==Phrack Inc.==

Volume 0x0d, Issue 0x42, Phile #0x0E of 0x11

```
|=-----------------------------------------------------------------------=|
|=-----------------=[ manual binary mangling with radare ]=--------------=|
|=-----------------------------------------------------------------------=|
|=--------------------=[ by pancake <at> nopcode.org> ]=-----------------=|
|=-----------------------------------------------------------------------=|
```

1 - Introduction
  1.1 - The framework
  1.2 - First steps
  1.3 - Base conversions
  1.4 - The target

2 - Injecting code in ELF
  2.1 - Resolving register based branches
  2.2 - Resizing data section
  2.3 - Basics on code injection
  2.4 - Mmap trampoline
    2.4.1 - Call trampoline
    2.4.2 - Extending trampolines

3 - Protections and manipulations
  3.1 - Trashing the ELF header
  3.2 - Source level watermarks
  3.3 - Ciphering .data section
  3.4 - Finding differences in binaries
  3.5 - Removing library dependencies
  3.6 - Syscall obfuscation
  3.7 - Replacing library symbols
  3.8 - Checksumming

4 - Playing with code references
  4.1 - Finding xrefs
  4.2 - Blind code references
  4.3 - Graphing xrefs
  4.4 - Randomizing xrefs

5 - Conclusion
6 - Future work

7 - References
8 - Greetings

--[1 – 도입

리버스 엔지니어링은 w32 환경과 연관이 있다. W32 환경에는 많은 유료 소프트웨어가 있으며 평가판의 실행을 제어하거나 지적재산권을 보호하기 위해 바이너리 패킹과 코드 난독화 기법을 사용한 방어기법이 더 늘어나고 있다.

바이러스와 웜도 샌드박스를 탐지하여 백신을 우회할 수 있도록 이러한 방어기법들을 사용한다. 이것이 리버스 엔지니어링이 양날의 검으로 불리는 이유이다.

저 수준 분석을 하는 일반적인 방법은 펌웨어 내용에 대한 정보를 모으거나 키워드를 보여주고 블록을 덤프하기 위한 디스크 이미지의 정보를 제공하는 작은 특정한 유틸리티를 개발하고 사용하는 것이다.

실제로 매우 다양한 종류의 소프트웨어와 하드웨어 플랫폼이 있으며 리버스 엔지니어링 툴들은 이것들을 반영해야 한다.

명백히 리버스 엔지니어링 즉 크래킹 같은 것들이 합법적인 것만은 아니다. Crackme[1]와 같은 해킹 과제들과 이것들을 배우는 것은 바이너리를 다뤄볼 좋은 이유가 된다. 우리들에게는 이러한 모든 것들을 뒤로한 간단한 이유가 있다. 바로 재미이다.

--[1.1 – 구성

이 시점에서 우리는 추상화된 입출력 단계 위의 기능들의 집합을 제공하는 *nix[2]의 기초들을 포함하는 구성을 상상할 수 있다.

이러한 가정에 따라 블록에 기초한 16진수 에디터를 서술하는 것부터 시작한다. 이것은 상처 없이 디스크 장치, 파일, 소켓, 시리얼 또는 프로세스를 열 수 있도록 해준다. 그리고 스크립트로 기능들의 수행을 자동화 할 수 있고 수동으로 대화식 분석을 수행할 수도 있다.

전체적인 구성은 파이프, 파일 또는 API에 의해 쉽게 통합되는 다양한 작은 유틸리티들로 이루어진다.

---

[1] http://en.wikipedia.org/wiki/Crackme
[2] http://en.wikipedia.org/wiki/Unix-like

```
radare: 모든 것에 대한 엔트리포인트(entrypoint)[3]
rahash: 블록 단위의 해싱 유틸리티
radiff: 다중 바이너리 디프(diffing) 알고리즘
rabin:  바이너리로부터 정보를 추출 (ELF,MZ,CLASS,MACHO..)
rasc:   쉘코드 제작 도우미
rasm:   커맨드라인 어셈블러/디스어셈블러
rax:    인라인 다중 기반 컨버터
xrefs:  연관된 코드 참조를 찾는 블라인드 서치
```

Radare의 마법은 자동화의 구현을 쉽게 해주는 명령과 메타데이터의 직교성(ortogonality) - "역자 주 : orthogonality의 오타로 생각된다." 안에 존재한다.

Radare에는 고유의 디버거가 딸려있는데 많은 운영체제에서 작동 되며 소스 코드 없이 바이너리 분석을 쉽게 해주는 많은 흥미로운 명령들이 제공된다.

또 다른 하나의 흥미로운 특징은 고 수준 스크립트 언어인 파이썬, 루비, 루아 등을 지원하는 점이다. 입출력을 추상화하는 원격 프로토콜에 감사하게도 그것은 단지 스크립트 몇 줄을 쓰거나 원격 GDB 프로토콜을 사용하는 것만으로 immunity debugger, IDA, gdb, bochs, vmware 그리고 다른 많은 디버깅 백 엔드들의 사용을 가능하게 해준다.

이 문서에서는 다양한 주제들이 소개될 것이다. 이 하나의 문서에서 모든 것을 설명하는 것은 불가능 하다.

이것에 대해 더 배우고자 한다면 최신 높은 등급의 팁과 아래 주소에서 온라인 문서를 읽어볼 것을 추천한다.

http://www.radare.org [0]

아래 주소에서 또한 pdf 형식의 책 [1]을 읽었으며 html 형식도 사용 가능하다.

http://www.radare.org/get/radare.pdf

--[1.2 – 첫 단계

이 문서를 쉽게 읽을 수 있도록 가장 먼저 redare의 몇몇 기초적인 사용방법을 소개하려고 한다.

Redare를 실행하기 전에 아래와 같이 ~/.radarerc를 설정할 것을 추천한다.

```
e scr.color=1     ; enable color screen
```

[3] http://en.wikipedia.org/wiki/Entry_point

```
e file.id=1       ; identify files when opened
e file.flag=1     ; automatically add bookmarks (flags) to syms, strings..
e file.analyze=1 ; perform basic code analysis
```

어떻게 radare 명령들이 만들어지는지 더 잘 이해할 수 있도록 하는 중요한 요점들은 아래와 같다.

- each command is identified by a single letter
  - subcommands are just concatenated characters
- e command stands for eval and is used to define configuration variables
- comments are defined with a ";" char

이러한 명령 문법은 유연한 공통 언어 기반(CLI)[4]의 입력을 제공한다. 공통 언어 기반에서는 일시적인 검색 수행, 명령문의 많은 반복 수행, 파일 플래그 위에서의 반복, 실제 시스템 파이프의 사용 및 더 많은 것들이 가능하다.

여기에 몇 가지 명령의 예가 있다.

```
pdf @@ sym.     ; run "pdf" (print disassembled function) over all
                ; flags matching "sym."
wx cc @@.file  ; write 0xCC at every offset specified in file
. script        ; interpret file as radare commands
b 128           ; set block size to 128 bytes
b 1M            ; any math expression is valid here (hex, dec, flags, ..)
x @ eip         ; dump "block size" bytes (see b command) at eip
pd | grep call ; disassemble blocksize bytes and pipe to system's grep
pd~call         ; same as previous but using internal grep
pd~call[0]      ; get column 0 (offset) after grepping calls in disasm
3!step          ; run 3 steps (system is handled by the wrapped IO)
!!ls            ; run system command
f* > flags.txt ; dump all flags as radare commands into a file
```

--[1.3 – 기준 변환

rax는 기준들 사이의 값을 변환하는 유틸리티이다. 주어진 16진수 값에서 이는 10진수로 변환된 값을 반환하며 반대로도 작동한다.

이것은 또한 –s 인자를 사용하여 가공하지 않은 흐름(raw streams)이나 16진수 쌍 문자열(hexpair strings)들을 출력 가능한 C언어와 비슷한 문자열로 변환해준다.

---

4 http://ko.wikipedia.org/wiki/%EA%B3%B5%ED%86%B5_%EC%96%B8%EC%96%B4_%EA%B8%B0%EB%B0%98

```
$ 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 0xffffffb3
 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
```

이것으로 우리는 가공하지 않은 파일을 16진수 목록으로 변환할 수 있다.

```
$ cat /bin/true | rax -
```

모든 입력 형식에 대해 반대의 연산도 존재하므로 우리는 아래와 같이 할 수 있다.

```
$ cat /bin/true | rax - | rax -s - > /tmp/true
$ md5sum /bin/true /tmp/true
20c1598b2f8a9dc44c07de2f21bcc5a6 /bin/true
20c1598b2f8a9dc44c07de2f21bcc5a6 /tmp/true
```

rax(1)은 표준 입력으로부터 한 줄씩 읽는 대화식 모드로 사용할 수도 있다.

--[1.4 – 대상

스크립트화 가능한 16진수 에디터는 리버스 엔지니어링 문제를 쉽게 풀 수 있고 이를 해결할 새로운 전망을 제공해준다. 여기에서 이것의 실용적인 사례를 설명하는데 초점을 맞출 것이다.

우리는 x86 [2] GNU/Linux ELF [3] 바이너리를 주요 대상으로 정한다. Radare는 많은 아키텍처 (ppc, arm, mips, java, msil..)와 운영체제 (osx, gnu/ linux, solaris, w32, wine, bsd)를 지원한다. 이 문서에서 설명한 명령과 디버거의 특징들은 gnu/linux-x86 이외의 다른 플랫폼에도 적용 가능하다.

이 문서에서는 독자들이 읽기 쉽도록 가장 널리 알려진 플랫폼 (gnu/linux-x86)에 초점을 맞출 것이다.

우리의 희생량은 본래의 기능을 유지하며 리버스 엔지니어링을 더 어렵게 하기 위해 다양한 조작

단계들을 견뎌낼 것이다.

어셈블리 스니팻(snippets)[5]은 AT&T 문법 [4] 으로 쓰여 있다. 필자가 생각하기에는 이것이 인텔보다 더 똑똑하다. 그러므로 멀티 아키텍처 어셈블러인 GAS를 사용해야만 할 것이다.

Radare2는 라벨과 몇몇 기본적인 어셈블러 명령을 지원하는 많은 아키텍처들을 어셈블 및 디스어셈블 하기 위한 장착가능한 백엔드와 함께 충분한 API를 갖고 있다. 이것은 libr을 사용한 rasm의 도구인 'rasm2'에서 이루어졌다.

그러나 이 문서에서는 더 안정된 radare1을 다루며 완전한 스니팻 대신에 간단한 명령어들을 어셈블 하기 위해서만 'rasm' 명령을 사용할 것이다.

이 문서를 통해서 드러나는 저 수준 분석과 메타데이터 처리는 프로그램의 일부분을 변환하거나 바이너리 블랍에서 정보를 얻는데 도움을 줄 것이다.

일반적으로 radare를 배우기 위한 최적의 프로그램이 'GNU'인 것은 진실이다. 이것은 다룰 코드의 여분을 남겨둔 채로 두 개의 어셈블리 명령어로 완전히 대체될 수 있다. 그러나 이는 이 문서에서 유효한 대상이 아니다. 왜냐하면 우리는 call trampolines, checksumming artifacts, ciphered code, 재할당, syscall obfuscation 등으로 제어 흐름을 숨기기 위한 복잡한 명령들을 찾는데 흥미가 있기 때문이다.

--[2 – ELF안에 코드 삽입

우리 마음속에 가장먼저 떠오르는 질문은 어떻게 우리가 이미 컴파일 된 프로그램에 코드를 더 추가할 수 있는가에 대한 것이다.

실제 패커가 하는 일은 메모리 안에서 조작을 통해 전체 프로그램의 구조를 메모리에 올리고 모든 코드를 재할당 하고 완전히 새로운 실행 파일을 생성하는 일이다.

프로그램 변환은 쉬운 일이 아니다. 이를 위해서 완벽한 분석을 해야 하는데 때때로 자동으로 분석 하기가 불가능 할 수 있다. 따라서 정적, 동적, 수동적인 코드 분석의 결과를 혼합할 필요가 있다.

이러한 접근 대신 우리는 밑에서부터 위로 해 나가는 방식으로 저 수준 코드 구조를 단위로 하여 전체 프로그램을 이해할 필요 없이 변환을 수행할 것이다.

이것은 변환이 프로그램을 어떠한 방식으로도 망가뜨리지 않는 것을 보장해준다. 왜냐하면 로컬영

---

[5] http://en.wikipedia.org/wiki/Snippet_(programming)

역과 내부의 종속성은 확인하기 쉽기 때문이다.

대상 프로그램이 컴파일러에 의해 생성되므로 우리는 몇 가지 가정을 할 수 있다. 예를 들면 코드를 삽입할 공간이 있을 수도 있고 함수들이 순차적으로 정의되어 있을 수도 있다. 또 변수로의 접근이 추적하기 쉬울 수도 있고 함수를 호출하는 방식도 모든 프로그램에 대해 동일할 수 있다.

우리가 처음으로 취할 행동은 사용되지 않고 실행 불가능한 코드로 텍스트 섹션의 모든 부분을 찾아 확인하는 것이다.

실행 불가능한 ('noppable') 코드란 절대로 실행되지 않는 코드 또는 덮어 쓸 수 없는 바이트의 이점을 가진 더 작은 구현에 의해 교체될 수 있는 코드를 의미한다.

컴파일러에 연관된 스터브는 때때로 실행할 수 없거나 교체할 수 없을 수도 있다. 그리고 noc-C native-compiled 프로그램들 (haskell, C++, ...) 에서 더 잘 발견된다.

함수들 사이에서 어떤 코드도 실행될 수 없는 공간들을 함수 패딩 ('function paddings') 라고 한다. 컴파일러는 cpu 상에서 작업들이 쉽도록 정렬된 주소에서 함수의 위치를 최적화 하기 위해 이러한 공간들이 존재한다.

몇몇 다른 컴파일러와 관련된 최적화들은 여러 번 같은 부분의 코드를 인라인화 시킬 것이다. 특정 패턴을 확인하면 우리는 마지막 분기의 단일루프 안으로 스파게티 코드를 끼워 넣을 수 있고 나머지 공간을 임의로 사용할 수 있다.

다음에 오는 단락에서 이러한 상황들을 장황하게 설명할 것이다.

절대로 실행되지 않는 코드:

  우리는 프로그램의 실행을 여러 번 추적하고 노출된 범위를 찾음으로써 이러한 코드를 찾을 수 있다. 이 옵션은 아마도 인간의 상호작용을 필요로 할 것이다.

  정적인 코드 분석을 통해서는 실행 중에 포인터 계산을 따라가거나 테이블을 호출할 수 없다. 그러므로 이 방법은 또한 약간의 인간 상호작용을 필요로 하고 사건 안에서 우리는 패턴을 분석하며 이 작업을 자동화 하는 스크립트를 만들 수 있고 단일 소스 주소로부터 N 코드 외부참조(xrefs)를 더할 수 있다.

  또한 우리는 레지스터 기반 호출/분기 명령에 도달하기 전에 레지스터를 결정하기 위한 몇몇 부분을 대리 실행할 수 있다.

인라인된 코드와 펼쳐진(unrolled) 루프:

프로그래머와 컴파일러는 일반적으로 인라인 (또 다른 하나의 코드 가운데 있는 외부 함수의 중복된 코드)을 선호한다. 왜냐하면 이는 호출(call) 명령의 실행 비용을 줄여주기 때문이다.

루프는 또한 펼쳐져 있고(unrolled) 이 프로시저는 더 길어지지만 더 최적화된 코드이다. 왜냐하면 cpu가 매번 필요치 않은 분기문을 계산할 필요가 없어지기 때문이다.

일반적으로 속도를 올리는 실행들이 있다. 이들은 우리의 코드를 위치시킬 좋은 대상이 된다. 이 과정은 4가지 단계로 진행된다:

- 인라인된 코드나 펼쳐진(unrolled) 루프 (비슷한 블록들에서 반복되는)를 찾기/확인하기
- 루프를 인라인 하지 않도록(uninline) 수정하거나 다시 감기도록(re-roll) 수정하기
- ...
- 획득!

Radare는 이러한 종류의 코드 블록을 찾고 확인하는 것을 돕기 위해 다른 명령을 제공한다. 이런 연관된 명령들은 모든 작업을 자동으로 할 수는 없을 것이다. 이들 명령은 약간의 자동화나 스크립팅 또는 수동의 상호작용을 필요로 한다.

