

Memory forensics for FreeBSD

Document Language: Korean

본 문서는..

본 문서는 작성된 챕터를 상세하게 기술하거나 새로운 챕터를 추가하는 등, 계속해서 그 내용을 채워갈 예정이며, 하나의 게시글과 현재 페이지에 그 업데이트 내용을 간략하게 기재할 예정입니다. 또는 본 내용을 토대로 더 큰 주제를 가지는 물리 메모리 포렌식 문서를 생성할 생각도 하고 있습니다.

문서에 기재된 내용은 저자가 커널 소스를 분석하고 그간 연구한 내용을 토대로 작성되었기 때문에, 틀린 점이 발생할 수도 있다. 하지만, 대부분의 내용이 실제 커널 소스를 기반으로 구성되어 있기 때문에, 내용이 크게 틀리진 않을 것이라 생각한다.

문서는 편의를 위해 경어체를 사용할 예정이며, 문서를 통해 공개된 도구는 모두 구글 프로젝트에 오픈소스로 공개되어 있습니다. 제가 제공하는 모든 분석 도구는 파이썬 기반으로 개발되었으며, 소스 공유 차원에서 GPL v2 라이선스를 걸었습니다. 많은 분들이 도구를 사용하시고 좋은 아이디어가 있으면 저와 같이 정보 교류를 통해 더 유용하고 효율적인 물리 메모리 분석 도구로 발전했으면 하는 바람이 있습니다.

< 문서의 저작권은 본 글쓴이에게 있으며, 타 사이트에 올려도 되지만 원 출처만 통보해주시면 감사하겠습니다. >

Writer: n0fate (Real Name: Kyeongsik Lee)

Email: rapfer@gmail.com or n0fate@live.com

업데이트 내용

2011. Jan. 10. 최초 문서 발행.

Table of Contents

Memory forensics for FreeBSD

1

소개 1

물리 메모리 포렌식을 통한 데이터 습득 1

물리 메모리 이미징 방법

2

물리 메모리 분석을 위한 추가 정보 획득

3

OS Feature

4

운영체제의 프로세스 관리 4

운영체제의 메모리 관리 방식 5

프로세스가 사용하는 메모리 영역 관리

6

외부 커널 모듈 관리 9

소개 9

커널 모듈 정보 관리

9

물리 메모리 분석을 통한 외부 커널 모듈 정보 습득

10

커널 이미지 분석 10

ELF 파일 포맷

11

volafunx(volafox for Unix)

13

프로세스 목록 출력

13

프로세스 덤프

14

외부 커널 모듈 목록 출력

15

외부 커널 모듈 덤프

15

시스템 콜 목록 출력

15

시스템 콜 후킹 탐지

16

Appendix

17

volafox (<http://code.google.com/p/volafox>)

17

Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A

17

An error 0xBAADC0DE occurred (<http://baadc0de.blogspot.com>)

17

FreeBSD/Linux Cross Reference (<http://fxr.watson.org>)

17

Detecting Loadable Kernel Module(LKM) Rootkit

17

Designing BSD Rootkits

17

Executable and Linkable Format(ELF)

17

Memory forensics for FreeBSD

소개

FreeBSD는 개인 데스크탑 시장보다는 서버 시장에 중점을 둔 시스템으로 미 버클리 대학에서 개발한 오픈소스 운영체제이다. 본래 BSD라는 단일 커널로 시작하여 최근에는 FreeBSD, NetBSD, OpenBSD와 같이 각각의 특징점에 초점을 두어 운영체제가 개발되고 있다. 이 중 가장 높은 점유율을 가지는 운영체제가 FreeBSD이며, 이를 반증하듯, 야후와 같은 IT 대기업에서 많이 채용하고 있는 운영체제이다.

국내뿐만 아니라 해외의 디지털 포렌식 분야에서 현재 메모리 포렌식 연구가 진행되고 도구화까지 이루어진 운영체제는 다음과 같다.

