh      68  Executes /bin/sh
x86.openbsd.binsh    23  Executes /bin/sh
x86.openbsd.bind6969 147 Executes /bin/sh
x86.osx.binsh        45  Executes /bin/sh
x86.osx.binsh2       24  Executes /bin/sh
x86.osx.bind4444     112 Binds a shell at port 4444
x86.solaris.binsh    84  Runs /bin/sh
x86.solaris.binshu   84  Runs /bin/sh (toupper() safe)
x86.solaris.bind4444 120 Binds a shell at port 4444
x86.w32.msg          245 Shows a MessageBox
x86.w32.cmd          164 Runs cmd.exe and ExitThread
x86.w32.adduser      224 Adds user 'x' with password 'y'
x86.w32.bind4444     345 Binds a shell at port 4444
x86.w32.tcp4444      312 Binds a shell at port 4444
[비누~/code/radare/patch]$

```

[그림 7] 사용가능한 셸코드 목록

위와 같이 여러 가지 종류의 셸코드를 사용가능하고 실제 플랫폼에서 사용가능한지 테스트 해 볼 수도 있으며, 사용하기 쉽도록 여러 가지 출력 방법을 제공합니다.


```

[비누~/code/radare/patch]$ rasc -i x86.linux.binsh -c
unsigned char shellcode[] = {
    0x31, 0xc0, 0x50, 0x68, 0x2f, 0x2f, 0x73, 0x68, 0x68, 0x2f, 0x62, 0x69,
    0x6e, 0x89, 0xe3, 0x50, 0x53, 0x89, 0xe1, 0x99, 0xb0, 0x0b, 0xcd, 0x80,
};
[비누~/code/radare/patch]$ # 셸코드 테스트!
[비누~/code/radare/patch]$ rasc -i x86.linux.binsh -X
$ id
uid=1000(binoopang) gid=1000(binoopang) groups=4(adm),20(dialout),24(cdrom),46(plugdev),106(lpadmin),121(admin),122(sambashare),1000(binoopang)
$

```

[그림 8] rasc를 사용한 셸코드 생성 및 테스트

위와 같이 셸코드를 직접 생성하고 테스트할 수 있습니다. 소켓을 사용하는 셸코드도 생성할 수 있기 때문에 익스플로잇을 생성하는데 사용하면 도움이 될 것 같습니다.

1.2.6. rasm

rasm은 인라인 어셈블러 혹은 디어셈블러입니다. 셸코드를 생성한다거나, 디버깅 도중 opcode를 수정할 때 사용하면 좋습니다. 다음은 사용가능한 opcode의 목록입니다.

```

[비누/usr/local/radare/bin]$ rasm -l
Usage: rasm [-elvV] [-f file] [-s offset] [-a arch] [-d bytes] "opcode"|-
Architectures:
  olly, x86, ppc, arm, java
Opcodes:
  call [addr] - call to address
  jmp [addr] - jump to relative address
  jz [addr] - jump if equal
  jnz - jump if not equal
  trap - trap into the debugger
  nop - no operation
  push 33 - push a value or reg in stack
  pop eax - pop into a register
  int 0x80 - system call interrupt
  ret - return from subroutine
  ret0 - return 0 from subroutine
  hang - hang (infinite loop
  mov eax, 33 - assign a value to a register
Directives:
  .zero 23 - fill 23 zeroes
  .org 0x8000 - offset
[비누/usr/local/radare/bin]$

```

[그림 9] 사용 가능한 opcode 목록

모든 opcode를 제공하지 않고 일부만 제공합니다. 하지만 자주 사용되는 opcode를 지원함으로써 그다지 불편하지 않습니다.

```
[비누/usr/local/radare/bin]$ rasm -ev 'mov eax, 0x123'
b8 23 01 00 00
[비누/usr/local/radare/bin]$ █
```

[그림 10] rasm의 사용

rasm은 위와 같이 사용될 수 있습니다.

1.2.7. rax

rax는 진법 변환을 해 주는 프로그램입니다. 10진수를 16진수로 바꾸거나, 혹은 그 역변환 등 다양한 진수를 제공합니다.

```
[비누/usr/local/radare/bin]$ rax -h
Usage: rax [-] | [-s] [-e] [int|0x|Fx|.f|.o] [...]
int   -> hex      ; rax 10
hex   -> int      ; rax 0xa
-int  -> hex      ; rax -77
-hex  -> int      ; rax 0xfffffb3
float -> hex      ; rax 3.33f
hex   -> float    ; rax Fx40551ed8
oct   -> hex      ; rax 035
hex   -> oct      ; rax 0x12 (0 is a letter)
bin   -> hex      ; rax 1100011b
hex   -> bin      ; rax Bx63
-e    swap endianness ; rax -e 0x33
-s    swap hex to bin ; rax -s 43 4a 50
-     read data from stdin until eof
[비누/usr/local/radare/bin]$ █
```

[그림 11] rax의 사용

위와 같이 다양한 방법을 제공합니다. 아래는 사용의 예입니다.

```
[비누/usr/local/radare/bin]$ # 10진수 -> 16진수
[비누/usr/local/radare/bin]$ rax 123
0x7b
[비누/usr/local/radare/bin]$ # 16진수 -> 10진수
[비누/usr/local/radare/bin]$ rax 0x7b
123
[비누/usr/local/radare/bin]$ # 16진수 -> 8진수
[비누/usr/local/radare/bin]$ rax 0x12b
453o
[비누/usr/local/radare/bin]$ # 8진수 -> 16진수
[비누/usr/local/radare/bin]$ rax 453o
0x12b
[비누/usr/local/radare/bin]$ # 16진수 -> 2진수
[비누/usr/local/radare/bin]$ rax Bx123
100100011b
[비누/usr/local/radare/bin]$ # 2진수 -> 16진수
[비누/usr/local/radare/bin]$ rax 100100011b
0x123
```

[그림 12] rax 사용의 예

1.3. 기본적인 사용 방법

1.3.1. radare 설정

radare는 vi와 같이 실행하면서 환경설정 파일을 읽어 들입니다. 그 파일은 홈 디렉터리의 .radarerc라는 이름으로 존재합니다.

```
[비누~]$ # 파일 확인
[비누~]$ ls -l .radarerc
-rw-r--r-- 1 binoopang binoopang 148 2009-08-04 14:11 .radarerc
[비누~]$
[비누~]$ # 파일의 내용
[비누~]$ cat .radarerc
; Automatically generated by radare
e file.id=true
e file.flag=true
e file.analyze=true
e scr.color=true
e dbg.bep=main
e dir.project=~/.radare/rdb
[비누~]$
```

[그림 13] radarerc의 내용

내용을 보면 알겠지만 radare의 환경변수를 설정합니다. 만약 환경변수를 무시하고 radare를 실행하려면 -n 플래그와 함께 실행하면 됩니다. radare에는 매우 많은 변수들을 가지고 있기 때문에 매뉴얼을 확인해서 변수에 대해 조금 파악해 놓는 게 좋을 것입니다.

1.3.2. 기본 문법

주관적인 생각이지만 radare는 문법의 가독성이 좋지 않습니다. 모든 명령은 한 글자로 이루어지고 서브 명령은 문자 뒤에 댛 붙여지는 형식입니다.

```
[#][!][cmd] [arg] [@ offset:size|@@ flags|@@=off:sz ..] [> file] [| shell-pipe] [~grep#[]] [ && ...]
```

[표 1] radare 명령 형식

(표 1)은 명령어 형식을 잘 나타낸 것입니다. 가장 앞에 주석이 올 수 있고, 다음엔 느낌표가 오는데 이것은 뒤에 오는 명령을 어디로 보낼 것인지 결정합니다. 느낌표가 하나 사용되면 로드된 플러그인으로 전송되고, 두 개가 오면 플러그인을 지나쳐서 셸로 전송됩니다. 즉 셸 명령이 실행됩니다.

cmd는 실제 radare를 통해서 실행 할 명령이 옵니다. 명령에는 인자(arg)가 있을 수 있고 없을 수도 있습니다. 그리고 이어지는 @는 명령이 어떤 주소에서 실행될 것인지 표기하는 것이고 이것이 두 개가 사용되면 등록된 플래그와 일치하면 실행하는 것입니다.

다음은 radare로 어떤 프로그램을 열어서 디어셈블된 결과를 출력한 것입니다.

```
[0xB7F5C810]> pd 10
0xb7f5c810, eip: 89e0 mov eax, esp
0xb7f5c812, e829020000 call 0xb7f5ca40 ; 1 = 0xb7f5ca40
0xb7f5c817, 89c7 mov edi, eax
0xb7f5c819, e8e21fffff call 0xb7f5c800 ; 2 = 0xb7f5c800
0xb7f5c81e, 81c3d6c70100 add ebx, 0x1c7d6
0xb7f5c824, 8b8300ffff mov eax, [ebx-0x100]
0xb7f5c82a, 5a pop edx
0xb7f5c82b, 8d2484 lea esp, [esp+eax*4]
0xb7f5c82e, 29c2 sub edx, eax
0xb7f5c830, 52 push edx
[0xB7F5C810]>
```

[그림 14] pd 10

위 그림은 '/bin/lis'를 radare로 불러온 뒤 pd 10 명령을 사용한 것입니다. 'pd 10'에서 알 수 있듯이 'p'는 'print'의 약자이고 'd'는 'disassemble'의 약자입니다. 즉 디어셈블된 결과를 출력하라는 것인데 인자로 10을 준 것입니다. 결과를 10개 출력하는 것이고 여기서 10개는 opcode의 개수입니다.

radare에는 수많은 명령어가 존재합니다. 특정 명령어의 사용방법에 대해서는 명령어에 '?'를 붙이면 됩니다.

```
[0xB7F5C810]> p?
Available formats:
p% : print scrollbar of seek (null)
p= : print line bars for each byte (null)
pa : ascii (null)
pA : ascii printable (null)
pb : binary N bytes
pB : LSB Stego analysis N bytes
pc : C format N bytes
pd : disassembly N opcodes bsize bytes
pD : asm.arch disassembler bsize bytes
pe : double 8 bytes
pF : windows filetime 8 bytes
pf : float 4 bytes
pi : integer 4 bytes
pl : long 4 bytes
pL : long (ll for long long) 4/8 bytes
pm : print memory structure 0xHHHH
pC : comment information string
po : octal dump N bytes
p0 : Overview (zoom.type) entire file
```

[그림 15] p명령의 help

위의 그림은 p 명령의 종류 중 일부입니다. 인자가 있는 것이 있고 없는 것도 있습니다.

1.3.3. Warming up!!

radare를 사용하여 간단한 프로그램 패치를 해보는 예제를 수행 해 보겠습니다. 1장이 끝나면 2장부터는 실제 프랙(phrack)에서 소개된 내용을 다루게 될 텐데, 사실 2장은 기본적인 radare지식이나 ELF 형식에 대해서 잘 모른다면 이해하기 힘든 내용들입니다. 따라서 radare의 기본사항은 숙지하고 보셔야 이해가 잘 되실 것입니다. 이 문서에서 radare의 기본적인 내용을 모두 다루는 것은 범위를 넘어서게 됩니다.

1) Target Program!!

다음은 프로그램 소스입니다.