이러한 종류의 코드를 확인하기 위한 기본적인 접근 방법은 동일성이 허락되는 비율의 반복되는 순서를 찾는 것이다. 대부분의 인라인 코드는 작은 함수들로부터 오고 펼쳐진(unrolled) 루프도 아마 그럴 것이다.

이것을 도울 수 있는 두 가지 기본적인 찾기 명령들이 있다:

```
[0xB7F13810]> /?
...
/p len     ; search pattern of length 'len'
/P count ; search patterns of size $$b matching at >N bytes of curblock
...
```

첫 번째 명령 (/p) 는 특정 길이의 반복되는 바이트를 찾아줄 것이다. 두 번째 명령 (/P) 는 현재의 블록과 비교하여 현재 블록 크기의 최소 N 바이트 이상 일치하는 블록을 찾아줄 것이다.

이러한 패턴 검색은 .text 섹션이나 함수 영역 같은 특정 범위에서는 제한 되어야 한다:

검색은 텍스트 섹션에서 제한 되어야 한다:

```
 > e search.from = section._text
 > e search.to = section._text_end
```

모든 검색 명령어들 (/) 은 'search.' 와 확장 변수로 구성된다.

'section.' 의 플래그 이름은 'rabin' 에 의해서 정의된다. -r 플래그로 rabin을 호출하면 radare 명령들처럼 표준 출력으로 바이너리 정보를 덤프 할 것이다.

file.id와 file.flag가 가능해 졌을 때 radare가 시작되는 시점에 이것이 진행된다. 그러나 이 정보는 호출을 함으로써 쉘로부터 다시 가져올 수 있게 된다:

> .!!rabin -rS ${FILE}

이 명령은 점(dot) 명령 '.' 으로 이해 된다. 점은 rardare 명령처럼 다음에 오는 '!' 명령의 출력을 기계어로 해석한다. 아래의 예에서 두 개의 느낌표 (!!)가 존재한다. 왜냐하면 단일 '!' 은 radare의 입출력 서브시스템을 통해서 명령을 실행할 것이기 때문이다. 만약 디버거 명령의 것과 같은 이름을 가진 쉘에서 프로그램을 실행하려고 시도한다면 디버거 계층에서 떨어진다.

이중의 '!!' 는 입출력 서브시스템으로부터 즉시 벗어날 때 사용되고 시스템에서 프로그램을 직접 실행한다.

이 명령은 다른 프로그램들을 실행, 데이터를 처리, 동적으로 생성된 명령들로 radare의 코어 안쪽(core back)을 피딩(feeding)함으로써 몇 가지 스크립팅 설비들을 구체화 하는데 사용될 수 있다.

일단 모든 인라인된 코드가 확인 되면 검색 적중들이 분석되어야 하고 거짓의 포지티브(positives)를 버리는 것과 그러한 모든 블록들을 만드는 것은 호출/리턴 패턴을 사용하거나 나머지 바이트를 무작동 연산으로 채우는(nopping) 서브루틴들로 진행된다. 사용되지 않는 바이트들은 이제 아무 코드로나 채워질 수 있다.

인라인된 코드를 확인하는 또 다른 한가지 방법은 노드를 쪼개거나 (e graph.split=false) 비슷한 기초 블록 또는 반복되는 것을 찾지 않고 기본적인 블록으로 함수 코드 분석을 분담하는 것 이다.

펼쳐진(unrolled) 루프는 코드의 반복되는 순서에 기반하고 있다. 이는 진행하는 상태(증가, 감소 등)에서 반복되는 블록을 따라 바뀌어야만 하는 반복자 변수에 기반을 둔 작은 변화로 나타난다.

언어 결합은 다음과 같은 종류의 작업을 구현하는데 편리하다. 이는 고수준 언어 표현을 하기 위해서 뿐만 아니라 파이썬(python)과 다른 스크립트 언어(perl, ruby, lua)를 위한 radare로 분배된 코드 분석을 하는 고수준 API를 위해서 필요하다.

'/' 명령은 검색을 하는데 사용된다. Radare 명령은 검색이 적중 될 때마다 실행되도록 정의할 수 있다. 또는 다음으로 '@@' 명령을 각각의 반복자에 사용할 수 있다. 반복자는 접두사 'hit0' 다음에 오는 모든 플래그 상에서 사용 가능하다.

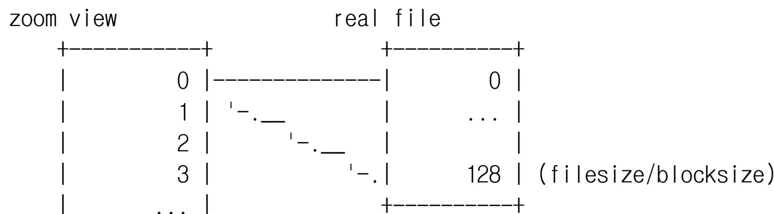
검색과 관련된 명령들의 도움말을 보기 위해 '/?'을 살펴보자. 간단하게 살펴보면 일반 텍스트 또는 16진수에서 검색하는 명령, 바이너리 마스트를 키워드로 적용하는 명령, 한번에 여러 키워드를 던지는 명령, 파일 안에서 키워드를 정의하는 명령, 검색과 새로 바꾸기 등의 명령들이 있다.

'/' 명령을 통하여 접근하기 쉬운 검색 알고리즘 중의 하나는 '/p'로 불린다. 이 명령은 패턴 검색을 수행한다. 이 명령은 분해된 패턴의 최소 크기를 결정하는 숫자 값을 받아들인다.

패턴은 한번 이상 나타나는 연속된 바이트의 시리즈 중 가장 작은 길이로 확인 된다. 결과로 패턴 아이디와 패턴 바이트 그리고 패턴이 발견된 곳의 오프셋 목록이 반환 될 것이다.

이런 종류의 검색 알고리즘은 펌웨어, 디스크 이미지, 압축되지 않은 그림들 같은 거대한 바이너리 파일들을 분석하는데 유용할 수 있다.

큰 파일의 전체적인 개관을 얻기 위해서 'p0' 명령이 사용된다. 이 명령은 16진수 블록크기 바이트로 전체 파일을 보여줄 것이다. 각각의 바이트는 블록의 출력 가능한 문자들의 수를 대표한다. 엔트로피 계산, 그곳에서 확인된 함수들의 수, 실행된 오피코드의 수, 또는 무엇이든 간에 'zoom.byte' 변수의 확장에 놓여있다.



확대한 뷰의 바이트에서 나타나는 실제 파일의 바이트 수는 파일크기와 블록크기 사이의 몫이다.

함수 패딩과 머리말:

컴파일러는 종종 다음에 오는 것을 정렬하기 위해서 함수들을 무작동 명령어를 이용하여 패딩한다. 그것들은 보통 작지만 때때로 실행 흐름을 어지럽게 하는 것을 도울 트램펄린(trampolines)을 저장하기에 충분히 크다.

'트램펄린 확장하기' 장에서는 'call trampoline' 안의 함수 머리말을 숨기는 기술을 설명할 것이다. 'call trampoline'은 각각의 함수의 초기화 코드를 실행하고 다음 명령으로 점프하여 돌아간다.

.data 또는 메모리에서 로드된 호출 테이블을 사용함으로써 리버서(reverser)에게는 더 따르기 어려울 것이다. 왜냐하면 단일 블랍처럼 읽혀지도록 프로그램이 모든 함수 머리말과 꼬리말을 없앨 수 있기 때문이다.

컴파일러 스터브:

함수 머리말과 호출 규약은 컴파일러 사용을 확인하기 위한 워터마크처럼 사용될 수 있다. 그러나 컴파일러 최적화가 가능할 때 생성된 코드는 크게 변형될 것이고 그 머리말들은 아마도 잃게 될 것이다.

몇몇 새로운 버전의 gcc들은 새로운 함수 머리말을 사용한다. 이는 지역 변수 상에서 cpu에 의한 메모리 접근을 만들기 위해 코드를 넣기 전에 스택을 정렬한다. 그러나 우리는 그것을 손수 만든 더 짧은 것으로 대체할 수 있다. 이것은 구체적인 함수 요구 사항을 반영하며 재미로 더 많은 바이트를 자르기도 한다.

엔트리 포인트는 생성자/메인(main)/파괴자를 위한 런처로 작동하는 시스템에 종속적인 로더를 끼워 넣는다. 우리는 생성자와 파괴자가 void인지 점검하는 분석을 할 수 있다.

우리는 단지 메인 포인터에 분기문을 추가함으로 이 모든 코드를 드롭할 수 있다. 이를 통해 C 프로그램을 위한 190 바이트의 공간을 얻게 된다. 아마도 D, C++, haskell 또는 다른 고수준 언어로 쓰여진 프로그램 안에서 더 많은 사용되지 않는 코드를 찾을 수 있을 것이다.

일반적인 gcc의 엔트리 포인트는 아래와 같은 모습이다.

```
0x08049a00, / xor ebp, ebp
0x08049a02  | pop esi
0x08049a03  | mov ecx, esp
0x08049a05  | and esp, 0xf0
0x08049a08, | push eax
0x08049a09  | push esp
0x08049a0a  | push edx
0x08049a0b  | push dword 0x805b630 ; sym.fini
0x08049a10, | push dword 0x805b640 ; sym.init
0x08049a15  | push ecx
0x08049a16  | push esi
0x08049a17  | push dword 0x804f4e0 ; sym.main
0x08049a1c, | call 0x80495b4  ; 1 = imp.__libc_start_main
0x08049a21  ₩ hlt
```

fini/init 포인터 대신에 0들을 채워 넣음으로써 프로그램의 생성자와 파괴자 함수를 무시할 수 있다. 대부분의 프로그램들은 그것들 없이도 아무 문제 없이 수행할 것이다.

두 가지 (sym.init 와 sym.fini) 포인터들 안에 코드를 분석하여 바이트의 수를 결정할 수 있다:

```
[0x080482C0]> af @ sym.__libc_csu_fini~size
size = 4
```

```
[0x080482C0]> af @ sym.__libc_csu_init~size
size = 89
```

추적

실행 추적은 멋진 유틸리티 이다. 이는 작업이나 영역과 같은 부분을 확인하여 바이너리의 뷰를 쉽게 정리할 수 있다. 예를 들면 우리는 추적을 시작하고 모든 메뉴 상의 것과 대상 어플리케이션의 보이는 옵션을 화면에서 클릭함으로써 GUI 코드를 빠르게 확인할 수 있다.

추적을 할 때 가장 일반적인 문제는 !stepall 명령의 낮은 성능이다. 이 명령이 느린 가장 큰 이유는 이미 추적한 코드의 블록을 다시 추적하는 루프 때문이다. 처리의 속도를 높이기 위해서 radare는 “touchtrace” 메소드(!tt 명령 하에서 사용 가능한)를 구현하였는데 이는 MrGadix의 개념을 완성 하였다. (감사!)

이 메소드는 디버거 명령 상의 대상 프로세스 메모리 공간의 교환된 복사본을 유지하고 소프트웨어 중단점 명령들로 이를 교체한다. 인텔에서는 CC(aka int3)를 사용할 것이다.

한번 프로그램이 실행되면 디버거에 의해 다뤄지는 함정 명령 때문에 예외가 발생한다. 그 다음에 디버거가 이 단계에 대한 정보(타임 스탬프, 실행된 횟수, 스텝 카운터와 같은)를 저장하면서 %eip에 있는 명령이 교환되고 유저 공간을 대체 하였는지 검사한다.

명령이 교체되면 디버거는 실행을 계속 하도록 처리를 명령한다. 교환되지 않은 명령 실행이 될 때마다 디버거는 그것을 교체한다.

마지막에는 (끝 중단점을 주는) 디버거는 실행되는 범위를 결정하는 충분한 정보와 함께 버퍼를 갖고 있다. 이 쉬운 메소드를 사용하면 바이너리 분석 속도를 올리면서 모든 reps, 루프(loops), 이미 분석된 코드 블록들을 쉽게 뛰어 넘을 수 있다.

“at” 명령(분석 추적을 대표하는)을 사용하면 통계나 추적에 대한 정보를 얻는 것이 가능하다. 예를 들자면 차례로 나열된 프로그램이나 펼친(unroll) 루프가 있겠다.

우리는 또한 !trace를 사용할 수 있다. 이는 특정 디버그 레벨에 숫자 인수를 전달하고 레지스터 변경, 실행된 명령, 버퍼, 스택의 변화 등을 로그로 남긴다.

이러한 과정 후에 우리는 .text 섹션에 반하는 결과 범위를 뺀다. 그리고 우리가 코드를 삽입하는데 사용할 수 있는 바이트가 얼마인지 얻는다.

만약 우리의 목적대로 사용할 결과 공간이 부족하면 우리의 코드를 넣기 위해 섹션 크기 조절에 직면해야만 할 것이다. 이것은 나중에 설명하도록 하겠다.

Radare는 범위 리스트를 관리하기 위한 "ar"(분석 범위 – analyze ranges) 명령을 제공한다. 이것은 덧셈, 뺄셈, 불 연산을 수행할 수 있다.

```
> ar+ 0x1000 0x1040 ; manual range addition
> ar+ 0x1020 0x1080 ; manual add with overlap
> ari    ; import traces from other places
.....
> .atr* ; import execution traces information
> .CF*  ; import function ranges
> .gr*  ; import code analysis graph ranges
.....
> ar     ; display non overlapped ranges
>        ; show booleaned ranges inside text section
> arb section._text section._text_end
```

우리는 또한 외부 파일이나 프로그램으로부터 범위 정보를 가져올 수 있다. *nix의 더 좋은 특징 중의 하나는 다른 프로그램 언어 안에 단지 하나의 프로그램 'talk'를 만듦으로 파이프를 사용하여 어플리케이션과 통신할 수 있는 능력이다. 따라서 만약 IDA 데이터베이스(또는 IDC)를 파싱하는 펄(perl) 스크립트를 작성했다면 우리는 우리가 필요로 하는 정보를 뽑아낼 수 있고 표준 출력으로 radare 명령들을 출력할 수 있으며 '.' 명령으로 결과를 파싱할 수 있다.

이번에는 ar이 이론적으로 실행되지 않을 바이트의 수와 그것의 범위에 대한 정보를 우리에게 준다.

우리는 이제 몇몇 중단점들을 우리가 어떤 실수도 하지 않았다는 것을 보장하기 위한 장소 상에 위치시킬 수 있다. 자동화 코드 분석과 추적 결과를 손수 개정할 것을 추천한다.

--[2.1 – 레지스터 기반의 분기 해결

'av' 명령은 본래 가상 머신 안쪽의 오피코드를 분석을 하는 서브 명령들을 제공한다.

기본적인 명령들은 다음과 같다: 'av-'는 vm 상태를 재시작, 'avr'은 레지스터 관리, 'ave'는 VM 표현식을 확장, 'avx'는 현재의 탐색으로부터 N 명령들을 실행할 것이다.(eip VM 레지스터를 오프셋 값으로 설정할 것이다.)

내부적으로 가상 머신 엔진은 명령을 메모리를 조작하고 레지스터의 값을 바꾸는데 사용 가능한 문자열로 번역한다.

이런 간단한 개념은 새로운 명령이나 아키텍처의 지원을 쉽게 만들어준다.

여기에 예가 있다. 코드 분석은 레지스터의 번지를 지정하는 호출 명령에 도달한다. 이 코드는 다음과 같이 보인다:

```
/* --- */
$ cat callreg.S
.global main
.data
.long foo
.text

foo:
   ret
main:
   xor %eax, %eax
   mov $foo, %ebx
   add %eax, %ebx
   call *%ebx
   ret

$ gcc callreg.S
$ radare -d ./a.out
( ... )
[0xB7FC6810]> !cont sym.main
Continue until (sym.main) = 0x08048375
[0x08048375]> pd 5 @ sym.main
0x08048375    eip,sym.main:
0x08048375    / xor eax, eax
0x08048377 |    mov ebx, 0x8048374 ; sym.foo
0x0804837c,|    add ebx, eax
0x0804837e |    call ebx
0x08048380, ₩ ret

[0x08048375]> avx 4
Emulating 4 opcodes
MMU: cached
Importing register values
0x08048375       eax ^= eax
     ; eax ^= eax
0x08048377       ebx = 0x8048374 ; sym.foo
     ; ebx = 0x8048374
0x0804837c,      ebx += eax
     ; ebx += eax
```

```
0x0804837e     call ebx
     ; esp=esp-4
     ; [esp]=eip+2
     ;==> [0xbf9be388] = 8048382   ((esp]))
     ; write 8048382 @ 0xbf9be388
     ; eip=ebx

[0x08048375]> avr eip
eip = 0x08048374
```

이 시점에서 우리는 가상 머신 관점에서 'eip' 레지스터가 있는 곳 에서 코드 분석을 계속 할 수 있다.

```
[0x08048375]> .afr @ `avr eip~[2]`
```

--[2.2 – 데이터 섹션의 크기조절

Radare1에서 섹션의 크기를 조절하는 방법은 간단하지 않다. 이 과정은 완전히 수동으로 이뤄지고 복잡하다.