- Windows - HB Gary (commercial), pt finder, volatility framework, MEMA (DFRC, commercial)
- Linux - Draugr, Volatility
- Mac OS X - volafox (my project :))
- BSD - volafunx (my project :))
- HP-UX - X
- Solaris - X

이러한 상황이다보니 실제 기업에서 서버로 사용되는 유닉스 운영체제에서는 메모리 포렌식 기법에 한계에 직면하게 된다. 이에 본 글에서는 저자가 개발한 FreeBSD 메모리 포렌식 도구를 소개하고 이를 통해 어떠한 데이터를 수집 및 분석할 수 있는지 알아보기로 한다.

물리 메모리 포렌식을 통한 데이터 습득

우선, 우리가 유닉스 시스템에서 메모리 포렌식을 수행할 경우 필요한 정보는 크게 네 가지로 볼 수 있다.

첫 번째는 프로세스 목록을 확인할 수 있다. 리눅스/유닉스 기반 운영체제를 공격하기 위해 해커는 시스템 콜의 함수 주소를 수정하여, 자기자신을 숨기는 기법을 사용한다. 윈도우의 경우 프로세스 관리를 프로세스가 아닌 스레드 체인을 통해 관리하기 때문에, EPROCESS 구조체간 더블 링크드 리스트 형태의 구조를 잘라내어 자기 자신을 숨길 수 있지만, 유닉스 시스템의 경우 proc 구조체의 프로세스 체인을 통해 관리하기 때문에, 이 부분을 제거할 경우, 루트킷이 정상적으로 동작하지 않게 된다. 이에 프로세스를 숨기기 위해서는 프로세스 목록을 호출하는 ps, top와 같은 함수에서 사용하는 시스템 콜의 함수 포인터를 변경하는 방법을 통해 결과를 조작하여 자신을 숨기는 기법을 사용한다. 이러한 프로세스 은닉 기법은 본 도구의 프로세스

덤프에서 사용되는 메모리 포렌식 기법을 이용할 경우, 자료형에 대한 무결성 손실 없이, 메모리 상에 있는 프로세스 체인을 추적하여 데이터를 획득하므로, 프로세스 은닉 기법을 탐지할 수 있다.

두 번째는 프로세스 메모리 영역 덤프가 될 수 있다. 이는 메모리 이미징 시점의 프로세스의 상황을 확인하고, 각 프로세스의 가상 메모리 영역에 존재하는 다양한 정보를 수집하여, 각 사용자가 프로세스를 생성하여 어떠한 행동을 하였는지 확인할 수 있다. 보통 사용자가 생성하는 정보는 사용자가 명시적 저장을 수행하지 않는 이상 프로세스의 가상 메모리 상에 버퍼로 존재하기 때문에, 프로세스 덤프는 디지털 포렌식 및 악성코드 분석가 입장에서 유용하게 사용할 수 있는 정보이다.

세 번째는 유닉스의 외부 커널 모듈 목록 정보를 획득할 수 있다. 유닉스와 리눅스 시스템은 구 커널에 존재했던 커널 컴포넌트 업데이트를 위해 시스템을 재부팅해야했던 단점을 보완하기 위해 하나의 커널 코어에 여러 컴포넌트 형태로 들러붙도록 동적 커널 모듈 기능을 제공하고 있다. BSD에서는 이를 LKM(Loadable Kernel Module)이라고 부른다. 동적 커널 모듈은 동작할 경우 루트 권한을 가지며, 커널의 가상 메모리와 모듈의 가상 메모리 간에 정보를 공유하는 API를 제공(`copyin()`, `copyout()`)을 제공하기 때문에, 이를 이용하여, 시스템 콜 후킹을 하거나, 커널 가상 메모리를 조작하는 등 다양한 행위를 할 수 있다. 메모리 포렌식 기법을 이용하면, 커널 모듈을 숨기는 행위를 탐지하는 등, 동적 커널 모듈의 정보를 획득 및 덤프 할 수 있다.