```
/* ----- */
/* Print password if user root
 * binoopang's crackme
 * ----- */
/* -- printf password -- */
void print_pass()
{
    int i;
    for(i=0x30 ; i<0x40 ; i++)
        printf("%c", i+15);
    printf("\n");
}

int main()
{
    // Show password if user root
    if(getuid()==0)
        print_pass();
    // or not!!
    else
        printf("bye~ :)#n");

    return 0;
}
```

[그림 16] 간단한 crackme

위 프로그램은 만약 root 유저라면 패스워드를 출력해 주는 프로그램입니다. 이 프로그램을 패치해서 root가 아니더라도 패스워드를 출력하도록 해 보겠습니다.

2) Disassemble with radare

컴파일한 프로그램을 radare를 사용하여 불러옵니다.

```
$ radare -w ./crackme
```

[표 2] radare 명령 형식

```

[Information]
ELF class:      ELF32
Data encoding:  2's complement, little endian
OS/ABI name:    linux
Machine name:   Intel 80386
Architecture:   intel
File type:      EXEC (Executable file)
Stripped:       No
Static:         No
Base address:   0x08048000
> Importing symbols...
11 fields added
5 imports added
32 symbols added
36 sections added
128 strings added
> Analyzing code...
  [/] 14:00:0a:03 =====
strings: 128
functions: 15
structs: 0
data_xrefs: 16
code_xrefs: 8
[0x08048370]>

```

[그림 17] crackme 로드

파일을 수정해야 하기 때문에 -w 플래그를 사용했습니다. 위처럼 간단한 파일 정보들이 출력되고 엔트리에서 대기하고 있습니다. pd 명령을 사용해서 현재 위치에서 디스어셈블 해 보겠습니다.

```

[0x08048370]> pd 10
; [13] 0x08048370 size=00000460 align=0x00000010 -r-x .text
; framesize = 4,0
; args = 0
; vars = 0
; drefs = 3
0x08048370, /entrypoint,sym._start,section._plt_end,section._t
ext,str.by:
0x08048370, | 31ed      xor ebp, ebp
0x08048372, | 5e        pop esi
0x08048373, | 89e1      mov ecx, esp
0x08048375, | 83e4f0    and esp, 0xf0
0x08048378, | 50        push eax
0x08048379, | 54        push esp
0x0804837a, | 52        push edx
0x0804837b, | 68a0840408 push dword 0x80484a0 ;
sym.__libc_csu_fini
0x08048380, | 68b0840408 push dword 0x80484b0 ;
sym.__libc_csu_init
0x08048385, | 51        push ecx

```

[그림 18] 현재 위치에서 pd 명령 사용

엔트리 포인트가 sym._start로 잡혀 있는 것을 확인할 수 있습니다. 이 함수는 초기화 작업과 함께 main() 함수를 호출하는 역할입니다. 따라서 여기에는 관심이 없습니다. main() 위치로 이동해야 합니다.

```
[0x08048370]> s sym.main
```

[표 3] 위치 변경

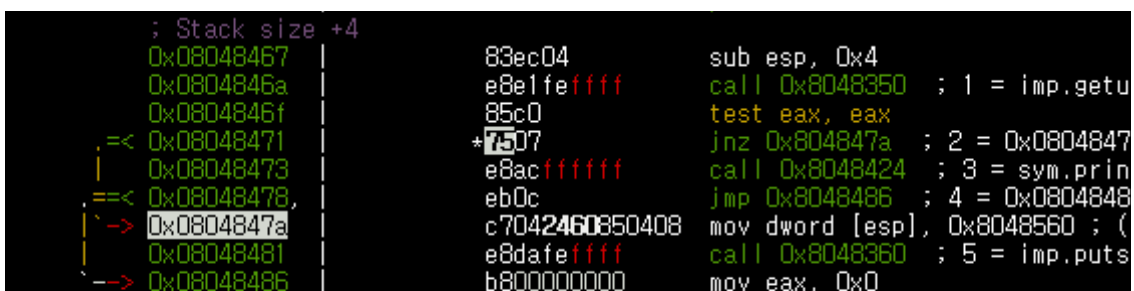
s 명령을 사용하여 위치를 main()으로 이동합니다.



```
0x08048459 / sym.main: 8d4c2404 lea ecx, [esp+0x4]
0x0804845d | 83e4f0 and esp, 0xf0
0x08048460, | 71fc push dword [ecx-0x4]
0x08048463 | 55 push ebp
0x08048464, | 89e5 mov ebp, esp
0x08048466 | 51 push ecx
; Stack size +4
0x08048467 | 83ec04 sub esp, 0x4
0x0804846a | e8e1fefff call 0x8048350 ; 1 = imp.getu
id
0x0804846f | 85c0 test eax, eax
a, =< 0x08048471 | 7507 jnz 0x0804847a ; 2 = 0x0804847
| 0x08048473 | e8acfffff call 0x8048424 ; 3 = sym.prin
t_pass
a, ==< 0x08048478, | eb0c jmp 0x08048486 ; 4 = 0x0804848
6
| -> 0x0804847a | c7042460850408 mov dword [esp], 0x8048560 ; (
0x08048560)
| 0x08048481 | e8dafeffff call 0x8048360 ; 5 = imp.put
- -> 0x08048486 | b800000000 mov eax, 0x0
[0x08048459]>
```

[그림 19] main() 함수

위 그림은 main()함수의 디스어셈블 결과입니다. radare의 Visual모드로 변환해서 편집해 보겠습니다. Visual 모드는 'V'를 입력하면 됩니다. vi와 똑같이 hjkl을 사용하여 화면 이동이 가능합니다.



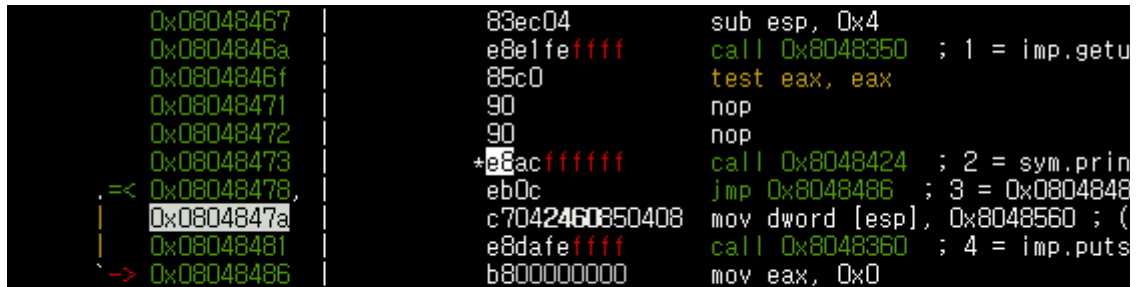
```
; Stack size +4
0x08048467 | 83ec04 sub esp, 0x4
0x0804846a | e8e1fefff call 0x8048350 ; 1 = imp.getu
0x0804846f | 85c0 test eax, eax
a, =< 0x08048471 | *7507 jnz 0x0804847a ; 2 = 0x0804847
| 0x08048473 | e8acfffff call 0x8048424 ; 3 = sym.prin
t_pass
a, ==< 0x08048478, | eb0c jmp 0x08048486 ; 4 = 0x0804848
6
| -> 0x0804847a | c7042460850408 mov dword [esp], 0x8048560 ; (
0x08048560)
| 0x08048481 | e8dafeffff call 0x8048360 ; 5 = imp.put
- -> 0x08048486 | b800000000 mov eax, 0x0
```

[그림 20] Visual의 커서 모드

(그림 20)은 Visual의 커서모드입니다. 'c'를 입력해서 커서모드 변환이 가능합니다. 현재 커서가 가리키고 있는 '7507'은 우리가 수정해야 할 대상입니다. jnz 명령인데 root가 아닐 경우 점프하는 부분입니다. 이 루틴을 nop 처리하면 됩니다.

3) Modify Opcode

vi와 똑같이 커서모드에서 'i'를 입력하면 입력모드로 변환됩니다. 여기서 '7507'을 '9090'으로 바꾸게 되면 'jnz' 명령이 nop 처리 됩니다.

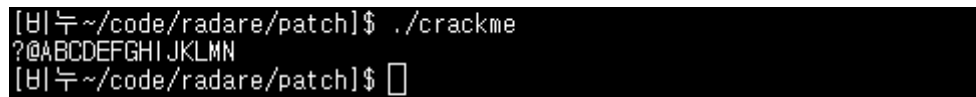


[그림 21] jnz -> nop

실시간으로 opcode를 해석해 주기 때문에 실수 없이 수정할 수 있습니다. nop 처리 된 후 파일을 저장 하고 종료합니다.

4) Run Program!

패치 된 프로그램을 실행 해 보겠습니다.



[그림 22] 패치 성공

성공적으로 패치 되었습니다. gdb나 edb와 같이 메모리 수정이 아니라¹⁾ 파일에서 수정하는 것이기 때문에 파일로 그대로 저장이 됩니다.

1) gdb나 edb도 위와 같이 opcode 수정이 가능하지만 메모리의 코드를 수정하는 것이기 때문에 파일 패치를 위해서 별도의 덤프 작업이 수행되어야 합니다.

2. 바이너리 조작

2.1. 바이너리 조작의 의미

바이너리를 조작한 다는 것은 여러 가지 의도가 들어갈 수 있습니다. 리버스 엔지니어링에서 가장 흔하게 접할 수 있는 것은 패킹입니다. 패커는 바이너리를 수정해서 주요 정보를 은닉하거나 파일의 크기를 줄여줍니다. 하지만 동작은 바이너리 수정 전과 동일하게 동작합니다.

패커와 같이 리버서로부터 바이너리를 보호하는 차원의 바이너리 조작이 있는가 하면 크래커가 바이너리 파일을 조작해서 보호 장치를 깨뜨리는 것도 바이너리 조작의 한 부분이라고 할 수 있습니다. 따라서 바이너리를 조작한다는 것은 정의로운 행동이 될 수도 있고 그렇지 않을 수도 있습니다.

이 장에서는 이와 같은 크래커들로부터 바이너리를 보호하기 위한 수단으로서의 조작을 다룹니다.

2.2. 바이너리 조작 환경

2.2.1. 바이너리 형식

이 문서에서 다루는 바이너리는 리눅스나 유닉스에서 넓게 사용되는 ELF 형식입니다. radare는 ELF 뿐만 아니라 Microsoft의 PE형식과 같이 다른 형식도 지원하고 있습니다. 모든 형식파일에 대해서 다룰 수는 없고, 가장 폭넓게 사용되는 형식을 생각했을 때 ELF 형식을 선택하게 되었습니다.

2.2.2. radare 혹은 radare2?

radare는 현재 버전 2가 개발되고 있습니다. radare2는 radare1에 비해서 구조적으로 많은 변화가 있고, 프로그램을 처음부터 완전히 다시 쓰는 것이기 때문에 아직은 불안정합니다. 그래서 radare2는 이 문서에서 다루지 않습니다. 대신 radare1을 사용해서 모든 과정을 설명합니다.

2.3. 바이너리 조작의 방향

2.3.1. Code Injection

코드 인젝션이란 바이너리 파일에 기존에 없던 코드를 추가로 넣는 것을 말합니다. 이것은 패커가 일반적으로 하는 행위이기도 합니다. 패커는 프로그램을 실행할 때 압축되어져 있는 코드영역이나 데이터영역을 다시 원래대로 해제하기 위해서 언 패킹 코드를 삽입해야 합니다. 이와 같이 기존에 없던 새로운 코드를 추가하는 것입니다.

이러한 작업을 위해서는 ELF 형식에 대해서 이해력이 필요합니다. ELF 형식에서 코드는 .text 섹션에 존재합니다. 이 영역에 보통 코드를 삽입하는데 일반적으로 코드를 아무렇게나 삽입하게 되면 자칫 프로그램이 망가지기 쉽습니다.

때문에 프로그램을 전체적으로 분석해야 하고 코드가 삽입될 만한 공간을 찾아내야 합니다. 이러한 작업을 위해서 바텀 업 방식의 분석을 사용하는데, 최소한의 영역을 분석해 가면서 함수의 의존성을 깨뜨리지 않고 프로그램을 수정하여 결국 프로그램이 망가지지 않고 정상적으로 실행될 수 있도록 유지합니다.

2.3.2. Protections and manipulations

바이너리를 보호하기 위해서 프로그램을 일부러 복잡하게 조작하거나, 헤더를 조작하여 잘못된 정보가 공격자에게 전달될 수 있도록 합니다. 이러한 정보의 수정도 결국 프로그램이 정상적으로 실행되어야 하기 때문에 먼저 자세한 분석이 선행되어야 합니다.

2.4. 배경지식

2.4.1. 실행되지 않는 코드

프로그램을 리버싱하다 보면 알게 되지만, 정상적으로 실행되어도 결국 실행되지 않는 구역을 발견하곤 합니다. 이러한 코드는 정적인 분석으로는 자세한 정보를 얻을 수 없기 때문에 동적인 분석으로 발견할 수 있습니다. 왜냐하면 보통 분기 문들이 레지스터를 기반으로 분기를 하는데 정적인 분석으로는 도저히 레지스터에 어떤 값이 들어올지 예측이 불가능하기 때문입니다.

또한 컴파일러의 stub 코드들이 있는데, 이러한 코드들 중 단순히 공간을 채우기 위한 역할로 존재하는 코드들이 존재합니다. 이러한 코드들은 절대로 실행되지 않으며, 만약 코드를 삽입해야 한다면 이러한 공간이 사용될 수 있습니다.

2.4.2. unrolled loops

unrolled loop가 뭔지 먼저 이해 할 필요가 있습니다. 이것은 loop를 펼치는 것을 말하는데 다음 코드를 보겠습니다.