이런 제한을 해결하기 위해서 radate2는 라이브러리의 집합(libr로 명명된)을 제공한다. 이 라이브러리의 집합은 radare 1.x의 모든 기능들을 예전의 하나의 단위로 접근하는 것 대신에 모듈화 디자인을 따르도록 다시 구현한다.

발전의 방향 둘 다 병렬적으로 작동한다.

어디에서 radare2 api (libr)가 작동상태가 되는지 여기에서 살펴보자:

```
$ cat rs.c
#include <stdio.h>
#include <r_bin.h>

int main(int argc, char *argv[])
{
   struct r_bin_t bin;
   if (argc != 4) {
      printf("Usage: %s <ELF file> <section> <size>₩n", argv[0]);
   } else {
      r_bin_init(&bin);
```

```
    if (!r_bin_open(&bin, argv[1], 1, NULL)) {
      fprintf(stderr, "Cannot open '%s'.₩n", argv[1]);
    } else
    if (!r_bin_resize_section(&bin, argv[2], r_num_math(NULL,argv[3]))) {
      fprintf(stderr, "Cannot resize section.₩n");
    } else {
      r_bin_close(&bin);
      return 0;
    }
  }
  return 1;
}
```

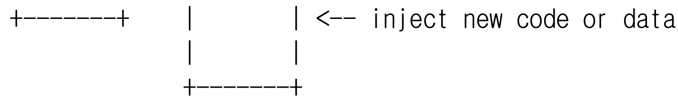
```
$ gcc -I/usr/include/libr -lr_bin rs.c -o rs

$ rabin -S my-target-elf | grep .data
  idx=24 address=0x0805d0c0 offset=0x000150c0 size=00000484 ₩
  align=0x00000020 privileges=-rw- name=.data

$ ./rs my-target-elf .data 0x9400

$ rabin -S my-target-elf | grep .data
  idx=24 address=0x0805d0c0 offset=0x000150c0 size=00009400 ₩
  align=0x00000020 privileges=-rw- name=.data
```

우리의 'rs' 프로그램은 바이너리 파일 (ELF 32/64, PE 32/32+, ...)을 열 것이다. 이것은 .data 섹션의 크기가 변하는 것을 해결해 줄 것이며 elf 정보를 그대로 유지하기 위해 나머지 섹션을 쉬프트 할 것이다.

.data 섹션의 크기를 조정하는 이유는 일반적으로 text 섹션 다음에 링크되기 때문이다. 새로운 주소로의 모든 데이터 접근을 재위치 시키기 위해 프로그램 코드를 다시 분석해야 할 필요가 없을 것이다.

대상 프로그램은 섹션의 크기 조절 이후에도 계속 수행되고 있어야 한다. 우리는 트램펄린을 삽입하기 위해서 프로그램의 어떤 장소가 사용될 수 있는지 이미 알고 있다.

```
  +-------+     +-------+
  | .text |     | .text |
  |       |     |       | <-- inject trampoline
  |-------| ->  |-------|
  | .data |     | .data |
```



다음 챕터에서는 포인터 테이블이나 새로운 코드를 저장하기 위해 데이터 섹션의 크기를 조정하는데 사용될 다중 트램펄린 메소드를 설명할 것이다.

--[2.3 – 코드 삽입의 기초

스스로 변경되는 코드는 매우 좋다. 그러나 종종 위험한 일이기도 하다. SElinux와 같은 패치/툴은 이미 실행 가능한 섹션에 쓰기를 허용하지 않는다.

이 문제를 피하기 위해 우리는 ELF 해더의 .text perms를 쓰기는 가능하지만 실행은 불가능 하도록 변경할 수 있다. 그리고 새로운 실행 가능한 섹션의 복호화 함수를 포함할 수 있다. 복호화 과정이 완료 되었을 때 이것이 text perms를 실행 가능하도록 바꾸어야 하며 그때 제어가 이곳으로 변경된다.

다중의 복잡한 인크립션 레이어를 추가하기 위해서 작업은 더 어려워 질 것이다. 작업을 어렵게 하는 큰 이유는 공간의 재할당 문제이다. 그리고 만약 프로그램이 PIC-ed가 아니면 메모리 상에서 프로그램을 여러 번 복사하고 할당함으로써 전체 어플리케이션을 재배치 하는 작업이 매우 복잡해 질 수 있다.

간단한 구현은 아래와 같은 단계를 포함하고 있다.

- from/to 범위를 정의한다.
- 암호 키와 알고리즘을 정의한다.
- 엔트리포인트 안에 암호를 해독하는 코드를 삽입한다.
- 우리는 반복시키기 전에 mprotect를 사용해야 할 필요가 있을 것이다.
- 정적으로 섹션을 암호문으로 쓴다.

From/to의 주소들은 어셈블리 스니팻에 하드코딩 되어 있거나 데이터 또는 다음 어딘가에서 읽혀진다. 우리는 또한 사용자에게 키를 물어볼 수 있고 패스워드만으로 바이너리 작업을 만들 수 있다.

바이너리에서 인크립션 키가 발견되지 않으면 언패킹 과정은 더 어려워 진다. 그리고 우리는 정확한 키를 찾는 동적인 메소드를 구현해야만 할 것이다. 이는 다음에 설명할 것이다.

범위를 정의하는 또 다른 하나의 방법은 워터마크를 사용하는 것이다. 이는 우리가 삽입한 코드를 메모리 위에서 이러한 마크들을 찾으며 수행되게 해준다.

마지막 단계는 아마 좀 쉬울 것이다. 우리는 이미 키와 알고리즘을 알고 있다. 이제 radare 명령을 사용하여 반대로 다시 써야 할 차례이며 이 범위 안에서 스크립트가 실행될 수 있도록 해준다.

여기에 예제가 있다.

'wo' 명령은 'w' (write) 명령의 서브 명령이다. 'w' 명령은 현재 블록 상에서 산술적인 연산을 제공하는 명령이다.

```
[0x00000000]> wo?
Usage: wo[xrlasmd] [hexpairs]
Example: wox 90    ; xor cur block with 90
Example: woa 02 03 ; add 2, 3 to all bytes of cur block
Supported operations:
  woa  addition        +=
  wos  subtraction     -=
  wom  multiply        *=
  wod  divide          /=
  wox  xor             ^=
  woo  or              |=
  woA  and             &=
  wor  shift right    >>=
  wol  shift left     <<=
```

이것은 16진수쌍의 순환하는 바이트 배열을 인자로 사용한다. 단지 "from" 주소에서 검색 및 블록의 크기를 "to-from" 으로 설정함으로써 'wo' 명령을 입력할 준비가 끝날 것이다.

이 과정을 매크로로 구현하면 코드는 유지 및 관리하기가 더 쉬워질 것이다. 함수형 프로그래밍의 이해가 radare 매크로를 작성하고 코드를 최소 작업을 수행하는 함수로 나누어 그 함수들이 서로 상호작용 할 수 있도록 만드는데 도움이 될 것이다.

매크로는 lisp처럼 정의 되지만 문법은 펄(perl)과 브레인퍽(brainfuck)[6]을 혼합한 것과 매우 흡사하다. 첫 번째 사례에 겁먹을 필요는 없다. 이것은 당신이 생각할 수 있는 것 보다 더 다치게 하지 않으며 결과 코드는 예상한 것 보다 더 우스꽝스러울 것이다. ;)

```
(cipher from to key
  s $0
  b $1-$0
  woa $2
  wox $2)
```

우리는 이것을 한 줄에 입력할 수 있고 (뉴라인을 콤마 ','로 대체 함으로써) 우리의 ~/.radarerc:

[6] http://ko.wikipedia.org/wiki/%EB%B8%8C%EB%A0%88%EC%9D%B8%ED%8D%BD

에 넣을 수 있다.

(cipher from to key,s $0,b $1-$0,woa $2,wox $2,)

이제 우리는 단지 괄호 전에 점을 미리 첨부하고 인자로 이것을 채우는 것 만으로 이 매크로를 언제든지 호출할 수 있다.

```
f from @ 0x1200
f to @ 0x9000
.(cipher from to 2970)
```

코드 삽입을 쉽게 하기 위해서 매크로를 아래와 같이 쓸 수 있다.

```
(inject hookaddr file addr
  wa call $2 @ $0
  wf $1 @ $2)
```

이 매크로를 적절한 값으로 채우면 호출 오피코드를 어셈블 하게 될 것이다. 이 오피코드는 ‘file’ 또는 $1에 저장된 우리의 코드 (addr 또는 $2)를 우리가 삽입할 곳으로 분기한다.

Radare는 Victor Mu~noz가 개발한 확장된 AES 키 검색 알고리즘을 갖고 있다. 이 알고리즘은 이러한 종류의 암호화 알고리즘을 사용하는 프로그램을 리버싱 하는데 도움이 된다. 입출력이 추상화 되었기 때문에 파일, 프로그램 메모리, 램 덤프(ram dumps) 등등에 대해 이 검색을 시작하는 것이 가능하다.

예를 들자면:.

```
$ radare -u /dev/mem
[0x00000000]> /a
...
```

주의: 현재 대부분의 GNU/Linux 배포는 /dev/mem의 단지 처음 1.1M로의 접근을 제한하는 CONFIG_STRICT_DEVMEM 옵션으로 커널 컴파일 된 것이다.

모든 쓰기 명령은 삭제된 내용을 갖고 있는 백업의 링크드 리스트로 저장된다. 이것 때문에 파일 상에 모든 변경 사항에 대해 즉시 되돌리기(undo) 및 다시 실행(redo)이 가능해진다. 이것은 바이너리 상에 다른 변경을 시도할 때 원래의 것을 수동으로 복원할 필요가 없게 해준다.

이러한 변화는 “radiff –r” 명령으로 이후에 비교될 수 있다. 이 명령은 두 파일 간의 델티파이된(deltified) 바이너리 디프를 수행한다. ‘-r’은 첫 번째 바이너리를 두 번째 것으로 변형하는 redare

명령처럼 명령의 결과를 보여준다.

출력은 파일에 파이프 될 수 있고 실행 중에 또는 정적으로 패치하기 위해서 radare로부터 이후에 사용될 수 있다.

예를 들자면:

```
$ cp /bin/true true
$ radare -w true
[0x08048AE0]> wa xor eax,eax;inc eax;xor ebx,ebx;int 0x80 @ entrypoint

[0x08048AE0]> u
03 + 2 00000ae0: 31 ed => 31 c0
02 + 1 00000ae2: 5e => 40
01 + 2 00000ae3: 89 e1 => 31 db
00 + 2 00000ae5: 83 e4 => cd 80

[0x08048AE0]> u* | tee true.patch
wx 31 c0 @ 0x00000ae0
wx 40 @ 0x00000ae2
wx 31 db @ 0x00000ae3
wx cd 80 @ 0x00000ae5
```

'u' 명령은 쓰기 명령을 'undo'(되돌리기) 할 때 사용된다. '*'을 추가함으로써 'u' 명령이 radare 명령처럼 쓰기 변경 사항의 목록을 보여주도록 한다. 그리고 결과를 'tee' 유닉스 프로그램으로 파이프 할 수 있다.

디프를 생성해 보면:

```
$ radiff -r /bin/true true
e file.insertblock=false
e file.insert=false
wx c0 @ 0xae1
wx 40 @ 0xae2
wx 31 @ 0xae3
wx db @ 0xae4
wx cd @ 0xae5
wx 80 @ 0xae6
```

이러한 출력은 표준입력이나 '.' 명령 사용을 통하여 radare로 파이프 될 수 있다. '.' 명령은 파일의

내용이나 어떤 명령의 출력을 기계어로 번역한다.

```
[0x08048AE0]> . radiff.patch
or
[0x08048AE0]> .!!radiff -r /bin/true $FILE
```

$BLOCK, $BYTES, $ENDIAN, $SIZE, $ARCH, $BADDR, … 와 같은 다른 변수들 뿐만 아니라 '$FILE' 환경도 radare 안으로부터 쉘로 보내진다.

Radare 안에서 메모리 주소와 디스크 상의 (on-disk) 주소 사이에 정말로 큰 차이가 없다는 것에 주목하자. 왜냐하면 가상 주소는 io.vaddr와 io.paddr 덕택에 입출력 계층에 의해 대행된다. Io.vaddr와 io.paddr은 섹션 또는 전체 파일을 위한 가상 주소와 물리적 주소를 정의하기 위해 사용된다.

'S' 명령은 오프셋의 배열된 섹션을 위한 가상, 물리적 번지지정 규칙의 좋은 입자를 가진(fine-grained) 세트업(setup)을 구성하기 위해 사용된다.

이것에 지나치게 주의를 기울일 필요는 없다. 왜냐하면 'rabin'이 시작될 때 file.id eval 변수가 참이면 이러한 변수들을 구성할 것이기 때문이다.

--[2.4 – Mmap 트램펄린

.text 섹션의 크기를 조정할 때 직면하는 문제는 .data 섹션이 바뀌는 것이다. 그리고 프로그램은 이곳에 절대적으로 또는 상대적으로 접근하려고 시도할 것이다.

이러한 상태는 우리에게 모든 텍스트 섹션을 다시 형성시키고 나머지 섹션을 적합 시키도록 한다.

문제는 우리가 다시 형성시키기 위한 심층적인 코드 분석을 할 필요가 있다는 것이다. 우리가 단지 정적인 코드 분석만 시도한다면 그것은 아마도 잘못하고 있는 것이다. 이 상태는 종종 복잡한 동적인, 때때로 수동적인, 모든 가능한 포인터를 수정하기 위한 분석을 필요로 한다.

이러한 문제를 건너 뛰기 위해 우리는 단지 .text 섹션 다음에 있는 .data 섹션의 크기를 조정할 수 있고 .data로부터 코드를 로드하고 mmap을 사용하여 메모리에 코드를 넣는 .text 섹션 안에 트램펄린을 쓸 수 있다.

Mmap을 사용하는 결정은 SELinux을 가능하게 하는 것과 함께 일반적이다. 왜냐하면 쓰기를 허용한 실행 가능한 페이지를 얻기 위한 유일한 방법이기 때문이다.