네 번째는 시스템 콜 목록 정보를 획득할 수 있다. 앞 서 설명한 바와 같이 유닉스 시스템에서 파일/프로세스/네트워크 포트 은닉과 같은 기법은 대부분 시스템 콜 후킹을 통해 이루어 진다. 보통 시스템 콜은 시스템 콜 테이블에 존재하는 실제 콜 함수의 포인터 주소를 변경하여 자신이 원하는 행위를 수행하거나, 정상 시스템 콜을 자신이 호출하여 그 결과를 조작하는 행위를 한다. 시스템 콜 후킹 탐지 기법은 본 저자가 개발한 도구인 'volafox'의 시스템 콜 후킹 탐지 기법을 이용하여 BSD에서 후킹된 시스템 콜을 탐지할 수 있도록 그 기능을 지원한다.

물리 메모리 분석은 이러한 네 가지 요소를 수집하기 위해서도 필요하지만, 어떠한 시스템의 라이브 포렌식 분석 시, 해당 시스템으로부터 얻은 디지털 증거의 무결성과 신뢰성을 확보하기 위해서도 꼭 필요한 부분이다. 만약 시스템 콜 후킹을 통해 몇몇 이름의 파일을 은닉하는 악성코드가 존재하는 경우, ls와 같은 파일 목록을 출력하는 프로그램을 통한 라이브 포렌식 기법의 결과물은 그 증거의 신뢰성에 타격을 입게 된다. 또한 라이브 포렌식이 해당 시스템이 켜져있는 상황에서 다양한 포렌식 도구를 사용하기 때문에, 주 저장 장치인 물리 메모리와 보조 저장 장치인 하드 디스크의 무결성 훼손이 발생할 수 있다. 이러한 문제를 해결하기 위해서라도 물리 메모리를 무결성 침해로 최소화하는 방법(firewire 케이블의 사용 등)을 통해 물리 메모리를 통해 최대한 데이터를 추출할 수 있도록하고, 차 후 부족한 부분에 대해서만 활성 시스템 포렌식 기법을 적용하도록 하는 것이 디지털 포렌식 관점에서 올바른 방법이라 할 수 있다.

물리 메모리 이미징 방법

물리 메모리를 수집하는 방법은 여러가지가 있을 수 있다. 가장 먼저 생각할 수 있는 방법은 firewire 라인을 통해 하드웨어 컨트롤러에 직접 접근하여, 물리 메모리를 수집하는 방법이 있다. 이 방법은 해당 시스템에 어떠한 프로세스를 실행하지 않고도 분석관 컴퓨터로 물리 메모리 이미징을 수행할 수 있다는 측면에서 매우 유용하지만, 애플 제품과 다르게 해당 단자가 없는 경우가 상당히 많기 때문에 어느정도 한계를 가질 수 밖에 없다. 하지만 무결성 유지 측면에서 바라볼 경우 디지털 포렌식에 가장 적합한 방법이라 할 수 있다. firewire 단자를 이용한 데이터 수집은 'pyfw(<http://c0re.23.nu/c0de/pyfw/>)'를 이용하여 수집할 수 있다. pyfw 도구에는 메모리 조작을 통한 루트권한 획득이나 현재 화면 정보를 획득하는 코드, 그리고 메모리 덤프 코드가 데모로 제공되고 있으며, 이 중 메모리 덤프 코드(demo_dump_mem.py)의 소스 코드를 수정하여, 최대 크기를 수집 대상 시스템의 물리 메모리 크기와 맞춰주면, 대상 시스템에 루트 권한이 없더라도 쉽게 수집할 수 있다.

이를 보완하기 위해 생각해볼 수 있는 방법은 물리 메모리를 맵핑한 블록 디바이스인 '/dev/mem' 에 접근하여 직접 바이트를 읽어들이어 저장하는 방법이 있다. 보통 내부적으로 가지고 있는 'dd' 명령을 통해 물리 메모리 이미징을 수행하며, 외부 저장장치를 연결할 수 있는 단자가 존재한다면, 외부 저장 장치로 해당 데이터를 저장하고, 만약 존재하지 않는다면, nc 명령어로 네트워크를 통해 전송하는 방법을 생각해볼 수 있다. 이 방법은 기본으로 설치된 모든 BSD에 적용될 수 있는 유용한 방법

Memory Forensics for FreeBSD

이지만, 무결성 침해가 발생할 수 있으며, 외부 저장 장치 연결 단자가 존재하지 않아, 네트워크 전송 시 패킷 손실로 인한 수집된 데이터 손상으로 이어져 증거의 무결성 침해와 도구를 통한 분석이 원활하게 진행되지 않을 수 있다.