```

int main()
{
    int i;
    char buf_1[40000];
    char buf_2[40000];

    memset(buf_2, 'b', 40000);

    for(i=0 ; i<40000 ; i++)
        buf_1[i] = buf_2[i];

    buf_1[39999]='\0';
    printf("buffer : %s\n", buf_1);

    return 0;
}

```

[그림 23] rolled loop

rolled loop의 예입니다. 40000개의 원소를 가지는 char 형 배열을 복사하는 루프입니다. 위 루프는 한 루프 당 한 개의 값을 복사하는 것입니다. 다음은 unrolled loop입니다.

```

int main()
{
    int i;
    char buf_1[40000];
    char buf_2[40000];

    memset(buf_2, 'b', 40000);

    for(i=0 ; i<40000 ; i+=4)
    {
        buf_1[i] = buf_2[i];
        buf_1[i+1] = buf_2[i+1];
        buf_1[i+2] = buf_2[i+2];
        buf_1[i+3] = buf_2[i+3];
    }

    buf_1[39999]='\0';
    printf("buffer : %s\n", buf_1);

    return 0;
}

```

[그림 24] unrolled loop

unrolled loop입니다. 이 루프도 rolled loop와 결과는 똑같지만 한 루프에 4개의 값을 복사합니다. 덕분에 코드의 길이는 늘어났지만 속도는 더 빠릅니다. 왜 그럴까요? 이것은 CPU가 레지스터 분기를 위한 계산의 횟수가 줄어들고, 파이프라인의 덕을 볼 수 있기 때문입니다. 위 코드를 어셈블리 수준에서 분석 해 보겠습니다.

```

--> 0x0804849d | 8b8574c7feff | mov eax, [ebp-0x1388c]
|| 0x080484a3 | 8b9574c7feff | mov edx, [ebp-0x1388c]
|| 0x080484a9 | 0fb6941578c7fe. | movzx edx, byte [ebp+edx-0x13888]
|| 0x080484b1 | 889405b863ffff | mov [ebp+eax-0x9c48], dl
|| 0x080484b8 | 838574c7feff01 | add dword [ebp-0x1388c], 0x1
|`-> 0x080484bf | 81bd74c7feff3f. | cmp dword [ebp-0x1388c], 0x9c3f
`==< 0x080484c9 | 7ed2 | jle 0x804849d ; 3 = 0x0804849d

```

[그림 25] rolled loop의 어셈블리 코드

rolled loop는 어셈블리 코드도 매우 간결합니다. 매 루프에서 수행하는 일이 적기 때문입니다. 문제라면 저 루프가 매우 자주 돌아간다는 것입니다. 다음은 unrolled loop의 어셈블리 코드입니다.

```

--> 0x080484a0 | 8b9574c7feff | mov edx, [ebp-0x1388c]
|| 0x080484a6 | 8b8574c7feff | mov eax, [ebp-0x1388c]
|| 0x080484ac | 0fb6840578c7fe. | movzx eax, byte [ebp+eax-0x13888]
|| 0x080484b4 | 888415b863ffff | mov [ebp+edx-0x9c48], al
|| 0x080484bb | 8b8574c7feff | mov eax, [ebp-0x1388c]
|| 0x080484c1 | 8d5001 | lea edx, [eax+0x1]
|| 0x080484c4 | 8b8574c7feff | mov eax, [ebp-0x1388c]
|| 0x080484ca | 83c001 | add eax, 0x1
|| 0x080484cd | 0fb6840578c7fe. | movzx eax, byte [ebp+eax-0x13888]
|| 0x080484d5 | 888415b863ffff | mov [ebp+edx-0x9c48], al
|| 0x080484dc | 8b8574c7feff | mov eax, [ebp-0x1388c]
|| 0x080484e2 | 8d5002 | lea edx, [eax+0x2]
|| 0x080484e5 | 8b8574c7feff | mov eax, [ebp-0x1388c]
|| 0x080484eb | 83c002 | add eax, 0x2
|| 0x080484ee | 0fb6840578c7fe. | movzx eax, byte [ebp+eax-0x13888]
|| 0x080484f6 | 888415b863ffff | mov [ebp+edx-0x9c48], al
|| 0x080484fd | 8b8574c7feff | mov eax, [ebp-0x1388c]
|| 0x08048503 | 8d5003 | lea edx, [eax+0x3]
|| 0x08048506 | 8b8574c7feff | mov eax, [ebp-0x1388c]
|| 0x0804850c | 83c003 | add eax, 0x3
|| 0x0804850f | 0fb6840578c7fe. | movzx eax, byte [ebp+eax-0x13888]
|| 0x08048517 | 888415b863ffff | mov [ebp+edx-0x9c48], al
|| 0x0804851e | 838574c7feff04 | add dword [ebp-0x1388c], 0x4
|`-> 0x08048525 | 81bd74c7feff3f. | cmp dword [ebp-0x1388c], 0x9c3f
`==< 0x0804852f | 0f8e6bffffff | jle dword 0x80484a0 ; 3 = 0x080484a0

```

[그림 26] unrolled loop의 어셈블리 코드

unrolled loop의 어셈블리 코드는 예상했겠지만 코드가 더 길습니다. 한 루프에서 수행하는 일이 위의 rolled loop보다 4개가 많기 때문입니다. 하지만 루프의 반복 횟수는 1/4이고 파이프라인을 생각하면 더 빠를 것이라 예상할 수 있습니다.

```

real    0m0.106s
user    0m0.000s
sys     0m0.000s
[비누~/code/radare/unrolled]$

```

[그림 27] rolled loop의 수행 시간

rolled loop는 수행시간으로 0.106초가 걸렸습니다.

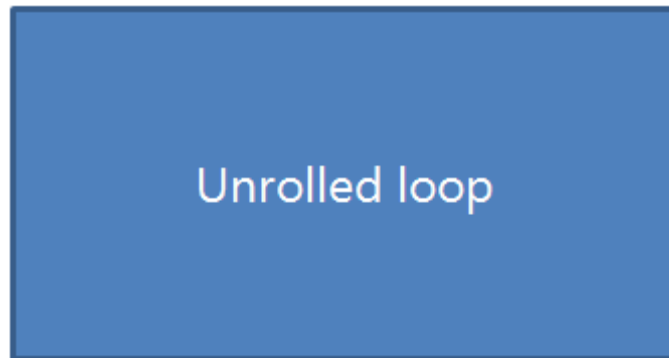
```

real    0m0.078s
user    0m0.000s
sys     0m0.000s
[비누~/code/radare/unrolled]$

```

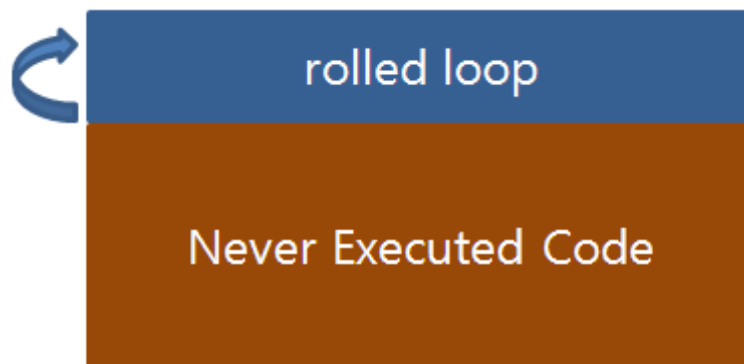
[그림 28] unrolled loop의 수행 시간

unrolled loop는 0.078초가 걸렸습니다. 즉 rolled loop와 unrolled loop는 결과는 같지만 수행시간에 있어서 차이를 보입니다. 최적화와 관련된 이야기란 것인데 이게 코드 인젝션과 무슨 관계가 있을까요?



[그림 29] 초기의 unrolled loop

위와 같은 unrolled loop가 있다고 가정하고, Unrolled loop는 rolled loop보다 코드가 길다고 하였지만 결과는 같다고 하였습니다. 물론 그것을 눈으로 확인하였습니다. 따라서 위의 unrolled loop를 조작해서 rolled loop로 변경해 버리면 나머지 코드는 절대로 실행되지 않는 코드가 될 것입니다.



[그림 30] 조작된 루프

(그림 30)과 같이 unrolled loop를 rolled loop로 조작하게 되면 잔여 공간이 생기게 되고 이 공간은 절대로 실행되지 않습니다. 따라서 이 공간에 코드를 삽입하게 되어도 프로그램 실행에 전혀 문제가 되지 않습니다.

2.4.3. Function padding and preludes

컴파일러가 함수를 정렬하는 방법에는 padding이 있습니다. padding은 CPU가 주소 연산을 쉽게 할 수 있도록 미리 주소를 정렬하는 데 사용하는 기법입니다.