C코드는 아래와 같다:

```
---------------------8<--------------------------
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>

int run_code(const char *buf, int len)
{
   unsigned char *ptr;
   int (*fun)();
   ptr = mmap(NULL, len,
         PROT_EXEC | PROT_READ | PROT_WRITE,
         MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
   if (ptr == NULL)
      return -1;
   fun = (int(*)(void))ptr;
   memcpy(ptr, buf, len);
   mprotect(ptr, len, PROT_READ | PROT_EXEC);
   return fun();
}

int main()
{
   unsigned char trap = 0xcc;
   return run_code(&trap, 1);
}
---------------------8<--------------------------
```

The manual rewrite in assembly become:

```
---------------------8<--------------------------
.global main
.global run_code
.data

retaddr: .long 0
shellcode: .byte 0xcc
hello: .string "Hello World₩n"

.text
```

```
tailcall:
        push %ebp
        mov %esp, %ebp
        push $hello
        call printf
        addl $4, %esp
        pop %ebp
        ret

run_code:
        pop (retaddr)
        /* lcall *0 */
        call tailcall
        push (retaddr)

        /* our snippet */
        push %ebp
        mov %esp, %ebp
        pusha

        push %ebp
        xor %ebp, %ebp      /* 0 */
        mov $-1, %edi       /* -1 */
        mov $0x22, %esi     /* MAP_ANONYMOUS | MAP_PRIVATE */
        mov $7, %edx        /* PROT_EXEC | PROT_READ | PROT_WRITE */
        mov $4096, %ecx     /* len = 4096 */
        xor %ebx, %ebx      /* 0 */
        mov $192, %eax      /* mmap2 */
        int $0x80
        pop %ebp

        mov %eax, %edi            /* dest pointer */
        mov 0x8(%ebp), %esi   /* get buf   (arg0) */
        mov $128, %ecx
        cld
        rep movsb
        mov %eax, %edi

        mov $5, %edx        /* R-X */
        mov $4096, %ecx     /* len */
```

```
        mov %edi, %ebx      /* addr */
        mov $125, %eax      /* mprotect */
        int $0x80

        call *%edi
        popa
        pop %ebp
        ret

main:
        push $shellcode
        call run_code
        add $4, %esp
        ret
---------------------8<--------------------------
```

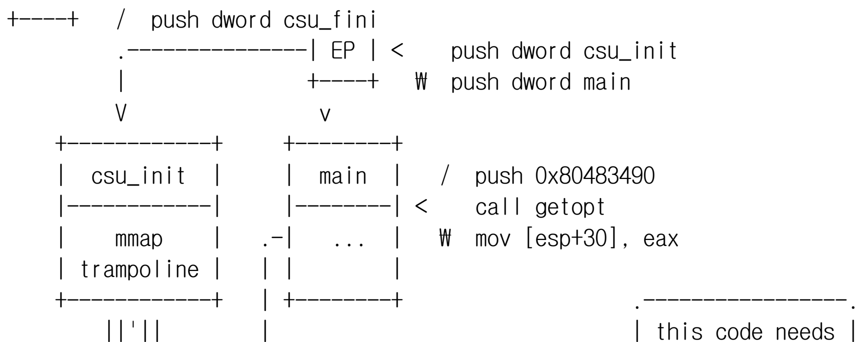
이 작은 프로그램을 실행하는 것은 mmap 되어진 공간에서 실행되는 함정 명령상의 결과로서 일어날 것이다. 나는 코드를 더 작게 만드는 리턴 값 상의 점검을 하지 않았다.

```
$ gcc rc.S
$ ./a.out
Hello World
Trace/breakpoint trap
```

이 코드의 바이트 사이즈는 쉘 안의 이 줄로 측정 가능하다:

```
$ echo $((`rabin -o d/s a.out |grep run_code|cut -d ' ' -f 2 |wc -c`/2))
76
```

아래 그래프는 필요한 패치의 레이아웃을 설명해준다.



```
|    |- - - - |- - - - - - - - - - - - - - - . <  the mmap tramp  |
:          __|___       +-----------+    .  | to be executed  |
.         / hook ₩ ---> | mmap code |    .  `-----------------'
.         ₩getopt/      +-----------+    .
 - - - - - - - - - - - - - - -|- - - - - - '
                              |
          +--------+          / fall to the getopt flow
          | getopt | <-----'
          +--------+
              |
            (ret)
```

삽입 처리는 다음과 같은 단계를 포함하고 있다.

* 우리의 코드를 위한 충분한 공간을 얻기 위해서 .data 섹션의 크기를 조정한다.

```
f newsize @ section._data_end - section._data + 4096
!rabin2 -o r/.data/`?v newsize` $FILE
q!
```

* 크기가 조정되지 않은 .data 섹션의 끝부분에 코드를 쓴다.

```
!!gcc our-code.c
"!!rabin a.out -o d/s a.out | grep ^main | cut -d ' ' -f 2 > inj.bytes"
wF inj.bytes @ section._data_end
```

* 엔트리포인트에(2nd push dword) 있는 __libc_csu_init 포인터를 NULL로 만든다.

```
e asm.profile=simple
s entrypoint
wa push 0 @ `pd 20~push dword[0]#1`
```

* 심볼 __libc_csu_init에 있는 mmap 로더를 삽입한다.

```
# generate mmap.bytes with the mmap trampoline
!!gcc mmap-tramp.S
"!!rabin a.out -o d/s a.out | grep ^main | cut -d ' ' -f 2 > mmap.bytes"
# write hexpair bytes contained in mmap.bytes file at csu_init
wF mmap.bytes @ sym.__libc_csu_init
```

* 우리의 삽입된 코드의 실행에 다리를 놓고 원래 호출의 작업흐름을 지속하게 하기 위해서 호출 명령을 낚아 챈다.

  - 낚아 채어질 대상 심볼에 대한 호출 주소를 결정한다.

예를 들어서 ‘ping’ 프로그램을 갖고 있으면 이 스크립트는 PLT 엔트리를 가져오는 getopt 에 대한 xref을 결정할 것이다.

```
s entrypoint
# seek to the third push dword in the entry point (main)
s `pd 20~push dword[3]#2`

# Analyze function: the '.' will interpret the output of the command
# 'af*' that prints radare commands registering xrefs, variable
# accesses, stack frame changes, etc.
.af*

# generate a file named 'xrefs' containing the code references
# to the getopt symbol.
Cx~`?v imp.getopt`[1] > xrefs

# create an eNumerate flag at every offset specified in 'xrefs' file
fN calladdr @@.xrefs
```

- 스택을 파괴하지 않고 실행에 다리를 놓기 위한 프록시 훅(hook)을 만든다.

  이것은 다음에 오는 줄들에서 설명되어 있다..

- 우리의 프록시 함수를 위해서 훅(hook)의 호출을 변경한다.

```
# write assembly code 'call <addr>'
# quotes will replace the inner string with the result of the
# command ?v which prints the hexadecimal value of the given expr.
wa call `?v dstaddr`
```

- 획득

여기에 패치 된 호출을 위한 트램펄린의 기능을 설명하는 어셈블리 코드의 예가 있다. 코드의 이러한 스니팻은 mmap 트램펄린 이후에 삽입될 수 있다.

```
----------8<-----------
.global main

trampoline:
  push $hookstr
```

```
   call puts
   add $4, %esp
   jmp puts

main:
   push $hello
   call trampoline
   add $4, %esp
   ret

.data

hello:
   .string "hello world"
hookstr:
   .string "hook!"
----------8<-----------
```

이전의 코드는 바이너리에 곧바로 삽입될 수 없다. 주요한 개념적인 이유중의 하나는 트램펄린의 목적지 주소가 실행 중에 정의될 것이라는 점이다. 데이터 섹션의 mmap된 영역이 실행 가능 허용으로 준비될 때 목적지 주소는 데이터 섹션의 어떠한 부분에 저장 되어야만 한다.

--[2.4.1 – 트램펄린 호출

일단 결함이 확인되면 우리는 이제 다른 방법으로 제어 흐름, 심볼 접근, 함수 호출, 인라인된 시스템 호출(syscalls)을 어지럽게 하도록 프로그램의 몇 부분을 변경할 수 있다.

프로그램 코드 블락을 탐색하기 위해서 우리는 간단한 것에서부터 복잡한 스크립트 코드 분석을 수행하는 엔진을 실행하도록 반복자(iterator) 또는 중첩된 반복자를 사용할 수 있다. 그리고 명령을 변경하거나 코드를 이동함으로써 프로그램을 검색 또는 조작을 할 수 있다.

함수 호출을 위한 호출 테이블 트램펄린을 추가하여 프로그램의 제어 흐름이 어지럽게 되도록 할 수 있다. 우리는 호출을 패치해야 할 것이고 데이터 섹션 안의 테이블에 있는 점프 주소를 저장해야 할 것이다.

스택의 eip을 낮추고 호출 대신 점프를 사용하기 위한 다른 트램펄린을 배치시켜서 긴 점프를 위한 같은 프로그래밍 기법을 수행하도록 할 수 있다.

Radare의 반복자는 각각의 @@ 마크 다음에 매크로 호출을 덧붙이는 방식으로 구현 가능하다.

예를 들면: "x @ 0x200"
0x200으로 임시 검색을 수행할 것이고 “x” 명령을 실행할 것이다.

사용법: "x @@ .(iterate-functions)"
매크로에 의해 리턴되는 모든 주소에 반복자 매크로를 실행할 것이다.

매크로에서 값을 리턴받기 위해서 우리는 null 매크로 본체를 사용할 수 있고 결과 오프셋이 덧붙일 수 있다. 아무 값도 주어지지 않으면 null이 리턴되고 반복자는 멈출 것이다.

여기에 함수 반복자(function-iterator)에 대한 구현이 있다:

```
"(iterate-functions,()CF~#$@,)"
```

우리는 또한 검색이 적중 하였을 때 명령을 실행하기 위한 cmd.hit eval 변수로 검색 엔진을 사용할 수 있다. 명백히 명령은 매크로 또는 어떤 언어의 스크립트 파일이 될 수 있다.

```
e cmd.hit=ar- $$ $$+5
/x xx yy zz
```

그러나 우리는 단지 명확하게 잘못된 적중을 제거하고 각각의 명령을 실행함으로써 검색의 처리 이후에 이것을 할 수 있다.

```
# configure and perform search
e search.from=section._text
e search.to=section._text_end
e cmd.search=
/x xx yy zz

# filtering hit results
fs search      ; enter into the search flagspace
f              ; list flags in search space
f -hit0_3      ; remove hit #3
# write on every filtered hit
wx 909090 @@ hit*
```

플래그스페이스(flagspaces)는 플래그(flag)들의 그룹이다. 이것들 중의 일부는 문자열, 심볼, 섹션 등등을 확인하는 중에 rabin에 의해 자동적으로 만들어진다. 그리고 이외의 것들은 ‘regs’ (registers) 또는 ‘search’ (search results)와 같은 명령에 의해 실행 시간 중에 업데이트 된다.

‘fs’ 명령을 사용하여 다른 하나의 플래그스페이스를 만들거나 이것으로 엇바꿀 수 있다. 플래그스페이스가 선택되었을 때 ‘f’ 명령은 단지 현재 플래그스페이스에 있는 플래그를 보여줄 것이다. ‘fs *’ 명령은 같은 시점(글로벌 관점)에서 모든 플래그스페이스를 가능하게 하는데 사용된다. fs?로 확

장된 도움말을 볼 수 있다.

내부 메타데이터 데이터베이스는 C 명령으로 관리될 수 있고 코드와 데이터 xrefs, 주석, 함수 정의, 타입, 디스어셈블러 엔진에 의해 사용되는 구조체 정의와 데이터 변환, 코드 분석 모듈과 유저에 의해 정의된 스크립트들에 대한 정보를 저장한다.

데이터와 코드 xrefs는 리버스 엔지니어링 시에 매우 유용한 참조 포인트들이다. 리버서를 실행 중에 사용하도록 하거나 문자열이나 버퍼가 사용될 때 결정하기 위한 환경을 대신 실행하면서 코드 분석 엔진을 속이기 위해 그것들 모두를 가능한 많이 어지럽게 하는 것이 흥미롭다.

Radare 안의 함수를 분석하기 위해서 차라리 'V' 명령을 사용해서 시각화 모드로 진행한다. 그리고 이때 "d" 와 "f" 키를 누른다.

이 행동은 현재 검색으로부터 또는 커서의 포인트가 있는 곳에서 코드 분석을 만들 것이다(커서 모드라면 시각화 모드에서 소문자 "c" 키로 전환 가능). 이 코드 분석은 인자, 로컬과 글로벌 변수 접근, 특정 포인터를 사용하는 프로그램의 포인트를 결정하기 위해 이후에 사용될 수 있는 xref 정보로 데이터베이스를 채우는 것과 같은 복합적인 것들을 정의할 것이다.

~/.radarerc 안의 거의 없는 eval 변수를 정의함으로써 이러한 기초적인 코드 분석의 대부분이 기본으로 수행될 것이다.

```
$ cat ~/.radarerc
e file.id=true    ; identify file type and set arch/baddr...
e file.analyze=true ; perform basic code analysis at startup
e file.flag=true  ; add basic flags
e scr.color=true  ; use color screen
```

참고: radare로의 '-n' 매개 변수를 통과함으로써 시작 과정은 .radarerc의 해석을 건너 뛸 것이다.

이러한 모든 정보는 프로젝트 파일을 사용하여 저장, 조작, 복원이 가능하다 (시작할 때 –P 프로그램 플래그를 사용하거나 'P' 명령으로 실행 중에 이 특성을 활성화 하여).

이 괄호 다음에 우리는 리버서 수명을 더 우습게 만들기 위해 xrefs을 어지럽게 하는 것을 지속한다.

이것을 하기 위한 한자기 간단한 방법은 초기화 코드를 만들고 임의로 mmap된 주소에서 실행하기 위한 프로그램에서 요구되는 몇 가지 포인터를 복사하고 이러한 접근을 순차적인 직쏘 형태로 변환하기 위한 방법을 정의하는 것이다. 이것은 각각의 포인터를 모드(mod)에 의해 접근되는 시프트된 배열처럼 몇 가지 규칙에 기반한 "random"으로 바꾸는 것으로써 달성 될 수 있다.

```
# calltable initializer
```

```
(ct-init tramp ctable ctable_end
  f trampoline @ $0
  f calltable @ $1
  f calltable_ptr @ calltable
  wf trampoline.bin @ trampoline
  b $2-$1
  wb 00 ; fill calltable with nops
  )

; calltable adder
(ct-add
  ; check if addr is a long call
  ? [1:$$]==0xeb
  ?!()
  ; check if already patched
  ? [4:$$+1]==trampoline
  ??()
  wv $$+5 @ calltable_ptr
  yt 4 calltable_ptr+4 @ $$+1
  f calltable_ptr @ calltable_ptr+8
  wv trampoline @ $$+1
  )
```

우리는 또한 .text 안의 모든 긴 호출 명령 상에서 자동적으로 반복되는 세번째 도우미 매크로를 만들 수 있다. 이것 위에서 ct-add가 실행된다.

```
(ct-run from to tramp ctable ctable_end
  .(ct-init tramp ctable ctable_end)
  s from
  loop:
    ? [1:$$]==0xeb
    ?? .(ct-add)
    s +$$$
  ? $$ < to
  ??.loop:
  )
```

혹은 우리는 또한 함수 상에서 수행하기 위한 더 좋은 입자를 가진(fine-grained) 매크로를 작성할 수 있다.

```
e asm.profile=simple
pD~call[0] > jmps
.(ct-init 0x8048000 0x8048200)
.(ct-add)@@.jmps
```

Trampoline.bin 파일은 아래와 같을 것이다.

```
----------------------------------------------------------------
/*
 * Calltable trampoline example // --pancake
 * ----------------------------------------------------
 * $ gcc calltrampoline.S -DCTT=0x8048000 -DCTT_SZ 100
 */

#ifndef CTT_SZ
#define CTT_SZ 100
#endif
#ifndef CTT
#define CTT call_trampoline_table
#endif

.text
.global call_trampoline
.global main

main:
 /* */
 call ct_init
 call ct_test
 /* */
 pushl $hello
 call puts
 add $4, %esp
 ret

call_trampoline:
 pop %esi
 leal CTT, %edi
 movl %esi, 0(%edi)          /* store ret addr in ctable[0] */
 0:
```

```
 add $8, %edi
 cmpl 0(%edi), %esi         /* check ret addr against ctable*/
 jnz 0b
 movl 4(%edi), %edi         /* get real pointer */
 call *%edi                 /* call real pointer */
 jmp *CTT                   /* back to caller */

/* initialize calltable */
ct_init:
 leal CTT+8, %edi
 movl $retaddr, 0(%edi)
 movl $puts, 4(%edi)
 ret

/* hello world using the calltable */
ct_test:
 push $hello
 call call_trampoline
retaddr:
 add $4, %esp
 ret

.data
hello:
 .string "Hello World"

call_trampoline_table:
 .fill 8*(CTT_SZ+1), 1, 0x00
-----------------------------------------------------------------
```

이러한 트램펄린의 구현이 쓰레드 세이프(thread safe) 하지 않다는 것을 눈여겨보자. 만약 두개의 쓰레드가 리턴 주소를 임시로 저장하기 위해서 동시에 같은 트램펄린 호출 테이블을 사용하면 응용프로그램은 기능이 멈출 것이다.

이런 문제를 고치기 위해서 '시스템호출 어지럽게 하기(syscall obfuscation)' 장에서 설명된 %gs 세그먼트를 사용해야 한다. 그러나 복잡성을 피하기 위해서 여기에서는 어떤 쓰레드 지원도 포함하지 않을 것이다.