물리 메모리 분석을 위한 추가 정보 획득

FreeBSD의 물리 메모리 분석을 위해서는 물리 메모리 이미지 뿐만 아니라 주소 변환 테이블을 구성하기 위한 시작 주소 (Base Address)와 데이터 추출을 위한 구조체 정보와 같은 부가적인 정보가 필요하다. 이러한 정보를 얻기 위해서는 FreeBSD의 심볼(symbol) 정보가 필요하다. 심볼 정보는 본래 커널 디버깅과 같은 프로그래머에 대한 지원을 목적으로 주로 제공되고 있으며, 이 정보에는 특정 함수의 시작 주소 뿐만 아니라 'extern'한 함수 및 변수가 위치하는 커널 프로세스 상의 가상 주소 정보를 가지고 있어서, 주요 데이터를 지닌 구조체의 커널 가상 주소를 획득할 수 있다. 이에 물리 메모리 분석을 위해서는 커널의 이미지 파일이 필요하게 된다. 커널 이미지 파일은 '/boot' 디렉터리의 'kernel'이라는 파일 명으로 되어 있으며, 해당 파일을 추출하거나, 'objdump'나 'nm'과 같은 도구를 활용하여 심볼 정보만 따로 추출하는 과정이 필요하다. 추가적으로 로드된 커널의 아키텍처에 따라 구조체 정보와 커널 가상 주소가 변경될 수 있으므로, 커널이 로드된 아키텍처 정보를 획득해야 한다.

<위의 두가지 방법 뿐만 아니라 다양한 방법이 존재할 수 있으며, 이러한 방법은 계속되어 본 문서에 추가될 예정이다.>

OS Feature

메모리 포렌식 도구를 사용하기 이전에 FreeBSD의 프로세스 및 메모리 관리 기법에 대해 알아볼 필요가 있다. 본 절에선 FreeBSD에서 어떤 식으로 프로세스를 관리하며, 각 프로세스의 가상 메모리 영역을 확인하는 방법에 대해 알아본다. 추가적으로 루트킷 로드를 위해 많이 사용되는 외부 커널 모듈(KLD or LKM)에 대해 알아보고, 커널 이미지의 파일 포맷인 elf 포맷에 대한 소개와 심볼을 추출하는 과정을 알아본다.

운영체제의 프로세스 관리

FreeBSD는 유닉스 시스템이지만, 리눅스와 동일하게 더블 링크드리스트 형태로 프로세스를 관리한다. FreeBSD는 'kernel' 자기자신을 PID 0번 프로세스로 최초 생성한다. kernel 프로세스는 'proc' 구조체에 여러 정보를 가지게 되며, 해당 프로세스를 기준으로 fork()를 통해 생성된 프로세스를 더블 링크드 리스트(Double Linked List) 형태로 관리하고 있다.

```
463 struct proc {
464     LIST_ENTRY(proc) p_list;          /* (d) List of all processes. */
465     TAILQ_HEAD(, thread) p_threads; /* (c) all threads. */
466     struct mtx p_slock;               /* process spin lock */
467     struct ucred *p_ucred;           /* (c) Process owner's identity. */
468     struct filedesc *p_fd;           /* (b) Open files. */
469     struct filedesc_to_leader *p_fdtol; /* (b) Tracking node */
470     struct pstats *p_stats;          /* (b) Accounting/statistics (CPU). */
471     struct plimit *p_limit;          /* (c) Process limits. */
472     struct callout p_limco;          /* (c) Limit callout handle */
473     struct sigacts *p_sigacts;       /* (x) Signal actions, state (CPU). */
474
475     /*
476      * The following don't make too much sense.
477      * See the td_ or ke_ versions of the same flags.
478      */
479     int p_flag;                      /* (c) P_* flags. */
480     enum {
481         PRS_NEW = 0,                /* In creation */
482         PRS_NORMAL,                 /* threads can be run. */
483         PRS_ZOMBIE
484     } p_state;                      /* (j/c) S* process status. */
485 }
```