```
0x08048428, / sym.main: 8d4c2404    lea ecx, [esp+0x4]
0x0804842c, |           83e4f0    and esp, 0xf0
0x0804842f, |           ff71fc    push dword [ecx-0x4]
0x08048432, |           55       push ebp
0x08048433, |           89e5     mov ebp, esp
0x08048435, |           51       push ecx
0x08048436, |           b800000000 mov eax, 0x0
0x0804843b, |           59       pop ecx
0x0804843c, |           5d       pop ebp
0x0804843d, |           8d61fc    lea esp, [ecx-0x4]
0x08048440, |           c3       ret
0x08048440, |           ; -----
0x08048441, |           90       nop
0x08048442, |           90       nop
0x08048443, |           90       nop
0x08048444, |           90       nop
0x08048445, |           90       nop
0x08048446, |           90       nop
0x08048447, |           90       nop
0x08048448, |           90       nop
0x08048449, |           90       nop
0x0804844a, |           90       nop
0x0804844b, |           90       nop
0x0804844c, |           90       nop
0x0804844d, |           90       nop
0x0804844e, |           90       nop
0x0804844f, |           90       nop
; args = 0
; vars = 0
; drefs = 0
0x08048450, / sym.__libc_csu_fini:
0x08048450, |           55       push ebp
0x08048451, |           89e5     mov ebp, esp
0x08048453, |           5d       pop ebp
```

[그림 31] Function Padding

(그림 31)에서 padding의 예를 확인할 수 있습니다. main함수 블록이 끝나고 __libc_csu_fini()함수 사이에 0x90의 padding이 존재합니다. 이러한 공간은 실제로 사용되지 않는 코드들이며, 이 공간에 코드를 삽입할 수 있습니다.

2.4.4. Compiler stubs

gcc로 프로그램을 컴파일 하게 되면 컴파일러가 생성한 코드 중 필요하지만 없어도 되는 코드가 존재합니다. 생성자/소멸자 코드는 이에 해당하는 부분입니다. C언어로 작성된 프로그램에도 생성자와 소멸자 비슷한 코드가 존재하는데 fini와 init입니다. 다음은 _start함수의 어셈블리 코드입니다.

```

0x08048310, / entrypt, sym._start, section._text:
0x08048310, | 31ed      xor ebp, ebp
0x08048312, | 5e        pop esi
0x08048313, | 89e1      mov ecx, esp
0x08048315, | 83e4f0    and esp, 0xf0
0x08048318, | 50        push eax
0x08048319, | 54        push esp
0x0804831a, | 52        push edx
0x0804831b, | 68f0830408 push dword 0x80483f0 ; sym.__libc_csu_fini
0x08048320, | 6800840408 push dword 0x8048400 ; sym.__libc_csu_init
0x08048325, | 51        push ecx
0x08048326, | 56        push esi
0x08048327, | 68c4830408 push dword 0x80483c4 ; sym.main
0x0804832c, | e8b3ffff  call 0x80482e4 ; 1 = imp.__libc_start_main

```

[그림 32] _start 함수

_start함수는 생성자/소멸자 함수 포인터를 설정하고 main() 함수를 호출합니다(물론 실제 main()함수는 __libc_start_main()에서 호출합니다.). 생성자와 소멸자 함수 중 생성자의 경우 상당히 큰 코드블럭입니다.

```

[0x08048310]> af @ sym.__libc_csu_init
offset = 0x08048400=====
label = sym.__libc_csu_init
size = 89

```

[그림 33] 생성자 함수의 크기

재밌는 것인 굳이 이런 함수가 호출되지 않아도 실행에 지장이 없다는 것입니다. 함수 포인터를 push 하는 부분에서 실제 함수 주소 대신 0 즉 NULL을 push 하도록 변경 해 보겠습니다.

```

0x0804831a, | 52        push edx
0x0804831b, | 6800000000 push dword 0x0
0x08048320, | 6800000000 push dword 0x0
0x08048325, | 51        push ecx

```

[그림 34] push 0

이제 실제 생성자/소멸자 함수 포인터 대신에 0을 push 하게 되었습니다. 이 경우 프로그램은 널 포인터를 검사한 후 널이라면 함수를 호출하지 않게 됩니다. 실제 실행이 되는지 확인 해 보겠습니다.

```

[비누~/code/radare/padding]$ ./stubs
Executed!!
[비누~/code/radare/padding]$ █

```

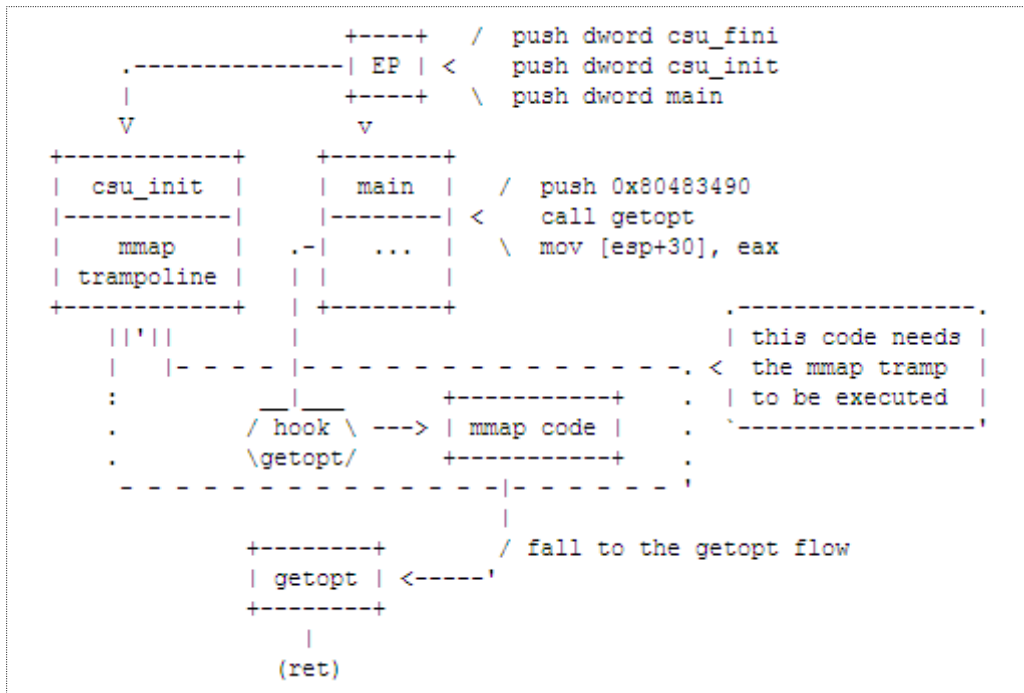
[그림 35] 정상적으로 실행되는 프로그램

따라서 이제 이 프로그램의 생성자/소멸자 함수는 절대로 실행되지 않을 것이고 그 공간에는 코드 인젝션이 가능 합니다.

2.5. 코드인젝션의 기초

2.5.1. mmap trampoline

mmap 트램펄린은 프로그램의 특정 함수를 후킹해서 mmap 코드가 실행되도록 만들고 mmap 코드가 실행된 후에 다시 호출하고자 했던 함수를 호출하는 것입니다.



[그림 36] mmap 트램펄린의 구조

위와 같은 구조를 가집니다. 특히 코드인젝션을 위해서 init()함수를 사용하는 것을 확인할 수 있습니다. 위에서 살펴본 대로 init() 함수는 컴파일 시 항상 존재하지만 호출되지 않아도 문제가 되지 않는 함수입니다. 따라서 init()함수가 호출되지 않도록 수정한 뒤 이 영역에 mmap 트램펄린 코드를 쓰는 것입니다.

1) csu_init() 포인터 수정

위에서 잠깐 언급했던 부분입니다. 먼저 csu_init()함수가 호출되지 않아야 조건이 성립되기 때문에 함수 포인터를 수정해야 합니다. 2.4.4.에서 살펴본 것처럼 수정합니다.

```

0x0804831b | 6850340408 | push dword 0x8048450 ; sym.__libc_
0x08048320, | >6800000000 | push dword 0x0
  
```

[그림 37] csu_init() 포인터 -> NULL

2) 트램펄린 코드

csu_init()의 공간에는 원래의 초기화 함수 대신에 mmap함수는 눈으로 확인하기 어렵기 때문에 여기서는 puts와 같은 눈으로 확인 가능한 함수를 사용해 보겠습니다.

```
.global main
main:
push $0x00
push $0x61616161
push %esp
call puts
add $0xc, %esp
jmp 0x8048000
ret
```

[그림 38] 삽입될 코드

위 코드를 csu_init()에 삽입하고 후킹 될 함수에서 코드를 실행합니다. 그리고 마지막 jmp에서 다시 원하던 함수를 실행하게 하는 것입니다.

3) 코드 추출

(그림 38)의 어셈블리코드에서 실제 삽입 할 바이너리 코드로 추출합니다.

```
[비누~/code/radare/mmap]$ # inject 할 byte 추출
[비누~/code/radare/mmap]$ rabin ./tram -o d/s | grep ^main | cut -
d ' ' -f 2 > inj_code
[비누~/code/radare/mmap]$ # 내용 확인
[비누~/code/radare/mmap]$ cat inj_code
6a00686161616154e823ffffff83c40ce927fcffffc3909090909090
[비누~/code/radare/mmap]$
```

[그림 39] rabin을 사용한 바이트 추출

'rabin'을 사용하면 바이너리 코드 추출을 쉽게 할 수 있습니다. 추출된 결과는 파일에 저장하고 radare에서 사용할 수 있습니다.

4) csu_init()에 코드 삽입

이제 타깃 프로그램을 불러오고 csu_init() 함수 영역에 추출한 코드를 삽입합니다. 먼저 코드가 삽입되기 전의 csu_init() 함수의 내용입니다.