바이너리에 쉘코드를 삽입하기 위해서 우리는 호출테이블에 대한 .data 세그먼트 안에 약간의 공간을 만들어야 한다. 이것은 gcc –D로 할 수 있다:

```
[0x8048400]> !!gcc calltrampoline.S -DCTT=0x8048000 -DCTT_SZ=100
> !! rabin -o d/s a.out|grep call_trampoline|cut -d : -f 2 > tramp.hex
> wF tramp.hex @ trampoline_addr
```

--[2.4.2 – 트램펄린 확장

우리는 대상 프로그램으로부터 실제 코드의 일부를 취할 수 있고 무작위 키로 암호화된 .data 섹션 안으로 이것을 이동시킬 수 있다. 그 다음에 초기화 함수를 실행하기 이전에 이 함수를 실행하는 호출 트램펄린을 확장할 수 있다.

우리의 호출 트램펄린 확장은 세가지 요점을 포함할 것이다.

- Function prelude obfuscation – Function mmaping (loading code from data)
- Unciphering and checksumming

마지막 포인트 주소를 호출한 이후 동시에 우리는 함수를 다시 암호화 할 수 있다. 이것은 방어자에게 좋을 것이다. 왜냐하면 리버서(reverser)가 코드 안으로 수동으로 나아가도록 강요될 것이기 때문이다. 왜냐하면 이 코드에서 사용 가능한 소프트웨어 브레이크포인트가 없기 때문이다. 왜냐하면 그것들은 다시 암호화 될 것이고 체크섬 하는 것은 실패할 것이기 때문이다.

이것을 하기 위해서 호출 트램펄린 테이블을 확장하거나 목적 코드 (암호화가 되었든 안되었든)의 상태와 선택적인 체크섬을 자세히 설명하는 호출 트램펄린 테이블 안의 단지 다른 하나의 필드를 추가해야 한다.

이 문서는 바이너리를 다룰 때 가능한 일과 가이드라인을 설명하는데 초점을 맞추고 있다. 그러나 진보된 기술을 설명하여 이 문서가 너무 커지는 것을 피하기 위해 이러한 종류의 확장은 이행하지 않을 것이다.

'p' 명령은 다중 형식의 바이트를 출력하는데 사용된다. 더 강력한 출력 형식은 'pm' 이다. 이 것은 형식 문자열(format-string)을 사용하는 메모리 내용을 기술하도록 허용한다. 이 형식 문자열은 필드 각각의 이름과 문자열 같은 것들이다.

형식 문자열이 숫자 값으로 시작되면 구조체의 배열로 여겨진다. 'am' 명령은 radare의 실행을 따라 구조체 서명의 재사용을 쉽게 하기 위한 'pm' 명령으로 명명된 리스트를 관리한다.

우리의 트램펄린 호출 테이블의 엔트리(entry) 형식을 기술하며 이 명령에 대한 소개를 시작할 것이다:

```
[0x08049AD0]> pm XX
0x00001ad0 = 0x895eed31
0x00001ad4 = 0xf0e483e1
```

이제 필드의 이름을 추가한다:

```
[0x08049AD0]> pm XX from to
      from : 0x00001ad0 = 0x895eed31
        to : 0x00001ad4 = 0xf0e483e1
```

그리고 이제 읽기 가능한 구조체의 배열을 출력한다:

```
[0x08049AD0]> pm 3XX from to
0x00001ad0 [0] {
      from : 0x00001ad0 = 0x895eed31
        to : 0x00001ad4 = 0xf0e483e1
}
0x00001ad8 [1] {
      from : 0x00001ad8 = 0x68525450
        to : 0x00001adc = 0x0805b6a0
}
0x00001ae0 [2] {
      from : 0x00001ae0 = 0x05b6b068
        to : 0x00001ae4 = 0x68565108
}
```

'데이터 분석(analyze data)'을 대표하는 'ad'라 명명된 다른 하나의 명령이 있다. 이것은 자동으로 인식된 포인터(구성된 엔디언(endianness)을 따르는)를 찾는 메모리의 내용(현재의 검색부터 시작하여), 문자열, 문자열 포인터, 그리고 더 많은 것들을 분석한다.

이 명령이 어떻게 작동하는지 보는 가장 쉬운 방법은 프로그램을 디버거 모드(f.ex: radare –d ls)로 실행하고 'ad@esp'을 실행하는 것이다.

```
# register a 'pm' in the 'am' command
  [0x08048000]> am foo 3XX from to

  # grep for the rows matching 'from'
  [0x08048000]> am foo~from
          from : 0x00001ad0 = 0x895eed31
          from : 0x00001ad8 = 0x68525450
          from : 0x00001ae0 = 0x05b6b068

  # grep for the 4th column
  [0x08048000]> am foo~from[4]
```

```
0x895eed31
0x68525450
0x05b6b068

# grep for the first row
[0x08048000]> am foo~from[4]#0
0x895eed31
```

--[3 – 보안과 조작

이번 장에서는 대상 바이너리에 대한 리버스 엔지니어링을 더 복잡하게 만드는 보안의 몇 단계를 더하는 바이너리 상에서 행해질 수 있는 다른 조작을 살펴볼 것이다.

--[3.1 – ELF 헤더 지우기

ELF 헤더를 파괴하는 것은 리버서의 삶을 약간 더 힘들게 만들기 위한 패커(packer)의 또 다른 하나의 가능한 대상이다.

단지 ELF 헤더의 한 바이트를 변경하는 것은 gdb, ltrace, objdump 등과 같은 도구들에 문제를 야기한다. 이는 그것들이 섹션, 심볼, 라이브러리 종속성 등에 대한 정보를 얻기 위한 섹션 헤더에 의존하고 있기 때문이다. 그리고 이런 도구들이 그것들을 분석할 수 없으면 오류와 함께 단지 멈추게 된다. 이것을 쓰는 시점에서는 오직 IDA와 radare만이 이 문제를 우회할 수 있었다.

이것은 아래와 같은 방식으로 쉽게 할 수 있다:

```
$ echo wx 99 @ 0x21 | radare -nw a.out
```

리버서는 ELF 헤더 안의 'e_sh_off' dword을 다시 재건하기 위해서 radare 소스 안의 fix-shoff.rs (스크립트 안에 포함된) 디렉토리를 사용할 수 있다.

```
$ cat scripts/fix-shoff.rs
; radare script to fix elf.shoff
(fix-shoff
  s 0            ; seek to offset 0
  s/ .text       ; seek to first occurrence of ".text" string
 loop:
  s/x 00         ; go next word in array of strings
  ? [1:$$+1]     ; check if this is the last one
  ?!.loop:       ; loop if buf[1] is == 0
  s +4-$$%4      ; align seek pointer to 4
```

```
    f nsht         ; store current seek as 'nsht'
    wv nsht @ 0x20 ; write the new section header table at 0x20 (elf.sh_off)
  )
```

스크립트는 아래와 같은 방법으로 사용된다:

```
  $ echo ".(fix-shoff) && q" | radare -i scripts/fix-shoff.rs -nw "target-elf"
```

이 스크립트는 파괴된 shoff을 포함한 어떤 일반적인 bin 상에서도 수행된다. 이것은 손상된 섹션 헤더('sstrip' f.ex. 이후에)를 포함한 바이너리에서는 실행되지 않는다.

이것은 GCC가 문자열 테이블 인덱스 다음에 곧바로 섹션 헤더 테이블을 저장하기 때문에 가능하다. 그리고 .text 섹션 이름이 어떤 표준 바이너리 파일상에 나타날 것이기 때문에 이 섹션은 쉽게 식별된다.

이 패치를 우회하는 다른 방법을 생각하는 것은 어렵지 않다. Shoff을 0으로 설정한 다음에 섹션의 나머지 모두를 분해함으로써(stripping) 잘 작동할 것이다. 그리고 리버서는 섹션 헤더를 완전히 재생할 것이다. Elf-kickers 패키지로부터 sstrip(1)(이중의 's'는 타입(typo)이 아니다)을 사용하여 이것을 할 수 있다. 이 방식으로 우리는 프로그램을 실행하기 위해서 모든 불필요한 섹션을 제거할 수 있다.

이 헤더 버그의 참 좋은 점은 e_shoff 값이 마치 두 번째 인자처럼 lseek(2)으로 곧바로 전달되는 것이다. 그리고 또한 우리는 단지 잘못된 주소를 주는 것 대신에 다른 에러를 얻기 위해서 0 또는 -1과 같은 값으로 시도해 볼 수 있다. 이 방법으로 다른 방법에서의 잘못된 ELF 파서를 우회하는 것이 가능하다.

SEGMENT_COMMAND 헤더 안에서 섹션의 번호를 -1로 설정하는 것과 같은 GNU 소프트웨어로부터 MACH-O 파서에서 이와 비슷한 버그가 발견된다. 이러한 종류의 버그는 커널을 적중하지 않는다. 왜냐하면 그것들은 주로 메모리에 맵핑하기 위한 프로그램 헤더로부터 최소한의 입력만 사용하기 때문이다. 그리고 유저 공간의 동적인 링커가 나머지를 수행한다. 그러나 디버거와 파일 분석기는 헤더로부터 충돌된 정보를 추출하려고 시도할 때 종종 실패한다.

--[3.2 – 소스 수준 워터마크

만약 우리가 대상 응용 프로그램의 소스를 갖고 있다면 이것을 가지고 놀 더 많은 길이 있을 것이다. 예를 들어 휘발성 asm 인라인 매크로 같은 정의를 사용해면 개발자는 바이너리 패커가 사용 가능한 패딩(paddings)이나 마크를 추가할 수 있다.

몇몇 상업적인 패커는 SDK을 제공한다. 이 SDK는 패커의 사용을 더 쉽게 만드는 워터마크와 함수 도우미를 제공하는 인클루드 파일의 집합일 뿐 그 이상은 아니다. 그리고 그들의 코드를 보호

하기 위한 더 많은 도구들을 개발자에게 준다.

```
$ cat watermark.h
#define WATERMARK __asm__ __volatile__ (".byte 3,1,3,3,7,3,1,3,3,7");
```

우리는 이후에 radare로 후처리(post-processing) 하기 위해 생성된 바이너리를 워터마크 하는 CPP 매크로를 사용할 것이다. 여러 가지 목적을 위한 다양한 다른 워터마크를 상술할 수 있게 되었다. 우리는 생성된 바이너리 안에 이미 워터마크가 없다는 것을 명심해야 한다.

심지어 .text 섹션 안에 시끄러운(noisy) 바이트로 우리의 프로그램을 덧붙이기 위해 더 큰 워터마크를 가질 수 있다. 따라서 이후에 우리는 이러한 워터마크를 어셈블리 스니팻으로 교체할 수 있다.

Radare는 워터마크를 찾고 코드를 삽입하는 수동의 과정을 다음과 같은 명령을 사용해서 쉽게 만들어줄 수 있다. 이것은 줄마다 하나의 오프셋을 포함하는 'water.list'라 불리는 파일을 생성할 것이다.

```
[0x00000000]> /x 03010303070301030307
[0x00000000]> f~hit[0] > water.list
[0x00000000]> !!cat water.list
0x102
0x13c
0x1a2
0x219
```

만약 우리가 파일 안에 명시된 모든 오프셋에서 명령을 실행하기 원한다면:

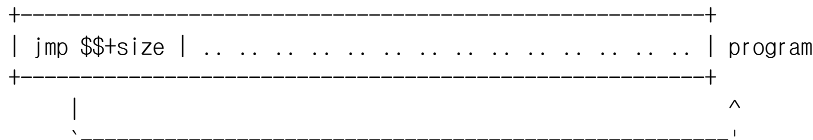
```
[0x00000000]> .(fill-water-marks) @@. water.list
```

모든 워터마크 채우기(fill-water-mark) 매크로 안에서 '적중(hit)' 플래그 번호 (우리는 이때 한 개 이상의 키워드를 찾을 수 있다)에 의존하거나 그곳의 바이트를 검사하여 워터마크 타입을 분석할 수 있다. 그리고 적절한 동작을 수행할 수 있다.

만약 저러한 워터마크가 고정된 크기를 갖는다면 (또는 크기가 워터마크 안으로 묻혀진다면) 이때 우리는 패딩을 뛰어넘기 위한 곳에 분기 명령을 쓰는 radare 스크립트를 작성할 수 있다. (이것은 워터마크의 더미 바이트를 실행하지 않을 것이기 때문에 프로그램의 충돌을 피할 수 있다).

여기에 아스키 아트로 만든 설명이 있다:

```
|<-- watermark ---------------------------------------->|
             |<-- unused bytes                        -->|
```



이 시점에서 우리는 코드에 넣을 약간 더 많은 통제된 결함들을 갖는다.

우리의 가변형(variable-sized) 워터마크는 이런 식으로 정의될 것이다:

```
.long 0,1,2,3,4,5,6,7,4*20,9,10,11,12,13,14,15,16,17,18,19,20
```

그리고 워터마크는 매크로를 채울것이다:

```
(fill-water-marks,wa jmp $${io.vaddr}+$$+[$$+8],)

# run the macro to patch the watermarks with jumps
.(fill-water-marks) @@. water.list
```

Io.vaddr은 메모리 안에 있는 프로그램의 가상 기준 주소를 구체화할 것이다.

--[3.3 – .data 섹션 암호화

정말 일반적인 바이너리 보호는 문자열을 숨기고 정적인 코드 분석을 더 어렵게 만드는 프로그램의 데이터 섹션을 암호화 하는 것을 포함한다.

여기에 소개된 이 기술은 데이터 섹션을 보호하는데 중점을 두고 있다. 이 섹션은 일반적으로 프로그램 문자열을 포함하고 있지 않다 (왜냐하면 그것들은 보통 rodate 섹션 안에서 정적으로 정의되기 때문이다). 그러나 ELF 로더의 몇몇 특징과 어떻게 프로그램이 rw 섹션에서 작동하는지를 이해하는데 도움이 될 것이다.

PT_LOAD 프로그램의 헤더 타입은 어디에서, 어느 것이, 어떻게 바이너리의 부분이 메모리 안에 맵핑 되어야 하는지 설명해준다. 기본적으로 커널은 초기 주소에 프로그램을 맵핑한다. 더욱이 이것은 ‘flags’ 허용으로 ‘가상(virtual)’ 주소 상의 ‘물리적(physical)’ 인 곳에서 프로그램 시작의 정렬된 크기 (assize)를 맵핑 할 것이다.

```
$ rabin -I /bin/ls | grep baddr
baddr=0x08048000

$ rabin -H /bin/ls | grep LOAD
```

```
virtual=0x08060000 physical=0x00018000 asize=0x1000 size=0x0fe0 ₩
flags=0x06 type=LOAD name=phdr_0
```

이 상황에서 파일의 0x18000 주소에서 시작하고 0x8060000 주소에서 로드 될 바이너리의 0xfe0 바이트의 정렬된 크기는 4K이다.

Radare로부터 '.!rabin –rIH $FILE'을 실행함으로써 이 모든 정보는 radare로 가져와질 것이고 이것은 그러한 환경을 대리 실행할 것이다.

여기에서 우리가 직면한 문제는 맵핑된 영역의 주소와 크기가 .data 섹션의 오프셋과 일치하지 않는 점이다.

몇 가지 명령은 바이트의 특정 범위를 암호화 하는 프로그램을 정적으로 변경하고, 그러한 바이트를 동적으로 판독하기 위한 어셈블리의 스니팻을 삽입하기 위해서 정확한 오프셋을 얻을 필요가 있다.

이러한 스크립트는 아래와 같다:

```
-------8<------------
# 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 entrypoint
s entrypoint
```

```
# 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!
-------8<------------
```

코드의 스니팻을 판독하는 것은 아래와 같다:

```
-------8<------------
.global main
main:
   mov $FROM, %esi
   mov $TO, %edi
   0:
   inc %esi
   xorb $0xa8, 0(%esi)
   cmp %edi, %esi
   jbe 0b
   xor %edi, %edi
   ret
-------8<------------
```

마침내 우리는 이 코드를 실행한다:

```
$ cat hello.c
   #include <stdio.h>
```

```
char message[128] = "Hello World";

int main()
{
   message[1]='a';
   if (message[0]!='H')
      printf("Oops₩n");
   else
      printf("%s₩n", message);
   return 0;
}

$ gcc hello.c -o hello
$ strings hello | grep Hello
Hello World
$ ./hello
Hallo World
$ radare -w -i xordata.rs hello
$ strings hello | grep Hello
$ ./hello
Hallo World
```

'Hello World' 문자열은 더 이상 보통의 텍스트 안에 있지 않다. 암호화 알고리즘은 정말 어리석다. 재구성된 데이터 섹션은 디버거를 사용한 실행 과정으로부터 추출 가능하다:

```
$ cat unpack-xordata.rs
   e asm.profile=simple
   s entrypoint
   !cont `pd 20~push dword[3]#2`
   f delta @ section._data- segment.phdr_0.rw_.paddr-section.
   s segment.phdr_0.rw_.vaddr+delta
   b section._data_end - section._data
   wt dump
   q!
   $ radare -i unpack-xordata.rs -d ./hello
   $ strings dump
   Hello World
```

또는 정적으로:

```
e asm.profile=simple
wa ret @ `pd 20~push dword[3]#1`
wox a8 @ section._data:section._data_end-section.data
```

이것은 어떻게 복잡하게 하는 것이 보호를 강화할 수 있고 어떻게 이것을 쉽게 우회할 수 있는지를 보여주는 명백한 예이다.