Proc Structure (sys/sys/proc.h, FreeBSD 8 Stable)

proc 구조체를 확인하면 첫 번째 항목이 모든 프로세스의 목록 정보를 의미하며, 이를 통해 프로세스의 링크 체인을 추적할 수 있다. LIST_ENTRY는 다음과 같은 구조를 가지고 있다.

```
347 #define LIST_ENTRY(type)
348 struct {
349     struct type *le_next; /* next element */
350     struct type **le_prev; /* address of previous next element */
351 }
```

LIST_ENTRY Structure (sys/sys/queue.h, FreeBSD 8 Stable)

커널 프로세스 또한 동일한 방법으로 프로세스 목록을 획득한다.

운영체제의 메모리 관리 방식

프로세스의 메모리 영역을 획득하는 과정을 설명하기 전에 운영체제의 메모리 관리 방식을 확인해보도록 한다. 본 글의 대상 시스템인 인텔 아키텍처는 각 프로세스만의 메모리 영역을 사용할 수 있도록하며, 이 메모리 영역을 가상 메모리라고 한다. 인텔 32비트 환경에서는 4기가의 가상 메모리 영역을 제공한다. 그렇다면, 물리 메모리가 512 또는 128 메가만 존재하는 시스템에서는 이러한 가상 메모리를 어떻게 관리하는지 의문을 가질 수 있다.

프로세스의 명령을 수행하기 위해선 해당 프로세스를 Context Switching을 통해 활성화 시키고, 필요한 가상 메모리 영역의 페이지를 물리 메모리에 맵핑하여, 해당 코드를 CPU가 읽어들이 처리하는 구조를 가진다. 인텔 32비트 CPU인 경우엔 4KB의 페이지 크기를 가지며, 만약 1바이트의 데이터를 읽더라도, 4KB의 페이지 전체를 물리 메모리에 맵핑하고, 데이터를 읽어들이는 구조를 가지고 있다.

인텔 아키텍처 문서를 확인하면, 이러한 가상 메모리를 물리 메모리에 맵핑하기 위해 Linear-Address Translation 기법을 이용하며, 32비트 환경의 주소 변환 테이블을 확인하면 다음과 같다.

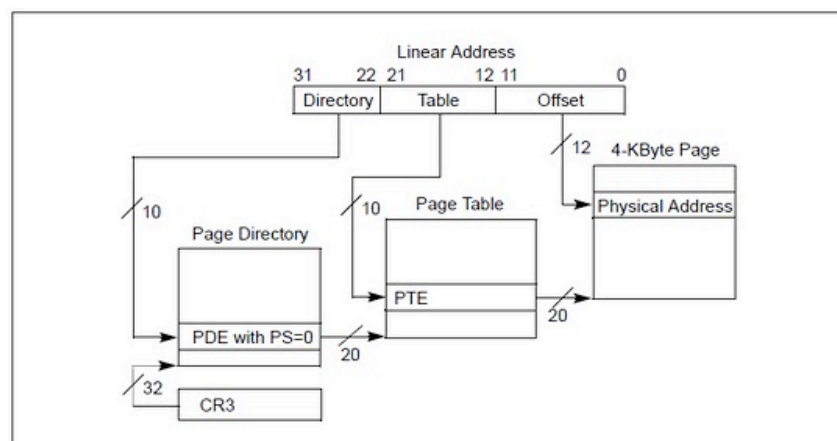


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

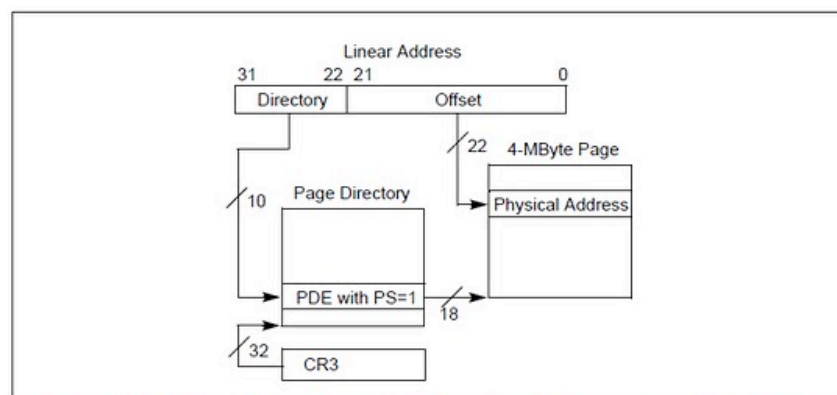


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

32bit Linear-Address Translation (Intel Architecture 3-A)