```
[ 0x08048440 (bs=512 mark=0x0) (null) (null)] sym.__libc_csu_init
[-----#-----]
; framesize = 12,0
; args = 3
; vars = 0
; drefs = 0
0x08048440, / sym.__libc_csu_init:
0x08048440, | 55          push ebp
0x08048441, | 89e5        mov ebp, esp
0x08048443, | 57          push edi
0x08048444, | 56          push esi
0x08048445, | 53          push ebx
0x08048446, | e84f000000  call 0x804849a ; 1 = sym.__i6
0x0804844b, | 81c3a91b0000 add ebx, 0x1ba9
; Stack size +12
0x08048451, | 83ec0c      sub esp, 0xc
0x08048454, | e863feffff  call 0x80482bc ; 2 = sym._ini
0x08048459, | 8dbb18fffff  lea edi, [ebx-0xe8]
0x0804845f, | 8d8318fffff  lea eax, [ebx-0xe8]
0x08048465, | 29c7        sub edi, eax
```

[그림 40] 코드가 삽입되기 전의 csu_init()함수의 내용

csu_init()은 위와 같은 내용입니다. 이제 이 영역에 코드를 삽입해 보겠습니다.

```
[0x08048440]> wF ini_code@ sym.__libc_csu_init
```

[그림 41] 코드 삽입

코드 삽입은 위와 같이 한 줄로 끝납니다. 'wF' 명령은 16진수 값들이 저장된 파일을 열어서 원하는 주소에 쓰는 명령입니다. 따라서 인자로 파일명이 와야 하고 @를 사용하여 위치를 지정해 주어야 하는데 위와 같이 csu_init()의 심볼을 넘겨주면 됩니다. 삽입이 정상적으로 되었다면 다음과 같은 결과를 확인할 수 있습니다.

```
[ 0x08048440 (bs=512 mark=0x0) (null) (null)] sym.__libc_csu_init
[-----#-----]
; framesize = 12,0
; args = 3
; vars = 0
; drefs = 0
| 0x08048440, / sym.__libc_csu_init:
| 0x08048440, | 6a00        push 0x0
| 0x08048442, | 6861616161  push dword 0x61616161 ; (0x616
| 0x08048447, | 54          push esp
| 0x08048448, | e823fffff  call 0x8048370 ; 1 = sym.__do
| 0x0804844d, | 83c40c      add esp, 0xc
| 0x08048450, | e927fcffff  jmp 0x804807c ; 2 = 0x0804807
| 0x08048455, | c3          ret
| 0x08048455, | ; ----->
```

[그림 42] 코드 삽입 후의 변화

코드가 바뀌었고 블록의 크기가 줄어든 것을 확인할 수 있습니다. 하지만 위의 코드를 그대로 사용할 수가 없습니다. 왜냐하면 call 함수나 jmp 함수는 자신의 위치에 상대적인 주소 계산을 하기 때문에 처음 의도했던 주소가 틀어지게 되었습니다. 따라서 다시 주소를 써 주

어야 합니다. 먼저 puts 주소 대신 프로그램에서 사용가능 한 printf를 사용해야 하는데 주소는 main()에서 분석 가능합니다.

```
[ 0x80483f4 (bs=512 mark=0x0) (null) (null)] sym.main
[ .....#.....]
; framesize = 20,0
; args = 0
; vars = 0
; drefs = 1
0x080483f4, / sym.main: 8d4c2404      lea ecx, [esp+0x4]
0x080483f8, |          83e4f0      and esp, 0xf0
0x080483fb, |          ff71fc      push dword [ecx-0x4]
0x080483fe, |          55          push ebp
0x080483ff, |          89e5      mov ebp, esp
0x08048401, |          51          push ecx
; Stack size +20
0x08048402, |          83ec14      sub esp, 0x14
0x08048405, |          e822ffff      call 0x804832c ; 1 = imp.getu
0x0804840a, |          89442404      mov [esp+0x4], eax
0x0804840e, |          c70424f0840408 mov dword [esp], 0x80484f0 ; s
0x08048415, |          e802ffff      call 0x804831c ; 2 = imp.prin
0x0804841a, |          b800000000      mov eax, 0x0
; Stack size -20
0x0804841f, |          83c414      add esp, 0x14
0x08048422, |          59          pop ecx
```

[그림 43] main() 함수의 내용

타깃 프로그램은 getuid()를 호출해서 결과를 printf()를 사용해서 출력해 주는 프로그램입니다. 여기서 getuid() 함수의 plt 주소로 0x804832c를 사용하고 printf는 0x804831c를 사용하고 있습니다. 따라서 삽입된 코드도 위와 같은 주소로 재지정 해주어야 합니다.

```
[0x08048440]> # printf의 주소 지정
[0x08048440]> !!rasm -s 0x8048448 -ev 'call 0x804831c'
e8 cf fe ff ff
[0x08048440]> # getuid의 주소 지정
[0x08048440]> !!rasm -s 0x8048450 -ev 'call 0x804832c'
e8 d7 fe ff ff
[0x08048440]>
```

[그림 44] printf와 getuid 호출 코드 획득

(그림 44)처럼 rasm을 사용해서 opcode를 다시 확인이 가능합니다. call과 jmp는 opcode 위치에 대해서 상대적인 offset을 계산하기 때문에 꼭 -s 플래그로 opcode의 주소를 함께 주어야 합니다. 이제 알아낸 주소를 삽입된 코드에 수정해 줍니다.

```
[ 0x804843c (bs=512 mark=0x0) (null) (null)] sym.__libc_csu_fini+0xc
[-----#-----]
0x0804843c,      0000      add [eax], al
0x0804843e      0000      add [eax], al
; framesize = -12
; args = 0
; vars = 0
; drefs = 1
0x08048440, / sym.__libc_csu_init:
0x08048440, |      6a00      push 0x0
0x08048442, |      68616161  push dword 0x61616161 ; (0x616
0x08048447, |      54        push esp
0x08048448, |      e8cffefff call 0x804831c ; 1 = imp.prin
; Stack size -12
0x0804844d, |      83c40c    add esp, 0xc
0x08048450, |      e8d7fefff call 0x804832c ; 2 = imp.getu
0x08048455, |      c3        ret
0x08048455, |      ; -----
```

[그림 45] 수정된 csu_init() 함수

이제 의도된 대로 코드가 수정되었습니다.

5) call 후킹

마지막으로 call 명령어를 후킹 해야 합니다. main()에서 getuid() 함수를 호출할 때 csu_init()을 호출 하도록 하고 csu_init()에서 다시 getuid()를 호출하게 하면 됩니다. main()의 getuid 호출 부분을 0x8048440 즉 csu_init()을 호출 하도록 수정합니다.

먼저 수정될 명령어를 구합니다. 역시 rasm을 사용합니다.

```
[비누~/code/radare/mmap]$ rasm -s 0x8048405 -ev 'call 0x8048440'
e8 36 00 00 00
[비누~/code/radare/mmap]$
```

[그림 46] 수정될 명령어

명령어는 'e8 36 00 00 00'으로 수정되어야 합니다. 다음과 같이 수정합니다.

```
0x08048402, |      83ec14    sub esp, 0x14
0x08048405, |      e83600000 call 0x8048440 ; 1 = sym.__li
0x0804840a, |      *80484204 mov [esp+0x4], eax
```

[그림 47] 수정된 call 명령

수정되었습니다. 이제 프로그램을 실행해서 결과를 비교해 보겠습니다.

```
[비누~/code/radare/mmap]$ ./target
UID : 1000
[비누~/code/radare/mmap]$
```

[그림 48] 바이너리를 조작하기 전

바이너리를 조작하기 전에는 (그림 48)과 같이 'UID : 1000'이라는 결과를 보여주었습니다. 하지만 바이너리를 조작하고 난 후에는 결과가 조금 다릅니다.

```
[비누~/code/radare/mmap]$ ./target
aaaaUID : 1000
[비누~/code/radare/mmap]$
```

[그림 49] 바이너리를 조작한 후

바이너리가 조작된 후에는 UID앞에 'aaaa'가 덧붙여졌습니다. 이 문자열은 csu_init()에서 출력된 것이고 뒤에 '1000'이라는 UID가 정상적으로 출력된 것으로 보아 getuid()함수가 정상적으로 호출 되었다는 것을 알 수 있습니다. 상당히 많은 부분에서 바이너리가 조작되었지만 실행에는 아무런 문제가 없다는 것을 알 수 있습니다.

정리하면 다음과 같습니다.

- 코드가 삽입될 공간을 확보하기 위하여 csu_init() 함수포인터를 NULL로 수정 (약 100바이트의 공간이 확보됨)
- 추가로 실행 할 코드를 csu_init() 함수 영역의 첫 바이트부터 복사
- 삽입된 코드의 jmp나 call 명령의 수정
- 후킹 할 함수의 call 명령을 수정
- 끝

2.5.2. 응용 가능한 트램펄린

1) Call 트램펄린

2.5.1.에서 트램펄린에 대한 간단한 예제를 확인하였습니다. 이를 응용하면 코드 난 독화에 사용할 수 있는데 Call 트램펄린이라는 것을 예로 들어보겠습니다. Call 트램펄린이란 Call 되는 모든 명령어를 수정하는 것입니다. 그리고 실제 프로그램의 동작에는 이상이 없도록 Call을 위한 테이블을 .data 섹션에 따로 유지해서 이 테이블을 참조하여 함수 호출을 하도록 합니다.

이렇게 되면 프로그램을 리버싱하는 과정에서 분석하기가 힘들어 집니다. 왜냐하면 어떤 함수를 분석함에 있어서 역 참조 정보가 매우 중요하고 키포인트가 되는 경우가 많은데 실제 이 함수를 호출한 함수를 알 수가 없기 때문입니다.

2) 암호화 된 코드영역을 위한 트랩펄린

보호하고자 하는 프로그램에서 실제 코드(.text영역의 코드들)를 어떤 암호화 알고리즘과 키를 사용하여 암호화 한 뒤 실제 함수를 사용하기 위해서 복호화 한 후 실행할 수 있습니다. 이 경우 함수가 호출되지 않은 시점에서는 항상 암호화 되어져 있기 때문에 크래커들로부터 코드를 보호할 수 있습니다. 하지만 프로그램이 실행되는 시점에서 함수가 사용될 때 코드가 복호화 되어져야 하기 때문에 그 시점에서 코드가 유출 될 수 있습니다.

2.6. 프로그램 보호를 위한 조작

2.6.1. ELF 헤더 조작

ELF 헤더는 프로그램의 분석을 위한 중요한 정보들을 가지고 있습니다. 실제로 많은 바이너리 분석 도구들은 ELF 헤더를 가장 먼저 분석하고, 분석한 정보로부터 유용한 정보들을 찾아갑니다. 따라서 ELF 헤더를 조작하게 되면 공격자로부터 프로그램을 보호하는데 유용합니다.

ELF헤더를 조작하는 가장 간단한 방법은 섹션헤더의 시작 위치를 조작하는 것입니다. 이것을 조작한다고 해서 프로그램의 실행에 지장이 가는 것은 아니지만 분석하는 입장에서는 충분히 방해가 됩니다. 섹션헤더의 위치 값은 파일 오프셋 0x21에 위치하기 때문에 radare를 사용하여 다음과 같이 쉽게 조작이 가능합니다.