언패커에 대한 마지막 수정으로 해독하는 루프가 존속하는 곳의 '__libc_csu_init' 심볼에 'ret' 명령만 써주면 된다:

```
wa ret @ sym.__libc_csu_init
```

아니면 엔트리포인트의 csu_init 포인터 대신에 null 포인터를 넣어주면 된다.

```
e asm.profile=simple
s entrypoint
wa push 0 @ `pd 20~push dword[0]#1`
```

--[3.4 – 바이너리의 차이점 찾기

서버 어디에서 변경된 바이너리(체크섬 파일 분석기에 의해 탐지되는)가 발견될 수 있는지 시나리오를 생각해 보자. 그리고 관리자와 시스템 침해 분석가는 그 파일에 무슨 일이 일어났는지 알기 원한다.

설명을 간략하게 하기 위해 이전 장의 예제에 기반을 둘 것이다.

첫 번째 포인트에서는 체크섬 분석이 체크섬 알고리즘에 어떤 충돌이 있는지 그리고 파일들이 실제로 다른지 결정할 것이다.

```
$ rahash -a all hello
par:      0
xor:      00
hamdist: 01
xorpair: 2121
entropy: 4.46
mod255:  84
crc16:   2654
crc32:   a8a3f265
md4:     cbfc07c65d377596dd27e64e0dcac6cd
md5:     2b3b691d92e34ad04b1affea0ad2e5ab
sha1:    8ac3f9165456e3ac1219745031426f54c6997781
sha256:  749e6a1b06d0cf56f178181c608fc6bbd7452e361b1e8bfbcfac1310662d4775
```

```
sha384:  c00abb14d483d25d552c3f881334ba60d77559ef2cb01ff0739121a178b05...
sha512:  099b42a52b106efea0dc73791a12209854bc02474d312e6d3c3ebf2c272ac...

$ rahash -a all hello.orig
par:     0
xor:     de
hamdist: 01
xorpair: a876
entropy: 4.29
mod255:  7b
crc16:   2f1f
crc32:   b8f63e24
md4:     bc9907d433d9b6de7c66730a09eed6f4
md5:     6085b754b02ec0fc371a0bb7f3d2f34e
sha1:    6b0d34489c5ad9f3443aadfaf7d8afca6d3eb101
sha256:  8d58d685cf3c2f55030e06e3a00b011c7177869058a1dcd063ad1e0312fa1de1
sha384:  6e23826cc1ee9f2a3e5f650dde52c5da44dc395268152fd5c98694ec9afa0...
sha512:  09b7394c1e03decde83327fdfd678de97696901e6880e779c640ae6b4f3bf...
```

3개의 다른 체크섬 알고리즘에 이르기 까지 두 가지 다른 비교 결과를 가져올 수 있다 (그리고 미래에는 더 많을 것이다). 만약 그것들 중 아무거나 다르다면 우리는 파일이 다르다고 결정할 수 있다.

```
$ du -bs hello hello.orig
4913     hello
4913     hello.orig
```

파일 둘 다 크기가 같기 때문에 바이트 수준 비교를 사용할 가치가 있을 것이다. 왜냐하면 차이점을 읽는 것이 델티파이된(deltified) 것(이것은 기본적인 비교 알고리즘이다)처럼 그것을 이해하려고 시도하는 것보다 더 쉽기 때문이다.

```
$ radiff -b hello.orig hello
------
000003ed 00
000003f0 55   |   60
000003f1 89   |   be
000003f2 e5   |   c0
...
00000406 fe   |   61
00000407 ff   |   c3
00000408 ff   |   90
00000409 8d   |   90
0000040a bb   |   90
0000040b 18   |   90
0000040c ff
------
000005bd 00
000005c0 00   |   a8
000005c1 00   |   a8
```

```
000005c2 00    |    a8
...
000005df 00    |    a8
000005e0 48    |    e0
000005e1 65    |    cd
000005e2 6c    |    c4
000005e3 6c    |    c4
000005e4 6f    |    c7
0000065f 00    |    a8
00000660 00
```

Radiff는 '------' 줄에 의해서 결정된 두 블락의 변화를 감지한다.

첫 번째 블락은 0x3f0에서 시작되고 두 번째 블락(더 큰)은 주소 0x5c0에서 시작된다.

방대한 00->a8 변환을 보고 있는 동안에 무턱대고 암호화 알고리즘은 XOR로 키는 0xa8로 결정하는 것이 가능하다. 그러나 변경의 첫 번째 블락을 보고 코드를 디스어셈블 하도록 하자.

```
$ BYTES=$(radiff -f 0x3f0 -t 0x40b -b hello.orig hello | awk '{print $4}')
$ rasm -d "$(echo ${BYTES})"
pushad
mov esi, 0x80495c0
mov edi, 0x8049660
inc esi
xor byte [esi], 0xa8
cmp esi, edi
jbe 0x38
popad
ret
...
```

이것이 여기에 있다! 어셈블러의 이 코드는 0x80495c0에서 0x8049660까지 반복을 하고 0xa8로 각각의 바이트를 XOR 한다.

Radiff는 또한 radare 명령 안에 있는 디프의 결과를 덤프할 수 있다. 스크립트의 실행은 하나의 파일을 또 다른 하나로 변환하는 바이너리 패치 스크립트를 결과로 도출할 것이다.

```
$ radiff -rb hello hello.orig > binpatch.rs
$ cat binpatch.rs | radare -w hello
$ md5sum hello hello.orig
6085b754b02ec0fc371a0bb7f3d2f34e   hello
6085b754b02ec0fc371a0bb7f3d2f34e   hello.orig
```

--[3.5 – 라이브러리 의존성 삭제

몇몇 gnu/linux 배포에서 모든 ELF 바이너리는 libselinux에 대해서 링크된다. 이것은 우리의 hello world을 이식 가능하도록 만드는데 있어 바라지 않는 의존성이다. SElinux는 모든 gnu/linux 배포판에서 이용할 수 없다.

이것은 간단한 예이다. 그러나 우리는 바이너리가 최소한의 필요보다 더 많은 라이브러리에 의존하도록 만들 수 있는 나쁜 pkg-config 구성 파일로 문제를 확장할 수 있다.

아마도 대부분의 유저는 걱정하지 않을 것이고 커널을 재컴파일 하던 무엇을 하던지 간에 그러한 의존성을 설치할 것이다. 그러한 라이브러리에 대해 링크하지 않기 위해 gcc에게 말하는 것 대신에 거꾸로 단계를 돌아가는 것은 애석한 일이다.

그러나 우리는 이러한 의존성을 쉽게 피할 수 있다: 모든 라이브러리 이름은 ELF 헤더 안에 0으로 끝나는 문자열로 저장된다. 그리고 로더(최소한 Solaris, Linux, BSD 것의)는 라이브러리 이름의 길이가 0이면 의존성 엔트리를 무시할 것이다.

패치는 요구되지 않은 라이브러리 이름(비슷한 이름의 라이브러리가 1개보다 많다면 접두사)을 찾는 것을 포함한다. 그 다음에는 각각의 검색 결과에 0x00을 쓴다.

```
$ radare -nw hello
[0x00000000]> / libselinux
... (results) ...
[0x00000000]> wx 00 @@ hit
[0x00000000]> q!
```

-nw을 사용하면 radare는 ~/.redarerc을 기계어로 번역하지 않고 실행할 것이다. 이것은 심볼을 탐지하지 않거나 코드를 분석하지 않을 것이다 (우리는 단지 기초 16진법 성능을 사용한다). –w 플래그는 읽기 쓰기 모드로 파일을 열 것이다.

각각의 검색 결과는 '검색(search)' 플래그스페이스에 기본값으로 hit%d_%d가 될 것인 'search.flagname' eval 변수에 의해 미리 정의된 이름으로 저장될 것이다. (이것은 keyword_index와 hit_index 이다).

마지막 명령은 현재 플래그스페이스에서 'hit'이 일치한 모든 '@@' 플래그에서 'wx 00'을 실행할 것이다.

끝내기 전에 변경 사항을 저장하는 것을 원치 않으면 'q!'을 사용한다.

--[3.6 – 시스템호출 어지럽히기

표준 바이너리 상에서 가공하지 않은(raw) int 0x80 명령을 찾는 것은 이것들이 정적으로 컴파일되지 않았다면 아마도 힘든 작업이 될 것이다. 왜냐하면 그들은 종종 프로그램 안에 내장된 라이브러리 안에 존속하기 때문이다.

Libc와 같은 라이브러리들은 시스템호출로 직접 맵핑된 심볼의 로드를 갖고 있다. 그리고 리버서는 정적으로 컴파일된 프로그램 안의 코드를 인식하기 위해서 이러한 정보를 사용할 수 있다.

심볼을 어지럽게 하기 위해서 표준 역추적(backtrace)을 피하기 위한 스택 파괴 이후에 시스템 상에 직접 시스템호출을 내장할 수 있다.

Linux의 새로운 시스템호출 시스템에 대해 깊게 들어가면 int 0x80 시스템호출 메커니즘이 성능 때문에 현재 x86 플랫폼에서 더 이상 사용되지 않는다는 것을 알아차릴 것이다.

새로운 시스템호출 메커니즘은 실행중인 프로세스 메모리 상에 맵핑된 객체를 공유하는 '가짜(fake)' ELF와 같은 커널에 의해서 설치된 트램펄린 함수를 호출하는 것을 포함한다. 이 트램펄린 함수는 현재 플랫폼을 위한 적절한 시스템호출 메커니즘을 구현한다. 이러한 공유 객체를 VDSO라 부른다.

Linux에서는 단지 두 개의 세그먼트 레지스터만 사용된다: cs와 gs이다. 이것에는 단순한 이유가 있다. 첫 번째 것은 전체 프로그램 메모리 맵으로 접근할 수 있도록 해주고 두 번째 것은 정보에 관련된 쓰레드(Thread)를 저장하는데 사용된다.

여기에 %gs로부터 내용을 가져오기 위한 간단한 세그먼트 덤퍼(dumper)가 있다.

```
----------------------8<----------------------------
$ cat segdump.S
#define FROM 0 /* 0x8048000 */
#define LEN 0x940
#define TIMES 1
#define SEGMENT %gs /* %ds */

.global main

.data
buffer:
.fill LEN, TIMES, 0x00
```

```
.text
main:
        mov $TIMES, %ecx
  loopme:
        push %ecx

        sub $TIMES, %ecx
        movl $LEN, %eax
        imul %eax, %ecx
        movl %ecx, %esi /* esi = ($TIMES-%ecx)*$LEN */
        addl $FROM, %esi
        lea buffer, %edi
        movl $LEN, %ecx

        rep movsb SEGMENT:(%esi),%es:(%edi)

        movl $4, %eax
        movl $1, %ebx
        lea buffer, %ecx
        movl $LEN, %edx
        int $0x80

        pop %ecx
        xor %eax, %eax
        cmp %eax, %ecx
        dec %ecx
        jnz loopme

        ret
----------------------8<----------------------------
```

이 포인트에서 우리는 실행 중에 항상 같은 주소에서 VDSO 맵의 내용을 분석하기 위한 임의의 (random) va 사용하지 못하도록 만든다.

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

파일로 세그먼트를 컴파일하고 덤프한다:

```
$ gcc segdump.S
```

```
$ ./a.out > gs.segment
```

그리고 이제 디버거에서 몇몇 작은 분석을 하자:

```
$ radare -d ls  ; debug 'ls' program
  ...
  ; 디버거 안에서 우리는 gs.segment 파일의 내용을 쓸 수 있다.
  ; 초기 상태에서 현재의 탐색(seek)은 eip를 가리키고 있다.

  [0x4A13B8C0]> wf gs.segment
  Poke 100 bytes from gs.segment 1 times? (y/N) y
  file poked.
     offset    0 1  2 3  4 5  6 7  8 9  A B  C D  E F 0123456789ABCDEF
  0x4a13b8c0, c006 e8b7 580b e8b7 c006 e8b7 0000 0000 ....X...........
  0x4a13b8d0, 1414 feb7 ec06 0a93 b305 6beb 0000 0000 ..........k.....
  0x4a13b8e0, 0000 0000 0000 0000 0000 0000 0000 0000 ................
  0x4a13b8f0, 0000 0000 0000 0000 0000 0000 0000 0000 ................
  0x4a13b900, 0000 0000 0000 0000 0000 0000 0000 0000 ................
  0x4a13b910, 0000 0000 0000 0000 0000 0000 0000 0000 ................

  ; 한 번 파일이 프로세스 메모리 안에 넣어지면 우리는 현재 데이터 블락의 내용을 분석할
  ; 다른 방법을 갖는다. 한 가지 가능한 옵션은 우리가 인자로 사용하는 포맷 문자열 다음의
  ; 메모리 내용을 출력하는 'pm' 명령을 사용하는 것이다. 구조체 안으로 엘리멘트(element)의
  ; 이름을 상술하는 것 또한 가능하다.
  ;
  ; 또 다른 하나의 옵션은 포인터 다음에 32bit 메모리를 읽고 코드, 데이터, 문자열 또는 null
  ; 포인터로의 참조를 식별하며 데이터를 분석하는 'ad' 명령이다.

  [0x4A13B8C0]> pm XXXXXXXXX
  0x4a13b8c0 = 0xb7fe46c0
  0x4a13b8c4 = 0xb7fe4ac8
  0x4a13b8c8 = 0xb7fe46c0
  0x4a13b8cc = 0x00000000 ; eax
  0x4a13b8d0 = 0xb7fff400 ; maps._vdso__0+0x400
  0x4a13b8d4 = 0xff0a0000
  0x4a13b8d8 = 0x856003b1
  0x4a13b8dc = 0x00000000 ; eax
  0x4a13b8e0 = 0x00000000 ; eax

  [0x4A13B8C0]> ad
  0x4a13b8c0, int be=0xc046feb7 le=0xb7fe46c0
  0x4a13b8c4, int be=0xc84afeb7 le=0xb7fe4ac8
  0x4a13b8c8, int be=0xc046feb7 le=0xb7fe46c0
     0x4a13b8cc, (NULL)
  0x4a13b8d0, int be=0x00f4ffb7 le=0xb7fff400 maps._vdso__0+0x400
  0xb7fff400,    string "QRU"
  0xb7fff404, int be=0xe50f3490 le=0x90340fe5

  ; 두 가지 분석 모두 0x4a13b8d0에서 vdso map으로의 포인터를 결과로 준다.
  ; 이러한 내용을 'pd'(print disassembly)와 함께 디스어셈블링 하면 sysenter를 사용하는
  ; 코드를 얻을 수 있다.
```

```
[0x4A13B8C0]> pd 17 @ [+0x10]
        _vdso__0:0xb7fff400,   51              push ecx
        _vdso__0:0xb7fff401    52              push edx
        _vdso__0:0xb7fff402    55              push ebp
    .-> _vdso__0:0xb7fff403    89e5            mov ebp, esp
    |   _vdso__0:0xb7fff405    0f34            sysenter
    |   _vdso__0:0xb7fff407    90              nop
    |   _vdso__0:0xb7fff408,   90              nop
    |   _vdso__0:0xb7fff409    90              nop
    |   _vdso__0:0xb7fff40a    90              nop
    |   _vdso__0:0xb7fff40b    90              nop
    |   _vdso__0:0xb7fff40c,   90              nop
    |   _vdso__0:0xb7fff40d    90              nop
    `=< _vdso__0:0xb7fff40e    ebf3            jmp 0xb7fff403
        _vdso__0:0xb7fff410,   5d              pop ebp
        _vdso__0:0xb7fff411    5a              pop edx
        _vdso__0:0xb7fff412    59              pop ecx
        _vdso__0:0xb7fff413    c3              ret
```

이 작은 분석 후에 우리는 시스템호출을 부르는 새로운 방법을 사용할 수 있게 되었고 “int 0x80” 명령을 명령의 다발(bunch)로 변환할 수 있게 되었다.

```
new_syscall:
push %edi
push %esi
movl $0x10, %esi
movl %gs:(%esi), %edi
call *%edi
pop %esi
pop %edi
ret
```

이러한 ‘큰(big)’ 원시 함수를 사용하여 우리는 더 많은 파괴 코드를 추가할 수 있고 정적인 분석을 더 어렵게 만들 수 있다.