Memory Forensics for FreeBSD

그림의 'Figure 4-2'를 확인하면, 32비트의 가상 주소를 물리 주소로 변환하는 과정이 나타나며, 만약 'Present' 플래그가 0 (Disable)이면, Page fault를 발생시키고, 디스크에 존재하는 해당 페이지를 물리 메모리에 맵핑시키고 데이터에 접근한다. 각 각의 프로세스는 자기 자신의 주소 변환 테이블의 시작 위치를 가리키는 CR3 레지스터의 값을 프로세스 구조체에 연결하여 관리하며, Context Switching이 발생할 때마다 해당 프로세스의 저장한 CR3 레지스터 정보를 기준으로 주소 변환 테이블을 생성한다. 주소 변환에 관한 세부적인 내용은 한지성님 (xnetblue) 의 '*virtual to physical memory address translation*'를 읽어보기 바란다.

프로세스가 사용하는 메모리 영역 관리

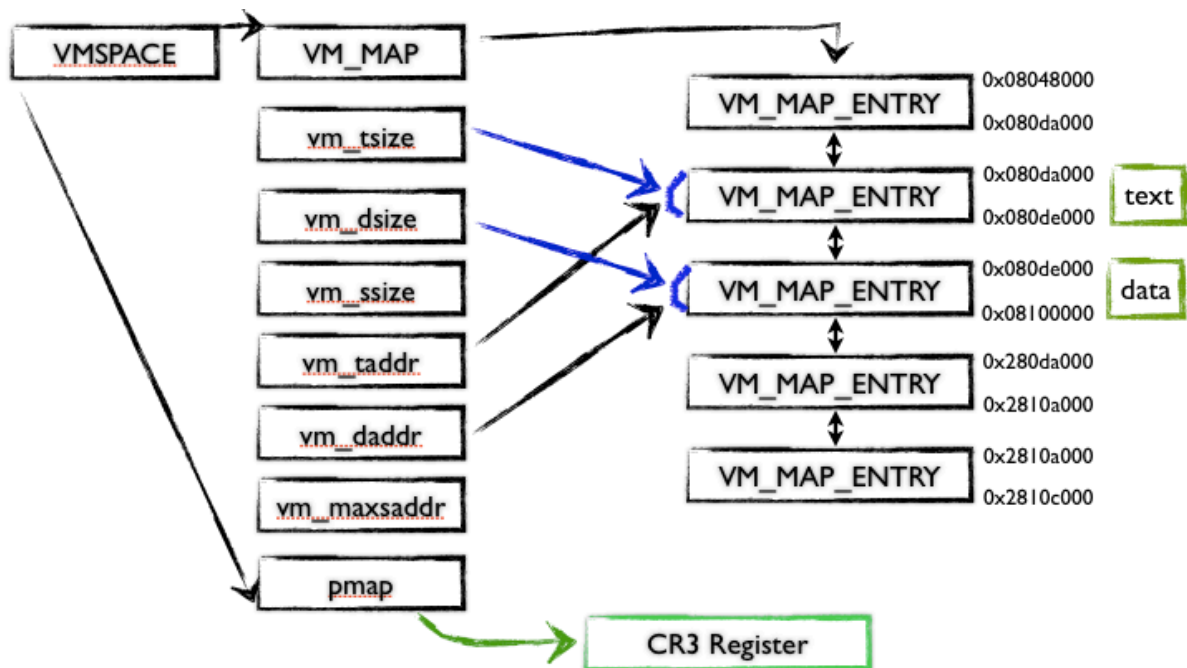
이제 각 프로세스의 가상 메모리를 어떠한 방식으로 관리하는지 확인할 필요가 있다. 우선 프로세스 구조체에 어떠한 필드가 메모리 관리 구조체와 연결되어 있는지 확인해야 한다. proc 구조체를 확인하면, vm_space 구조체의 포인터를 가진 필드를 확인할 수 있다. 해당 필드를 따라가면 다음과 같은 구조체를 확인할 수 있다.

```
229 struct vm_space {
230     struct vm_map vm_map; /* VM address map */
231     struct shmmap_state *vm_shm; /* SYS5 shared memory private data XXX */
232     seqsz_t vm_swrss; /* resident set size before last swap */
233     seqsz_t vm_tsize; /* text size (pages) XXX */
234     seqsz_t vm_dsize; /* data size (pages) XXX */
235     seqsz_t vm_ssize; /* stack size (pages) */
236     caddr_t vm_taddr; /* (c) user virtual address of text */
237     caddr_t vm_daddr; /* (c) user virtual address of data */
238     caddr_t vm_maxsaddr; /* user VA at max stack growth */
239     int vm_refcnt; /* number of references */
240     /*
241     * Keep the PMAP last, so that CPU-specific variations of that
242     * structure on a single architecture don't result in offset
243     * variations of the machine-independent fields in the vm_space.
244     */
245     struct pmap vm_pmap; /* private physical map */
246 };
```