```
[0x08048310]> wx 99 @ 0x21
[0x08048310]> q
00 + 1 00000021: 17 => 99
Do you want save these changes? (Y/n)
[비누~/code/radare/protection]$
```

[그림 50] 섹션의 시작 오프셋 수정

위처럼 수정하게 되면 섹션헤더 오프셋의 1바이트가 수정됩니다. 수정 후 readelf를 사용해서 섹션 정보를 불러오게 하면 다음과 같은 결과를 보입니다.

```
[0x08048310]> wx 99 @ 0x21
[0x08048310]> q
00 + 1 00000021: 17 => 99
Do you want save these changes? (Y/n)
[비누~/code/radare/protection]$ readelf -S ./target
readelf: Error: Unable to read in 0x28 bytes of section headers
There are 36 section headers, starting at offset 0x996c:
readelf: Error: Unable to read in 0x5a0 bytes of section headers
[비누~/code/radare/protection]$
```

[그림 51] readelf를 사용하여 섹션 정보 읽기

-S 플래그를 사용하여 섹션 정보를 읽어 오고자 하였으나 실패하였습니다. 사유는 섹션

헤더의 위치가 틀렸기 때문입니다. 이러한 경우 gdb로 디버깅도 할 수 없게 됩니다.

```
(gdb) r
Starting program:
No executable file specified.
Use the "file" or "exec-file" command.
(gdb) □
```

[그림 52] gdb로 실행 실패

마치 GDB로 로드한 프로그램이 실행 파일이 아닌 것으로 간주해 버립니다. 하지만 다음과 같이 일반적인 실행에는 문제가 없습니다.

```
[비누~/code/radare/protection]$ ./target
Executed!!
[비누~/code/radare/protection]$ □
```

[그림 53] 일반적인 실행 시 반응

(그림 53)에서처럼 일반적인 실행에 문제가 없는 것을 확인 할 수 있습니다. 프로그램을 실행하는데 있어서 섹션 헤더의 시작 위치 정보가 중요하지 않기 때문입니다. 하지만 분석하는 도구들은 이러한 정보가 매우 중요합니다.

2.6.2. 소스레벨의 워터마크 사용

만약 실제 프로그램을 개발하는 입장이라면, 소스코드를 가지고 있다면 소스코드에 워터마크를 삽입해서 프로그램을 보호할 수 있습니다. 만약 아래의 코드를 사용한다고 생각해 보겠습니다.

```
#define WATERMARK __asm__ __volatile__(".byte 0x50, 0x58, 0x50, 0x58, 0x50, 0x58");

int main()
{
    WATERMARK
    printf("Executed!!\n");
    WATERMARK
}
```

[그림 54] watermark 가 삽입될 프로그램

사실 워터마크는 꼭 보호가 아니더라도 여러 가지 목적에 의해서 사용될 수 있습니다. 만약 위와 같이 사용되어졌다면 어셈블리코드는 다음과 같이 나타날 것입니다.

```

; Stack size +20
0x080483d2 | 83ec14 | sub esp, 0x14
0x080483d5 | 50 | push eax
0x080483d6 | 58 | pop eax
0x080483d7 | 50 | push eax
0x080483d8 | 58 | pop eax
0x080483d9 | 50 | push eax
0x080483da | 58 | pop eax
0x080483db | c70424c0840408 | mov dword [esp], 0x80484c0 ; str.E
0x080483e2 | e80dffffff | call 0x80482f4 ; 1 = imp.puts
0x080483e7 | 50 | push eax
0x080483e8 | 58 | pop eax
0x080483e9 | 50 | push eax
0x080483ea | 58 | pop eax
0x080483eb | 50 | push eax
0x080483ec | 58 | pop eax
; Stack size -20
0x080483ed | 83c414 | add esp, 0x14
0x080483f0 | 59 | pop ecx
0x080483f1 | 5d | pop ebp
0x080483f2 | 8d61fc | lea esp, [ecx-0x4]
0x080483f5 | c3 | ret

```

[그림 55] watermark가 삽입된 프로그램

watermark는 실제로 기계어 코드이지만 해석되어도 프로그램의 실행에 전혀 문제가 없는 코드들입니다. 위의 경우에는 eax 레지스터를 push, pop 반복하는 것입니다. radare를 사용하게 되면 저런 워터마크를 쉽게 찾아낼 수 있습니다. 물론 패턴을 알고 있어야 합니다.

```

0x080483f9 | 90 | nop
0x080483fa | 90 | nop
0x080483fb | 90 | nop
[0x080483C4]> /x 50585058
001 0x000003d5 hit0_1 PXPXPX$PX
002 0x000003e7 hit0_2 PXPXPXY]a
[0x080483C4]> 

```

[그림 56] 찾아 낸 워터마크

워터마크의 오프셋을 찾아냈습니다. 여기서 워터마크의 위치를 찾은 것이 중요한 것은 아닙니다. 중요한 점은 워터마크가 nop 코드처럼 프로그램의 진행에 있어서 전혀 중요한 코드가 아니라는 것입니다. 따라서 워터마크도 하나의 코드를 삽입할 수 있는 공간으로 활용될 수 있습니다.

2.6.3. .data 섹션의 암호화

프로그램을 보호하기 위해서 가장 확실한 방법 중 하나는 바이너리 자체를 암호화 해 버리는 것이고 필요할 때만 복호화를 하는 것입니다. 다음은 radare에서 사용할 수 있는 스크립트입니다.


```

# flag from virtual address (the base address where the binary is mapped)
# this will make all new flags be created at current seek + <addr>
# we need to do this because we are calculating virtual addresses.
ff ${io.vaddr}

# Display the ranges of the .data section
?e Data section ranges : `?v section._data`- `?v section._data_end`

# set flag space to 'segments'. This way 'f' command will only display the
# flags contained in this namespace.
fs segments

# Create flags for 'virtual from' and 'virtual to' addresses.
# We grep for the first row '#0' and the first word '[0]' to retrieve
# the offset where the PT_LOAD segment is loaded in memory
f vfrom @ `f~vaddr[0]#0`

# 'virtual to' flag points to vfrom+ptload segment size
f vto @ vfrom+`f~vaddr[1]#0`

# Display this information
?e Range of data decipher : `?v vfrom`- `?v vto`= `?v vto-vfrom`bytes

# Create two flags to store the position of the physical address of the
# PT_LOAD segment.
f pfrom @ `f~paddr[0]#0`
f pto @ pfrom+`f~paddr[1]#0`

# Adjust deltas against data section
# -----
# pdelta flag is not an address, we set flag from to 0
# pdelta = (address of .data section)-(physical address of PTLOAD segment)
ff 0 && f pdelta @ section._data-pfrom
?e Delta is `?v pdelta`

# reset flag from again, we are calculating virtual and physical addresses
ff ${io.vaddr}
f pfrom @ pfrom+pdelta
f vfrom @ vfrom+pdelta

```

```

ff 0

?e Range of data to cipher: `?v pfrom`- `?v pto`= `?v pto-pfrom`bytes

# Calculate the new physical size of bytes to be ciphered.
# we dont want to overwrite bytes not related to the data section
f psize @ section._data_end - section._data
?e PSIZE = `?v psize`

# 'wox' stands for 'write operation xor' which accepts an hexpair list as
# a cyclic XOR key to be applied to at 'pfrom' for 'psize' bytes.
wox a8 @ pfrom:psize

# Inject shellcode
#-----

# Setup the simple profile for the disassembler to simplify the parsing
# of the code with the internal grep
e asm.profile=simple

# Seek at entryptoint
s entryptoint

# Seek at address (fourth word '[3]') at line'#1' line of the disassembly
# matching the string 'push dword' which stands for the '__libc_csu_init'
# symbol. This is the constructor of the program, and we can modify it
# without many consequences for the target program (it is not widely used).
s `pd 20~push dword[3]#1`

# Compile the 'uxor.S' specifying the virtual from and virtual to addresses
!gcc uxor.S -DFROM=`?v vfrom` -DTO=`?v vfrom+psize`

# Extract the symbol 'main' of the previously compiled program and write it
# in current seek (libc_csu_init)
wx `!!rabin -o d/s a.out|grep ^main|cut -d ' ' -f 2`

# Cleanup!
?e Uncipher code: `?v $$+${io.vaddr}`
!!rm -f a.out

```

q!

[표 4] 암호화 스크립트

이 스크립트는 타깃 프로그램의 .data 섹션을 xor 연산해서 암호화 합니다. 따라서 프로그램을 정적으로 분석할 경우 .data 섹션의 데이터를 알아내기 어렵습니다.

다음은 예제의 타깃 프로그램 코드입니다.

```
#include <stdio.h>

char message[128] = "Hello World";

int main()
{
    if(message[0]!='H')
        printf("Oops\n");
    else
        printf("%s\n", message);
    return 0;
}
```

[그림 57] 예제 프로그램

먼저 암호화를 거치지 않은 경우의 프로그램을 실행하면 다음과 같습니다.

```
[비누~/code/radare/protection]$ ./hello_orig.
Hello World
[비누~/code/radare/protection]$ strings ./hello_orig. | grep Hello
Hello World
[비누~/code/radare/protection]$
```

[그림 58] 암호화 되지 않은 프로그램의 실행

프로그램을 실행하면 Hello World가 출력되고 그 문자열이 바이너리에 그대로 존재한다는 것을 알 수 있습니다. 암호화를 한 경우를 보겠습니다.

```
[비누~/code/radare/protection]$ ./hello
Hello World
[비누~/code/radare/protection]$ strings ./hello | grep Hello
[비누~/code/radare/protection]$
```

[그림 59] 암호화된 프로그램의 실행

프로그램은 정상적으로 실행되었습니다. 하지만 'Hello World'라는 문자열이 바이너리에 존재하지 않았습니다. 즉 정적인 분석에서 위 문자열은 발견할 수 없습니다.

2.7. 라이브러리 의존성 제거

2.7.1. 라이브러리 의존성 제거의 의미

라이브러리의 의존성 제거는 사실 많이 유용한 방법은 아닌 것 같습니다. 라이브러리가 의존되어져 있다는 것은 곧 라이브러리의 어떤 코드가 사용된다는 것인데 이러한 의존성을 제거해 버리면 프로그램은 충돌 날게 뻔합니다. 하지만 어떤 경우에 라이브러리 의존성을 제거할 때 유용한 경우가 있습니다.

프랙의 문서에서는 이 상황을 libselinux를 예로 들고 있습니다. 이 라이브러리가 의존되어 있을 경우 바이너리는 SELinux에 종속적이게 되어 집니다. 따라서 이러한 라이브러리의 의존성을 제거해서 종속에서부터 피할 수 있는 것입니다.

2.7.2. 라이브러리 의존성의 파악

라이브러리의 의존성은 사실 매우 쉽게 확인 가능합니다. 단순히 라이브러리 이름을 검색해서 의존성을 나타내는데 만약 NULL이라면 라이브러리 의존 테이블에서 제거되게 됩니다. ELF형식에서 모든 문자열 테이블은 연속된 문자열로 나타내 지는데, 각 문자열은 NULL로써 구분 됩니다.

2.7.3. 라이브러리 의존성 제거 방법

만약 radare를 사용해서 libc.so.6 라이브러리 의존성을 제거하고 싶다면 다음과 같이 수행 될 수 있습니다.