동적인 분석을 사용하면 시스템호출이 실행될 때마다 역추적을 덤프하는 간단한 스크립트를 써서 이 보호를 우회할 수 있다.

```
(syscall-bt
  e scr.tee=log.txt
  label:
  !contsc
  !bt
  .label:
```

```
)
```

또는 쉘에서 한줄로:

```
$ echo '(syscall-bt,e scr.tee=log.txt,label:,!contsc,!bt,.label:,) ₩
  &&.(syscall-bt)' | radare -d ./a.out
```

분석을 더 어렵게 만드는 길은 우리가 시스템호출을 실행할 때마다 임의의 오프셋으로 스택의 내용을 파괴하는 것이다. 스택 포인터를 감소시키고 이후에 증가시킴으로써 역추적을 실패하게 만들 수 있다.

스택을 파괴하기 위해서 단지 시스템호출 전에 스택을 증가시키고 호출 이후에 감소시킬 수 있다. 분석가는 적절한 역추적을 하고 각각의 엔트리를 위한 xrefs을 분석하기 위해서 모든 시스템호출 트램펄린을 위한 증가/감소(inc/dec) 오피코드를 패치해야만 할 것이다.

Radare는 시스템호출 이전과 이후에 디버거를 멈추게 만드는 "!contsc" 명령을 제공해준다. 이것은 strace[7] 유틸리티의 손으로 만든 버전을 얻기 위한 "!bt"와 "!st"로 같이 스크립트로 서술될 수 있다.

우리가 홈브류 패커에 쓰기 위해 이용할 수 있는 또 다른 하나의 책략은 암호화 키로 적절한 분석을 위해 패치될 필요가 있는 명령을 사용하는 것이다. 잘못된 키를 사용하여 섹션을 해독하려고 시도할 때 프로그램이 충돌을 일으킨다.

--[3.7 – 라이브러리 심볼 대체

정적으로 링크된 바이너리는 사용된 바이트로 가득 차 있다. 우리의 코드를 삽입하기 위해 이것을 사용할 수 있다.

많은 라이브러리 심볼(libc로부터 예를 들면)은 심볼에서 시스템 호출에 의해서 직접 대체될 수 있거나 그것이 호출한 곳에서 직접 제자리에 놓일 수도 있다. 예를 들면:

```
call exit
```

이것은 아래와 같은 (또한 5 바이트 길이) 명령들로 대체될 수 있다:

```
xor eax,eax
inc eax
int 0x80
```

---

[7] http://en.wikipedia.org/wiki/Strace

```
$ rasm 'xor eax,eax;inc eax;int 0x80'
31 c0 40 cd 80
```

이 방식으로 우리는 계층화된 libc의 종료 구현으로 나아갈 수 있고 우리의 코드로 이것을 덮어쓸 수 있다.

바이너리가 손상된 심볼로 정적 컴파일 되면 우리는 서명을 찾거나 직접 'int 0x80' 명령을 찾아야만 할 것이다. 그러고 나서 구현의 시작을 찾기 위해 코드를 거꾸로 분석하고 이후에 이 포인트로 xrefs을 찾아야 할 것이다.

대부분의 컴파일러는 바이너리의 끝에 라이브러리를 추가한다. 그래서 우리는 쉽게 프로그램이 끝나고 라이브러리가 시작되는 곳에 가상의 선을 '그릴(draw)' 수 있다.

우리의 대상 프로그램 안에 어떤 라이브러리와 버전이 내장되어 있는지 식별하는데 도움을 주는 또 다른 도구가 있다. 이것은 'rsc gokolu' 라고 불린다. 이 이름은 'g**gle kode lurker'을 대표한다. 그리고 이것은 유명한 온라인 코드 브라우저 안에 주어진 프로그램 내의 문자열을 찾을 것이다. 그리고 문자열의 소스에 대한 대략적인 결과를 알려줄 것이다.

우리가 디버깅이나 심볼의 정보를 갖고 있을 때 처리는 훨씬 쉬워진다. 그리고 시스템호출을 사용하는 가공하지 않은 어셈블리 안의 작은 구현으로 내장된 glibc 코드의 첫 번째 바이트를 직접 덮어 쓸 수 있다.

Rabin은 주어진 프로그램의 헤더로부터 정보를 추출하는 radare 안의 유틸리티이다. 이것은 ELF32/64, PE332/32+, CLASS, MACH0을 지원하는 다중 이키텍처(multi-architecture) 다중 플랫폼(multi-platform) 그리고 다중 포맷(multi-format) 'readelf' 또는 'nm' 대체와 같다.

이 도구의 사용은 매우 간단하고 결과는 sed와 다른 쉘 도구로 파싱하기 쉽게 깔끔하다.

```
$ rabin -vi a.out
[Imports]
Memory address        File offset      Name
0x08049034     0x00001034       __gmpz_mul_ui
0x08049044     0x00001044       __cxa_atexit
0x08049054     0x00001054       memcmp
0x08049064     0x00001064       pcap_close
0x08049074     0x00001074       __gmpz_set_ui
...

$ rabin -ve a.out
[Entrypoint]
Memory address:        0x080494d0

$ rabin -vs a.out
```

```
[Symbols]
Memory address       File offset     Name
0x0804d86c    0x0000586c      __CTOR_LIST__
0x0804d874    0x00005874      __DTOR_LIST__
0x0804d87c    0x0000587c      __JCR_LIST__
0x08049500    0x00001500      __do_global_dtors_aux
0x0804dae8    0x00005ae8      completed.5703
0x0804daec    0x00005aec      dtor_idx.5705
0x08049560    0x00001560      frame_dummy
...

$ rabin -h
rabin [options] [bin-file]
 -e        shows entrypoints one per line
 -i        imports (symbols imported from libraries)
 -s        symbols (exports)
 -c        header checksum
 -S        show sections
 -l        linked libraries
 -H        header information
 -L [lib]  dlopen library and show address
 -z        search for strings in elf non-executable sections
 -x        show xrefs of symbols (-s/-i/-z required)
 -I        show binary info
 -r        output in radare commands
 -o [str]  operation action (str=help for help)
 -v        be verbose
```

더 많은 정보는 맨페이지에서 확인할 수 있다.

어떤 아키텍처에서나 바이너리의 어떤 타입에서도 같은 플래그를 사용할 수 있다.

-r 플래그를 사용하면 결과는 radare 명령 안에서 포맷될 것이다. 이것이 바이너리 정보를 가져올 수 있는 방법이다. 이 명령은 심볼, 섹션, 임포트(imports) 그리고 바이너리 정보를 가져온다.

```
[0x00000000]> .!!rabin -rsSziI $FILE
```

만약 file.id=true이면 이것은 파일을 로딩할 때 자동으로 수행된다. 분석을 사용하지 못하도록 하기 원한다면 'radare'을 '-n' 옵션으로 실행한다 (왜냐하면 이것이 ~/.radarerc을 기계어로 번역하지 않도록 하는 길이기 때문이다).

'rabin'에 의해 주어진 힌트 덕분에 심볼을 갖고 있는 프로그램과 디버그 정보는 'radare'의 코드 분석가에 의해 대부분 분석된다. 그래서 이론적으로 함수로의 상호 참조를 자동적으로 해결할 수 있을 것이다. 그러나 만약 저렇게 운이 좋지 못하면 우리는 간단한 디스어셈블리 출력 포맷(e asm.profile=simple)을 준비할 수 있고 원하는 소유한 심볼에 대한 호출을 분석하기 위한 'call 0x???????' 을 그레핑(grepping)하는 전체 프로그램 코드를 디스어셈블 할 수 있다.

```
[0x08049A00]> e scr.color=false
[0x08049A00]> e asm.profile=simple
[0x08049A00]> s section._text
[0x08049A00]> b section._text_end-section._text
[0x08049A00]> pd~call 0x80499e4
0x08049fd9    call 0x80499e4  ; imp.exit
0x0804fa89    call 0x80499e4  ; imp.exit
0x0805047b    call 0x80499e4  ; imp.exit
[0x08049A00]>
```

만약 우리가 운이 좋지 않아 심볼을 갖고 있지 않으면 우리는 바이너리의 어떤 부분이 우리의 대상 바이너리의 부분인지 확인해야 할 것이다.

```
$ rsc syms-dump /lib/libc.so.6 24 > symbols
```

--[3.8 – 체크섬 하기

바라지 않은 실행시간이나 정적인 프로그램 패치를 검사하기 위한 가장 일반적인 절차는 섹션, 함수, 코드, 또는 데이터 블락에 대한 체크섬 하는 명령을 사용하여 이루어진다.

이런 종류의 보호는 종종 쉽게 패치할 수 있다. 그러나 실수요자(end user)가 가공되지 않은 세그먼테이션 오류 대신 감사한 에러 메시지를 얻을 수 있게 해준다.

에러 메시지 문자열은 체크섬하는 코드의 위치를 잡기 위한 리버서에 대한 간단한 엔트리포인트가 될 수 있다.

Radare는 "h" 캐릭터(char) 하에서 해시 명령을 제공하고 이는 디버깅 동안 또는 정적으로 디스크로부터 파일을 염으로써 블락에 기반한 체크섬을 계산할 수 있도록 해준다.

여기에 사용 예가 있다:

```
hmd5
```

또한 특정 체크섬 값과 일치하는 프로그램의 조각을 찾기 위해서나 해시 검사를 패치하지 않기 위한 단지 새로운 합(sum)을 계산하기 위해서 이러한 해시 명령을 사용하는 것이 가능하다.

'h' 명령은 "rahash" 프로그램을 사용함으로써 쉘로부터 또한 수행될 수 있다. 이 프로그램은 바이트의 블락에 기반을 둔 단일 파일에 대한 다중 체크섬을 계산할 수 있도록 허용해준다. 단지 참 거짓 응답을 받지 않고 체크섬을 원할 때 (만약 프로그램이 변경되었으면) 이것은 일반적으로 침해 분석에 사용된다. 왜냐하면 같은 파일의 다른 오프셋에 다중 체크섬을 갖고 있기 때문에 이 방법으로 문제를 바이너리의 작은 조각으로 나눌 수 있기 때문이다.

하드디스크, 디바이스 펌웨어 또는 같은 크기의 메모리 덤프 위에서 이러한 동작을 취할 수 있다.

만약 델티파이된 내용과 같은 유사한 파일을 다룬다거나 더 많은 좋은 입자를 가진(fine grained) 결과를 필요로 한다면 "radiff"을 시도해 보아야 한다. 이것은 바이트 수준으로부터 기본 블락, 접근된 변수, 호출된 함수의 정보를 사용하여 코드 수준의 비교를 하기 위해 변경을 감지하는 다중 바이너리비교(bindiffing) 알고리즘을 제공한다.

우리는 또한 체크섬 알고리즘에 대한 보호 메소드를 결정해야 한다. 왜냐하면 이것은 소프트웨어 중단점을 가능하게 하고 유저가 정의한 패치를 되게 하기 위해 필요한 첫 번째 패치일 것이기 때문이다.

해시기(hasher)를 보호하기 위해서 프로그램의 나머지와 함께 이것의 코드를 섞음으로써 프로그램에 따라 분포되도록 이것을 놓을 수 있다. 이렇게 하여 이것의 위치를 찾기 더 어렵도록 만든다. 우리는 또한 프로그램을 실행하는데 필요한 코드와 함께 .data 안에 있는 암호화된 영역에 이 코드를 배치하는 선택을 할 수 있다.

리버서에 대한 주요한 골칫거리 중 하나는 보호를 계층화 하는 것이다. 이것은 적절한 통로 안에 위치시키고 패치하는 것을 더 어렵게 만드는 같은 코드로의 많은 엔트리포인트에 의해 기술될 수 있다.

섹션을 체크섬하는 것은 다른 암호화된 섹션에 대한 해독 키처럼 사용될 수 있다. 패치를 쉽게 하는 것을 피하기 위해서 프로그램은 해독 코드의 일관성을 검사해야 한다. 이것은 또 다른 하나의 체크섬을 검사하거나 해독과 mmap 코드의 시작에서 약간의 바이트를 비교함으로써 이루어질 수 있다.

더 유연성 있는 방법으로 체크섬하는 알고리즘을 구현할 수 있는 가능한 또 하나의 디자인이 있다. 이 방법은 결과 값을 어딘가에 하드코딩 하는 것 대신에 실행시간에 코드를 체크섬하는 것으로 이루어진다.

이 메소드는 영역의 체크섬 값을 정적으로 계산하지 않고도 다중 함수를 '보호(protect)' 한다. 이 메소드의 문제는 바이너리 상의 정적인 변환에 대한 보호를 제공하지 않는다는 것이다.

이러한 스니팻은 체크섬이 일치할 때 0을 리턴하는 수행되며 체크섬하는 함수를 완전히 얻기 위해 DEBUG=0으로 컴파일 될 수 있다.

```
--------------------8<-------------------------
#define CKSUM 0x68
#define FROM 0x8048000
#define SIZE 1024
```

```
str:
.asciz "Checksum: 0x%08x₩n"

.global cksum
cksum:
        pusha
        movl $FROM, %esi
        movl $SIZE, %ecx
        xor %eax, %eax
        1:
        xorb 0(%esi), %al
        add $1, %esi
        loopnz 1b
#if DEBUG
        pushl %eax
        pushl $str
        call printf
        addl $8, %esp
#else
        sub $CKSUM, %eax
        jnz 2f
#endif
        popa
        ret

#if DEBUG
.global main
main:
        call cksum;
        ret
#else
        2:
        mov $20, %eax
        int $0x80
        mov $37, %ebx
        mov $9, %ecx
        xchg %eax, %ebx
        int $0x80
#endif
```

---------------------8<--------------------------

밀어 넣어진 함수의 바이트를 얻기 위해서 우리는 바이너리 안의 심볼에 의해서 관리되는 포함된 16진수 쌍(hexpairs)을 덤프하는 'rsc' 스크립트를 사용할 수 있다:

```
$ gcc cksum.S -DDEBUG
$ ./a.out
Checksum: 0x69

$ gcc -c cksum.S
$ rsc syms-dump cksum.o | grep cksum
cksum: 60 be 00 80 04 08 b9 00 04 00 00 31 c0 32 06 83 c6 01 e0 f9 2d d2 04 ₩
00 00 75 02 61 c3 b8 14 00 00 00 cd 80 bb 25 00 00 00 b9 09 00 00 00 93 cd 80
```

우리의 정렬된 값으로 패치하기 위한 오프셋은:

```
+2: original address (4 bytes in little endian)
+7: size of buffer (4 bytes in little endian)
+21: checksum byte (1 byte)
```

대상 프로그램에 코드를 끼워 넣는 동안에 자동적으로 정확한 바이트에 패치를 하는 radare 매크로로부터 이 16진수 덤프(hexdump)를 사용할 수 있다.

```
$ rsc syms-dump cksum.o | grep cksum | cut -d : -f 2 > cksum.hex
$ cat cksum.macro
(cksum-init ckaddr size,f ckaddr @ $0,f cksize @ $1,)
(cksum hexfile size
   # kidnap some 4 opcodes from the function
   f kidnap @ `aos 4`
   yt kidnap ckaddr+cksize
   wa jmp $$+kidnap @ ckaddr+cksize+kidnap
   wa call `?v ckaddr`
   wF $0 @ ckaddr
   f cksize @ $$w
   f ckaddr_new @ ckaddr+cksize+50
   wv $$ @ ckaddr+2
   wv $1 @ ckaddr+7
   wx `hxor $1` @ ckaddr+21
   f ckaddr @ ckaddr_new
)
```

```
$ cat cksum.hooks
0x8048300
0x8048400
0x8048800

$ radare -w target
[0x8048923]> . cksum.macro
[0x8048923]> .(cksum-init 0x8050300 30)
[0x8048923]> .(cksum cksum.hex 100) @@. cksum.hooks
```

--[4 – 코드 참조와 놀기

프로그램 내 코드의 일부 사이의 관계는 코드 블락과 함수 사이의 의존 관계를 이해하는데 좋은 정보의 원천이다.