Virtual Memory Management Structure (sys/vm/vm_map.h FreeBSD 8)

vm_map 구조체는 실제 프로세스가 사용하는 가상 메모리 영역의 시작 주소와 크기 정보를 더블 링크드 리스트 형태로 관리하고 있다. pmap 구조체는 CR3 레지스터와 같은 해당 프로세스의 주소 변환 테이블의 시작 주소 정보를 가지고 있으며, vm_taddr, vm_tsize와 같은 필드 정보를 통해 가상 메모리 영역 중 어느 부분이 코드 영역이고, 데이터 영역이며, 스택영역인지 확인할 수 있다.

본 구조체를 그림으로 표현하면 다음과 같이 나타낼 수 있다.



Example - VMSPACE Structure

VM_MAP_ENTRY에 표현된 가상 메모리 영역은 해당 프로세스의 CR3 레지스터의 값을 토대로 재구성된 주소 변환 테이블을 통해 실제 물리 주소로 접근하여 데이터를 추출해야 한다. 이러한 구조는 리눅스에서도 비슷한 구조를 가지고 있으며, 윈도우의 경우 가상 메모리 영역을 VAD Tree 형태로 구성한다. 실제 vm_map_entry 구조체를 확인하면 다음과 같다.

```

99 struct vm_map_entry {
100     struct vm_map_entry *prev;          /* previous entry */
101     struct vm_map_entry *next;          /* next entry */
102     struct vm_map_entry *left;          /* left child in binary search tree */
103     struct vm_map_entry *right;         /* right child in binary search tree */
104     vm_offset_t start;                  /* start address */
105     vm_offset_t end;                    /* end address */
106     vm_size_t avail_ssize;              /* amt can grow if this is a stack */
107     vm_size_t adj_free;                  /* amount of adjacent free space */
108     vm_size_t max_free;                  /* max free space in subtree */
109     union vm_map_object object;          /* object I point to */
110     vm_offset_t offset;                  /* offset into object */
111     vm_eflags_t eflags;                  /* map entry flags */
112     vm_prot_t protection;                /* protection code */
113     vm_prot_t max_protection;            /* maximum protection */
114     vm_inherit_t inheritance;            /* inheritance */
115     int wired_count;                     /* can be paged if = 0 */
116     vm_pindex_t lastr;                   /* last read */
117     struct uidinfo *ui;                  /* tmp storage for creator ref */
118 };

```

vm_map_entry Structure

vm_map_entry 내부에는 이전 엔트리와 다음 엔트리의 포인터 주소를 가지고 있는 더블 링크드 리스트 형태로 구성되어 있으며, 맨 처음의 이전 주소는 자기자신을 가리키도록 구성되어 있다. 맨 끝의 가상 주소 영역은 다음 vm_map_entry를 맨 처음 가상 주소 엔트리를 가리키도록 한다.