```
[0x08048310]> / libc.so.6
001 0x00000210 hit0_1 libc.so.6_10_stdin_
[0x08048310]> wx 00 @ hit0_1
[0x08048310]> 
```

[그림 60] libc.so.6 검색

radare에서 '/'는 검색에 사용됩니다. (그림 60)은 'libc.so.6'을 검색한 것이고 0x210 위치에 존재한다는 것을 알게 되었습니다. 또한 이 오프셋은 hit0_1이라는 플래그로 저장되어졌습니다. 따라서 wx 명령을 사용해서 그 위치에 0x00을 기록하면 됩니다. 하지만 예상했듯이 이 라이브러리는 실행에 있어서 매우 중요한 라이브러리입니다. 즉 의존성이 제거되면 실행할 수 없게 됩니다.

2.8. 시스템 콜 난 독화

2.8.1. 디버깅 정보로서의 시스템 콜

보통 시스템 콜을 호출하기 위해서 `int 0x80`이 사용된다는 것을 알고 있을 것입니다. 시스템 콜 호출에 대한 루틴은 매우 직관적이고 보기 편합니다. `int 0x80` 명령을 보면 당연히 시스템 콜이 호출 될 것이라는 사실을 알고 있으며, 이 순간 `eax` 레지스터의 값을 확인하면 어떤 함수가 호출 될 것인지도 알 수 있습니다. 이렇게 쉽게 이해할 수 있는 정보는 디버깅 정보로서 가치가 있습니다.

2.8.2. 시스템콜 호출 방법의 변화

그동안의 시스템 콜 호출은 `int 0x80` 명령이 담당해 왔습니다. 하지만 CPU의 발전에 따라 `int 0x80` 명령에 병목 현상이 생겨서 `sysenter`라는 새로운 명령이 도입되었습니다. 이 명령은 pentium pro에서부터 제공 되었는데, 실제로 이 명령이 사용된 것은 최근 일입니다. 아마 최근의 리눅스 공유라이브러리를 보면 `vdso`라는 이름이 발견되는데 이 라이브러리의 정체에 대해서 궁금했던 사람이 많을 것입니다.

```
che
b7fd5000-b7fd6000 r--p 00000000 08:06 8594962 /usr/lib/locale/ko_KR.utf8/LC_
IDENTIFICATION
b7fd6000-b7fd8000 rw-p b7fd6000 00:00 0
b7fd8000-b7fd9000 r-xp b7fd8000 00:00 0 [vdso]
b7fd9000-b7ff5000 r-xp 00000000 08:06 12673047 /lib/ld-2.9.so
b7ff5000-b7ff6000 r--p 0001b000 08:06 12673047 /lib/ld-2.9.so
b7ff6000-b7ff7000 rw-p 0001c000 08:06 12673047 /lib/ld-2.9.so
bffe1000-bffe6000 rw-p bffe1000 00:00 0 [stack]
[비누~/code/radare/protection]$
```

[그림 61] vdso 라이브러리

(그림 61)의 라이브러리는 실제 라이브러리가 아닌 가짜 라이브러리입니다. 풀 네임은 Virtual Dynamic Shared Object입니다. 즉 가상 동적 공유 객체라는 것인데, 이 라이브러리를 통해서 시스템 콜을 호출할 수 있습니다. 즉 `sysenter`를 사용하는 것입니다.

2.8.3. 시스템 콜의 난 독화

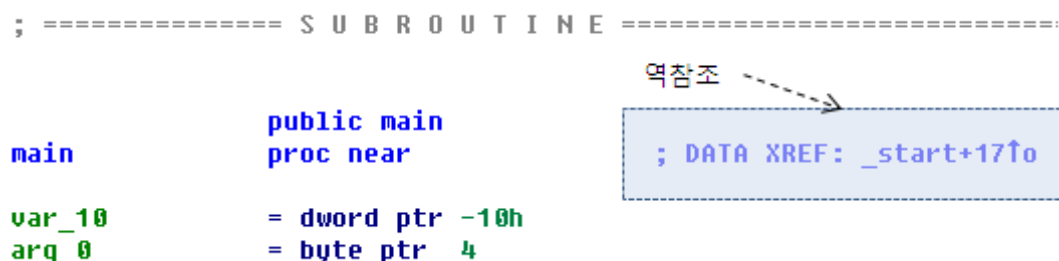
시스템 콜 난 독화를 위해서 `int 0x80` 명령어를 변경할 수 있습니다. `vdso`를 사용한 시스템 콜 호출 형식으로 바꾸는데, `int 0x80`을 사용하는 것 보다 더 많은 코드들이 추가될 수 있고 코드 분석을 어렵게 만들 수 있기 때문에 충분히 리버싱하는데 방해가 될 수 있습니다.

3. 역 참조(xrefs) 정보

3.1. 역 참조 찾기

3.1.1. 역 참조 정보의 유용성

역참조란 어떤 코드나 데이터가 어디에서 사용되었는지에 대한 정보입니다. 이런 정보는 리버싱하는 가운데 매우 유용합니다. 왜냐하면 어떤 코드 블록을 리버싱하다 보면 실제로 이 함수가 어디에서 사용되는지 알게 된다면 프로그램의 흐름을 더 쉽게 이해할 수 있고, 커다란 그림을 연상시킬 수 있기 때문입니다. 이런 역 참조 정보는 IDApro에서 매우 잘 제공됩니다.



[그림 62] IDA에서 보여주는 역참조 정보

IDA에서는 코드 블록의 시작부분에 현재 블록이 어디에서 사용되는지 알려줍니다. 위의 경우에는 `_start+17`에서 이 코드 블록의 첫 주소가 사용됨을 암시합니다.

3.1.2. radare의 역 참조 정보 관리

그렇다면 radare의 경우에는 이것을 어떻게 관리할까요? radare는 Cx와 CX명령어를 사용하여 역 참조 정보를 관리합니다. 소문자 x는 코드를 의미하고 대문자 X는 데이터를 의미합니다. 프로그램을 radare에 로드하고 나서 Cx를 사용하면 다음과 같이 역 참조 정보를 보여줍니다.

```
[0x080483C4]> Cx  
Cx 0x0804837b @ 0x08048460 ; sym.__do_global_ctors_aux+0x3b  
Cx 0x08048316 @ 0x080483a0 ; entrypoint+0x6  
Cx 0x0804843d @ 0x080482d4 ; sym.__libc_csu_init+0x3d  
Cx 0x080482a7 @ 0x080482a0 ; sym._init+0x13  
Cx 0x080483ec @ 0x080482f4 ; sym.main+0x28  
Cx 0x08048560 @ 0x08048294 ; section._eh_frame_end+0x98  
Cx 0x08048460 @ 0x0804845a ; sym.frame_dummy+0xc0  
Cx 0x080484cf @ 0x08048340 ; section._eh_frame_end+0x7  
Cx 0x0804849f @ 0x08048498 ; sym._fini+0x13
```

[그림 63] radare의 Cx

또한 radare의 Visual모드에서 IDA처럼 Xrefs 정보가 주석으로 표기됩니다. 하지만 이 경우 IDA처럼 가독성이나 정확성 면에서 아직은 떨어지기 때문에 좀 더 보완이 필요해 보입니다.

```

0x080483c4, / sym.main: 8d4c2404      lea ecx, [esp+0x4]
0x080483c8, |           83e4f0      and esp, 0xf0
0x080483cb, |           ff71fc      push dword [ecx-0x4]
0x080483ce, |           55          push ebp
0x080483cf, |           89e5      mov ebp, esp
0x080483d1, |           51          push ecx
; Stack size +4
0x080483d2, |           83ec04      sub esp, 0x4
; 0x080483d5 DATA xref from 0x080484b0 (sym._IO_stdin_used+0x4)
; 0x080483d5 DATA xref from 0x080484b0 (sym._IO_stdin_used+0x4)
0x080483d5, |           c70424b08408 mov dword [esp], 0x80484b0 ; s
0x080483dc, |           e813ffffff   call 0x80482f4 ; 1 = imp.puts
0x080483e1, |           b800000000   mov eax, 0x0
; Stack size -4

```

[그림 64] Visual 모드에서 xrefs 정보

가장 큰 문제점은 디버깅 모드에서 심볼 정보가 명확하지 않다는 것이었습니다. sym.main의 경우 main의 주소가 _start에서 데이터 참조가 되어야 합니다. 하지만 _start에서는 main의 주소 대신에 frame으로 명시가 되어 있습니다.

```

0x08048327 | 68c4830408 push dword 0x80483c4 ; sym.frame_dummy+0x24
0x0804832c, | e8b3ffffff call 0x80482e4 ; 1 = imp.__libc_start_main
0x08048331 | f4          hlt

```

[그림 65] 디버깅 모드에서 심볼 정보

위 그림에서 __libc_start_main이 호출되기 직전에 push되는 주소는 main()의 주소입니다. 하지만 주석에 frame_dummy로 표기가 되어 있습니다. 하지만 단순히 파일을 열었을 때에는 정상적으로 표기가 됩니다.