다음에 오는 하위 장에서는 메모리 내부나 정적으로 디스크 상에 있는 프로그램으로부터 정보를 검색하기 위한 코드 참조를 어떻게 생성하고, 검색하고, 사용해야 하는지 설명할 것이다.

--[4.1 – xrefs 찾기

어떻게 프로그램이 동작하고 무엇을 하는지 이해할 때 우리가 필요한 가장 흥미로운 정보는 함수 사이의 흐름 그래프이다. 이것은 프로그램의 큰 그림을 알려주고 중요한 포인트를 더 빨리 찾게 해준다.

radare에서 이 정보를 얻는 방법은 이것과 다르다. 그러나 마침내 'Cx'와 'CX' 명령(x=code xref, X=data xref)으로 이것을 다룰 것이다. 이것은 심지어 이러한 정보를 뽑아내기 위한 당신만의 프로그램을 작성할 수 있고 프로그램이나 파일의 출력을 단지 기계어로 번역함으로써 radare로 이것을 가져올 수 있음을 의미한다.

```
[0x8048000]> !!my-xrefs-analysis a.out > a.out.xrefs
[0x8048000]> . a.out.xrefs
or
[0x8048000]> .!my-xrefs-analysis a.out
```

이것을 작성할 때 radare는 기본적인 코드 분석 알고리즘을 구현한다. 이 알고리즘은 함수의 경계를 인지하고, 기본 블락을 쪼개며, 지역 변수와 인자를 찾고, 그것들의 접근을 추적하고 또한 각각의 명령어의 코드와 데이터 참조를 확인한다.

‘file.analyze’가 ‘true’일 때 이 코드 분석은 rabin에 의해 확인되는 모든 심볼과 엔트리포인트 위에서 재귀적으로 수행된다. 이것은 포인터 테이블, 레지스터 분기(branches) 등의 다음에 오는 코드 참조를 자동적으로 분석할 수 없으며 많은 코드가 분석되지 않은 상태로 남을 것이라는 것을 의미한다.

우리는 ‘.afr@addr’ 명령을 사용하여 특정 오프셋에서 함수를 분석하도록 수동적으로 강제할 수 있다. 이 명령은 특정 주소(addr) 에서(@) ‘함수의 재귀적 분석(analyze function recursively’)을 대표하는 ‘afr’ 명령의 결과(‘.’)를 기계어로 번역할 것이다.

‘호출(call)’ 명령이 인식되었을 때 코드 분석은 종료점(endpoint)으로의 명령으로부터 xref 코드를 기록할(register) 것이다. 이것은 왜냐하면 컴파일러가 서브루틴 또는 심볼 안으로 호출하기 위해 이러한 명령을 사용하기 때문이다.

우리가 이전에 했던 것과 같이 우리는 ‘호출(call)’ 명령을 찾기 위해 텍스트 섹션 전체를 디스어셈블 할 수 있다. 그리고 이런 두 줄로 소스와 목적지 주소 사이에 xrefs 코드를 기록할 수 있다:

```
[0x000074E0]> e asm.profile=simple
[0x000074E0]> "(xref,Cx `pd 1~[2]`)"
[0x000074E0]> .(xref) @@=`pd 100~call[0]`
```

디스어셈블리 텍스트 파싱을 더 쉽게 만들기 위해 간단한 asm.profile을 사용할 수 있다.

첫 번째 줄은 ‘xref’라 명명된 매크로를 기록할 것이다. 이 매크로는 디스어셈블 된 명령의 세 번째 단어에 의해서 강조된 주소에 xref 코드를 만든다.

두 번째 줄에서 이것은 “pd 100~call[0]” 명령의 출력에 의해 상술된 모든 오프셋에 이전의 매크로를 실행할 것이다. 이것은 다음에 오는 100개의 명령의 모든 호출 명령에 대한 오프셋이다.

코드를 디스어셈블링 할 때 (‘pd’ 명령을 사용하여 (디스어셈을 출력)) 만약 asm.xrefs가 가능하게 되어있다면 xrefs가 보여질 것이다. 임의로 코드 참조의 종료점(end-points) 둘 다에 xref 정보를 보이게 하는 asm.xrefsto을 놓을 수 있다.

--[4.2 – 눈 먼(blind) 코드 참조

Radare와 함께 온 xrefs(1) 프로그램은 특정 패턴과 일치하는 바이트의 순차(sequences)를 인지하는 것에 기반을 둔 눈 먼(blind) 코드 참조 검색 알고리즘을 제공한다.

이 패턴은 오프셋에 의존하여 계산된다. 이 방식으로 프로그램은 현재의 오프셋과 목적지 주소 사

이의 상대적인 변화량(delta)을 계산한다. 그리고 프로그램의 또 다른 곳을 가리키는 하나의 포인트로부터 상대적인 분기(branches)를 찾으려고 시도한다.

현재 구현은 x86, arm, powerpc을 지원한다. 그러나 명령의 크기, 엔디언(endianness), 변화량(delta)를 계산하는 방법 등을 결정하는 옵션 플래그를 사용함으로써 다른 아키텍처를 위한 분기(branch) 명령의 탬플릿을 정의할 수 있도록 허용한다.

이 프로그램의 주요 목적은 코드 분석이 매우 오래 걸리거나 많은 수동적인 상호 작용을 필요로 하는 곳에서 큰 파일(펌웨어 같은) 상의 xrefs 코드를 인식한다. 이것은 절대적인 호출과 점프를 인식하도록 설계되어 있지는 않다.

여기에 사용 예가 있다:

```
$ xrefs -a arm bigarmblob.bin 0x6a390
0x80490
0x812c4
```

--[4.3 – xrefs 그래프 그리기

Radare는 'gu' 명령(graph user)을 사용하여 기본 블락, 함수 블락 또는 우리가 원하는 것에 기반한 코드 분석을 보여주는 상호 작용하는 그래프를 만들고 관리하기 위한 약간의 명령을 포함하고 있다.

그래프 성능을 사용하는 가장 간단한 방법은 'ag' 명령을 사용하는 것이다. 이 명령은 기본 블락으로 코드를 나누면서 현재 탐색으로부터 코드 분석 그래프와 함께 gtk 윈도우를 바로 화면상에 보여줄 것이다. (기본 블락이나 함수 블락을 분석하기 위한 점프나 호출에 의해서 블락을 나누는 eval graph.jmpblocks와 graph.callblocks을 사용한다).

그래프는 내부적으로 저장된다. 그러나 우리만의 그래프를 만들고 싶다면 그래프를 초기화하고 가지(nodes)와 모서리(edges) 만들고 그것들을 보여주는 서브명령을 제공하는 'gu' 명령을 사용할 수 있다.

```
[0x08049A00]> gu?
Usage: gu[dnerv] [args]
 gur              user graph reset
 gun $$ $$b pd    add node
 gue $$F $$t      add edge
 gud              display graph in dot format
 guv              visualize user defined graph
```

이것은 xrefs 메타데이터 정보를 사용한 분석된 함수 사이의 그래프를 생성할 radare 스크립트 이

다.

그래프를 구성한 뒤에 내부 뷰어 (guv) 또는 graphviz's dot을 사용하여 이것을 보일 수 있다.

```
--------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<---------------
```

여기에 앞의 스크립트에 대한 설명이 있다:

모든 명령은 '"'로 둘러 쌓여 있음을 명심하자. 왜냐하면 이는 안쪽의 특수 문자('>', '~' ...)를 무시하는 방법이기 때문이다.

"(xrefs-to from,f XT @ `Cx~$0[3]`,)"

'from'으로 명명된 하나의 인자를 받아들이는 'xrefs-to'라 불리는 매크로를 기록(register)한다. 그리고 매크로 몸체 내부에 $0으로 별칭 한다.

매크로는 Xref 코드(Cx)를 포인트하는($0 (매크로(from) 의 첫 번째 인자)과 일치하는 줄의 세 번째 칸) 곳의 주소에서('@') 'f XT'(XT 플래그를 만든다)을 실행한다.

"(graph-xrefs-init,gur,)"

'graph-xrefs-init' 매크로는 그래프 사용자 메타데이터를 초기화 한다.

"(graph-xrefs,.(xrefs-to `?v $$`),gue $$F XT,)"

쉼표로 나눠진 명령을 실행하는 매크로는 'graphs-xrefs'라 불린다:

.(xrefs-to `?v $$`)

현재 탐색(seek) ($$)에서 'xrefs-to' 매크로를 실행한다. '$$' 특별한 변수의 값을 구하고 16진수 값으로 값을 출력하는 '?v $$' 실행의 출력으로 이 값이 연결된다.

gue XF XT

현재 검색(seek)에서 함수의 시작과 XT 사이에 'graph-user' 모서리(edge)를 추가한다.

"(graph-viz,gud > dot,!!dot -Tpng -o graph.png dot,!!rsc view graph.png,)"

이 명령은 점(dot) 파일을 생성하고 png로 랜더링 하고 이것을 보여줄 것이다.

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

그 다음에 우리는 이러한 모든 매크로를 묶을 것익 xrefs 그래프가 보여질 것이다.

--[4.3 – xrefs 무작위화

또 다른 하나의 우리가 할 수 있는 성가신 저 수준 조작은 임의 변경으로 임의의 장소에서 함수를 반복하는 것이다. 임의 변경은 프로그램 코드를 더 장황하게 만들고 제어 흐름 경로를 복잡하게 하기 때문에 이를 분석하기 어렵게 만든다.

심볼 사이의 교차 참조(cross-references) (xrefs) 분석을 함으로써 어떤 심볼이 단일 포인트로부터 보다 더 많이 참조되는지 결정할 수 있다.

내부 grep 문법은 주어진 위치에서 xrefs에 대한 오프셋의 목록을 검색하기에 어려울 수도 있다.

```
[0x0000c830]> Cx
Cx 0x00000c59 @ 0x00000790
Cx 0x00000c44 @ 0x000007b0
Cx 0x00000c35 @ 0x00000810
Cx 0x00000c26 @ 0x000008a0
...
```

이 숫자들은 콜러(caller)의 주소와 호출된 주소를 의미한다. 이것은 아래와 같다:

```
0xc59 -(calls)-> 0x790
```

다른 위치로 코드를 참조하는 오피코드를 디스어셈블 하기 위해 각각의 문장에 'pd' 명령을 사용할 수 있다:

```
[0x0000c830]> pd 1 @@= `C*~Cx[1]`
```

이 명령은 'C*~Cx[1]'의 출력의 첫 번째 칼럼 안에 상술된 모든 주소 상에서 'pd 1' 명령을 실행할 것이다.

--[5 – 결론

이 문서는 리버스 엔지니어링, 저 수준 프로그램 조작, 바이너리 패치와 디프 등의 많은 양상을 다루어왔다. 구체적인 작업을 하였음에도 불구하고 명령 줄 16진수 에디터가 필수적인 도구가 되는 곳에서 많은 시나리오를 보여주려고 시도해왔다.

몇몇의 외부 스크립트 언어가 사용될 수 있다. Radare 명령의 숨은 스크립팅에 대해 신경 쓰지 않아도 된다. Radare를 통제하기 위한 perl, ruby, python에 대한 API가 있으며 코드를 분석하거나 원격으로 radare에 연결하면 된다.

배치(batch) 모드를 사용하여 많은 분석 작업을 자동화 할 수 있고 코드의 그래프를 생성하고 이것이 패킹(packed) 또는 감염(infected)되었는지를 결정하고 취약점을 찾을 수 있다.

아마도 이 소프트웨어의 더 중요한 특징은 입출력(IO)의 추상화일 것이다. 이는 하드디스크 이미지, 시리얼 연결, 프로세스 메모리, haret wince 장치, bochs와 같이 gdb을 가능하게 해주는 응용프로그램, vmware, qumu, 공유 메모리 영역, winedbg 등을 열어볼 수 있게 해준다.

우리가 이 문서에서 다루지 않은 많은 부분이 있지만 바이너리에 대한 여러 보호 기법을 설명하는 동안 중요한 개념은 보여줬다.

바이너리 보호 기법을 구현하기 위한 스크립트 가능한 도구를 사용하면 개발이 쉽고, 재사용이 가능하며 분석과 파일의 조작을 빠르도록 해 줄 것이다.

--[6 – 미래 작업

2006년에 프로젝트가 시작되었고 매우 빨리 발전하였다.

이 문서를 쓰는 순간에 개발은 두 가지 다른 분기(branches): radare와 radrea2 로 완료 되었다.

두 번째 것은 여러 개의 라이브러리를 향한 radare 프레임워크의 모든 특징들을 다시 구현하는데 초점을 맞춘 원래 코드의 완전한 리팩토링이다.

이 디자인은 특정한 또는 조직된 라이브러리의 집합을 위한 전체 스크립트 지원을 가능하게 해주는 r1 디자인의 몇 가지 제한을 무시할 수 있도록 허용해 준다. 각각의 모듈은 플러그인으로 그것의 기능을 확장한다.

패롯(parrot)을 가상 머신으로 사용하는 동안에 여러 스크립트 언어가 함께 사용될 수 있다. 스크립트 언어 불가지론자가 되면 여러 프레임워크를 함께 섞을 수 있는 가능성을 열어준다. (metasploit [5], ERESI [6] 등 처럼)

라이브러리, 플러그인 그리고 스크립트에 기반한 모듈화 디자인은 써드파티(third party) 개발자에 의해서 프레임워크를 확장하고 추가로 코드를 더 유지 가능하게 하고 버그가 적도록 할 수 있는 새로운 방법을 열 것이다.

새로운 응용프로그램은 r2 라이브러리 집합의 제일 위에서 나타날 것이다. 웹 프런트 엔드(frontends)와 그래픽 프런트 엔드로부터 이것을 통합하기 위한 방법에 초점을 맞춘 몇 가지 프로젝트가 있다.

플러그인은 새로운 아키텍처, 바이너리 포맷, 코드 분석 모듈, 디버깅 백 엔드(backends), 원격 프로토콜, 스크립트 설비 등을 위한 지원을 추가하는 모듈을 확장할 것이다.

이것을 쓰는 시점에 radare2 (aka libr)는 실제로 충분히 개발되지 않았다. 그러나 radare로 배포되는 응용프로그램을 다시 구현하기 위해 시도하는 몇 가지 테스트 프로그램을 제공한다.

--[7 – 참조

[0] radare homepage
http://www.radare.org

[1] radare book (pdf)
http://www.radare.org/get/radare.pdf

[2] x86-32/64 architecture
http://www.intel.com/Assets/PDF/manual/253666.pdf

[3] ELF file format
http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

[4] AT&T syntax
http://sig9.com/articles/att-syntax

[5] Metasploit, ruby based exploit development framework
http://www.metasploit.com/

[6] ERESI
http://www.eresi-project.org/

--[8 – 감사의 말

이 문서를 쓰는 동안 지원을 아끼지 않은 JV에게 감사한다. 그리고 발행을 가능하게 해준 Phrack 직원에게도 감사를 표한다.

나는 또한 radare을 개발하고 이 문서를 쓸 동안 도움을 준 몇 사람들에게 인사하기를 원한다.

SexyPandas .. for the great quals, ctfs and moments
nopcode .. for those great cons, feedback and beers
nibble .. for those pingpong development sessions and support
killabyte .. for the funny abstract situations we faced everyday
esteve .. for the talks, knowledge and the arm/wince reversing tips
wzzx .. for the motivators, feedback and luls
lia .. my gf which supported me all those nightly hacking hours :)
..

확실히…

이 글을 읽으신 모든 분들께 감사한다. 또한 'underground'라 불리는 생태계(ecosystem)와 우리에게 많은 재미를 주는 컴퓨터 보안의 부분을 담당하고 계신 분들에게도 감사를 표한다.

여전히 이루어져야 하는 많은 작업이 있다. 아이디어와 기부는 언제나 환영한다. 나에게 코멘트하는 것은 자유이다.

--pancake

--------[EOF