```

0x08048327 | 68c4830408 push dword 0x80483c4 ; sym.main
0x0804832c, | e8b3ffffff call 0x80482e4 ; 1 = imp.__libc_start_main
0x08048331 | f4          hlt

```

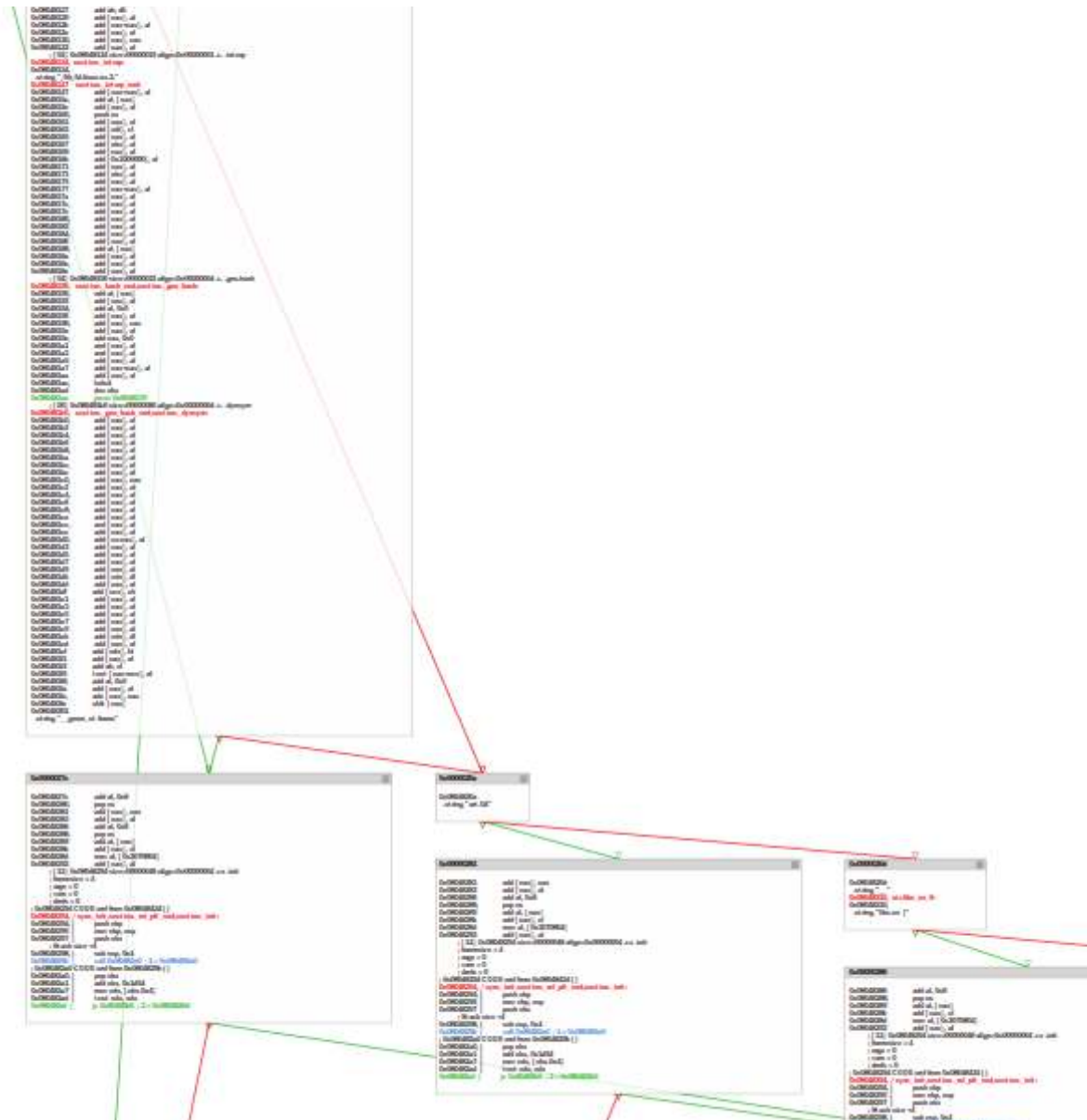
[그림 66] 파일 분석 모드에서 심볼 정보

주석의 정보가 대조적입니다. 버그로 보이는 현상이기 때문에 Fix가 필요할 것으로 보입니다.

3.2. 그래프 역 참조 정보

3.2.1. 그래픽 관련 명령

radare는 기본적으로 커맨드 입력 기반 프로그램이지만 좀 더 수월한 가독성을 위하여 그래픽 인터페이스를 지원합니다. 리눅스의 경우 gtk윈도우를 사용하는데, 코드 블록의 플로 차트나 3.1.에서 살펴본 역 참조 정보들을 보기 쉽게 출력 해 줍니다. radare의 기본적인 그래픽 관련 명령은 g로 시작합니다. 도움말을 보려면 g?를 사용하면 됩니다.



[그림 67] ag 명령을 사용한 코드 블록 차트

위 그림처럼 현재 분석중인 코드 블록을 그래피컬 하게 보여줄 수 있습니다. 만약 IDA에

익숙해 저 있다면 위 그래프가 좀 이해하기 힘들 수 있습니다. IDA와 병행해서 분석한다면 그래프 쪽은 IDA를 사용할 것 같습니다.

3.2.2. 사용자 그래프

3.2.1.에서 보여준 그래프는 radare가 자동으로 생성한 그래프 이지만 사용자가 직접 그래프를 만들고 구성할 수 있도록 지원해 줍니다. 이때 gu명령을 사용할 수 있습니다. 다음은 사용자가 정의한 그래프를 생성할 수 있도록 지원해 주는 명령어입니다.

```
[0xB8012810]> gu?
Usage: gu[dnerv] [args]
gur          user graph reset
gun $$ $b pd  add node
gue $$F $$t   add edge
gud          generate graph in dot format
gudv         view graph in dot format (like agdv does)
guy          visualize user defined graph
[0xB8012810]>
```

[그림 68] gu 명령어들

위 명령어들 중 gur은 사용자 그래프를 초기화 합니다. 즉 가장 처음 실행되어야 하는 명령어 이고 edge와 node를 추가하기 위해서 gue와 gun이 사용됩니다. gue는 에지이므로 두 가지 주소가 인자로 따라와야 합니다. 어디에서 어디로 이을 것인지 명시 해 주어야 하에서 어어야 하에그리고 gun은 node를 추가하는 것디에서 어디로어디 주소를 어떻게 보여줄 것인지로대한 가지 주인자로 따라와야 합니다. 다음은 엔트리 포인트에서 main()을 이어주는 그래프를 예제로 그려 보겠습니다.

1) 그래프 초기화

먼저 그래프를 초기화하기 위해서 gur을 실행합니다.

```
[0xB8012810]> gur
[0xB8012810]>
```

[그림 69] 그래프 초기화

2) 노드 추가

초기화를 하고 난 뒤 노드 2개를 추가해야 합니다. 하나는 엔트리 포인트이고 나머지 하나는 main()입니다. 엔트리 포인트는 radare에서 entrypoint 심볼로 명시되어 있습니다. 저는 이 심볼을 기점으로 해서 30개 opcode를 disassemble 하겠습니다.

```
[0xB8012810]> gun entrypoint 30 pd
[0xB8012810]>
```

[그림 70] entrypoint의 opcode 30개를 disassemble한 node 추가

이제 한 개의 노드가 추가 되어졌습니다. 이제 나머지 노드인 main의 노드를 추가해야 합니다. 엔트리 포인트를 disassemble하였으니 main은 그와 다르게 px를 사용해서 HEX덤프를 떼 보겠습니다.

```
[0xB8012810]> gun sym.main 30 px  
[0xB8012810]> █
```

[그림 71] main의 HEX덤프 결과 노드 추가

이제 엔트리 포인트와 main 두 개의 노드가 등록되어 있습니다.

3) 에지 추가

에지가 없다면 그것은 그래프라고 말 할 수 없습니다. 노드와 더불어 에지가 있어야 그래프이기 때문입니다. 등록한 두 노드를 이어주는 에지는 다음과 같이 등록 합니다.

```
[0xB8012810]> gue entrypoint sym.main  
[0xB8012810]> █
```

[그림 72] 두 노드를 잇는 에지 등록

이제 두 노드는 이어져 있는 상태입니다.

4) 그래프 이미지 생성

생성한 그래프는 graphviz 스크립트로 만들어집니다. 따라서 실제 그래프 이미지는 graphviz를 사용해서 만들어 냅니다. 만약 이 패키지가 없다면 새로 설치 해 주어야 합니다. 먼저 dot 파일을 생성합니다.

```
[0xB8012810]> gud > out.dot  
[0x08048459]> !!dot -Tpng -o graph.png out.dot  
[0x08048459]> █
```

[그림 73] dot 파일 생성

dot이라는 프로그램 역시 graphviz 패키지의 유틸리티입니다. gud를 사용하여 dot 스크립트를 생성한 다음 dot 으로 이미지를 생성합니다. 생성된 이미지를 확인하면 다음과 같습니다.



[그림 74] 생성된 그래프

그래프가 완성 되었습니다. entrypt는 pd를 사용했기 때문에 disassemble된 결과가 출력 되어있고, main은 px를 사용하였기 때문에 HEX 덤프가 떠져 있습니다.

위에서 만든 사용자 정의 그래프를 사용하게 되면 사용자가 중요하다고 생각되는 흐름을 직접 그래프로 만들어 놓아서 프로그램의 흐름을 이해하는 데 큰 도움이 될 수 있을 것으로 보입니다. 또한 phrack에서는 다음과 같은 radare 스크립트를 제공하였습니다.

```

-----8<-----
"(xrefs-to from,f XT @ `Cx~$0[3]`,`)"
"(graph-xrefs-init,gur,)"
"(graph-xrefs,(xrefs-to `?v $$`),gue $$F XT,)"
"(graph-viz,gud > dot,!!dot -Tpng -o graph.png dot,!!gqview graph.png,)"

.(graph-xrefs-init)
.(graph-xrefs) @@= `Cx~[1]`
.(graph-viz)
-----8<-----

```

[표 5] 그래프 생성 스크립트

4. 결론

phrack 66호의 "Manual Binary Mangling with radare" 문서를 모두 읽고 나서 우선은 실망감이 있었습니다. 매우 긴 내용의 본문에 비해서 제목에 부합하는 내용이 적었기 때문입니다. 제가 처음 이 문서에 관심이 간 것은 Code Injection이라는 챕터 때문이었습니다. 작년 6월부터 지금까지 Linux Code Injection이라는 개인적인 프로젝트를 진행하고 있는데, 이 프로젝트에서는 실행중인 프로세스의 메모리 공간에 새로운 코드를 삽입해서 삽입한 코드를 실행하는 것이 주된 내용입니다. 하지만 이 문서에서는 그 와 반대로 바이너리 파일에 추가의 코드를 넣는 것입니다. 얼핏 비슷해 보이지만 상당히 다른 부분이기 때문에 이전부터 머릿속으로 생각만 하고 있었는데 마침 이 문서를 보게 되었습니다.

이 문서에서 생각했던 것 이상의 바이너리 조작의 이론에 대해서 설명하고 있지는 않지만 ELF 형식 파일에서 코드삽입이 가능한 공간에 대한 이야기와 난독 화 부분은 개인적으로 큰 도움이 된 분야였습니다. 그리고 문서에서 바이너리 조작의 도구로 사용된 radare는 올해 초부터 사용하기 시작했던 툴인데, 사실 매력적인 부분이 많지만 불안정한 부분이 많아서 크게 활용하지는 못했었습니다. 하지만 이번 번역 문서를 쓰면서 새로운 radare의 매력을 알게 되어 앞으로 자주 사용할 것 같습니다.

여기까지 문서를 읽어 주셔서 감사합니다.

4. 참고 문헌

- [1] pancake, "manual binary mangling with radare", <http://phrack.org/issueseeeeee.html?issue=66&id=14#article>
- [2] radare.org, "The book(radare manual book)", <http://radare.org/get/radare.pdf.html>