각각의 구조체 내부에는 해당 프로세스가 사용하는 가상 주소 영역의 시작 주소와 끝 주소 정보를 담고 있으며, 읽기, 쓰기, 실행 정보를 가지는 `eflags` 정보와 같은 가상 메모리 맵의 권한 정보를 확인할 수 있다. 이를 통해 앞서 `vm_map` 구조체의 코드 영역과 데이터 영역의 정보와 가상 맵 엔트리의 속성 정보로 코드 영역에 쓰기 권한이 설정되어 있는지와 같은 악성코드 분석 관점의 유용한 정보를 획득할 수 있다.

`pmap` 구조체는 물리 메모리 맵핑 정보를 담고 있으며, CPU 의존적인 구조로 32비트 환경의 경우 `'i386/include/pmap.h'` 파일을 통해 구조체의 내용을 파악할 수 있다.

```
429 struct pmap {
430     struct mtx      pm_mtx;          pm_mtx;
431     pd_entry_t      *pm_pdir;        /* KVA of page directory */
432     TAILQ_HEAD(,pv_chunk) pm_pvchunk; /* list of mappings in pmap */
433     cpumask_t        pm_active;       /* active on cpus */
434     struct pmap_statistics pm_stats;  /* pmap statistics */
435     LIST_ENTRY(pmap) pm_list;         /* List of all pmaps */
436 #ifdef PAE
437     pdpt_entry_t     *pm_pdpt;       /* KVA of page director pointer
438                                     table */
439 #endif
440     vm_page_t        pm_root;        /* spare page table pages */
441 };
```

pmap Structure

구조체에는 페이지 디렉터리의 가상 주소 포인터 정보와 해당 프로세스가 CPU에 의해 현재 작동 중인지, 그리고 모든 물리 메모리 맵핑 정보를 더블 링크드 리스트 형태로 관리하고 있다. 앞선 그림에서 `pmap`에는 `CR3` 레지스터 정보를 가지고 있다고 표현되어 있었다. 이는 그냥 32비트 주소 변환 환경과 32비트에 PAE가 설정된 시스템에 따라 다른 데이터를 참조하도록 구성되어 있다.

만약 32비트 환경의 물리 메모리 이미지라면, 페이지 디렉터리부터 시작되므로, `pm_pdir`를 커널의 주소 변환 테이블을 통해 실제 물리 메모리에 접근하여, 4바이트의 프로세스 `CR3` 레지스터 정보에 접근할 수 있다. 만약 PAE가 활성화 되어 있다면, 페이지 디렉터리 포인터 테이블에서부터 테이블 구성이 이루어지므로, `pm_pdpt` 정보를 이용하여 프로세스의 테이블을 구성하여야 한다.

외부 커널 모듈 관리

소개

커널 구조는 크게 마이크로 커널(micro kernel)과 모놀리틱 커널(Monolithic kernel)로 나눌 수 있다. 마이크로 커널은 프로세스 스케줄링과 메모리 관리와 같은 커널의 코어 기능을 토대로 별도의 모듈과의 메시지 교환을 통해 커널의 기능을 수행하는 구조를 말하며, 모놀리틱 커널은 하나의 커널 내부에 여러 하위 모듈이 내제되어 있어 내부적으로 데이터를 교환하는 형태로 되어 있다. 기존 리눅스 커널은 모놀리틱 커널 형태로 구성되어, 시스템에 새로운 모듈을 추가하려면 커널을 재 컴파일하는 과정이 필요하였다. 하지만, 시스템 재부팅이라는 서비스 측면의 크리티컬한 문제로 인해 최근의 대부분 운영체제의 커널은 마이크로 커널 형태로 운영되고 있다. 마이크로 커널에 연동하는 외부 커널 모듈은 LKM, KEXT, KMOD, KLD와 같이 운영체제 별로 다양하게 부르지만, FreeBSD의 현재 모듈 명칭은 KLD (Dynamic Kernel Linker) 을 사용하므로, 본 글은 LKD로 칭하도록 하겠다.

FreeBSD의 KLD는 운영체제의 베이스 커널의 기능을 확장하기 위한 코드를 가지는 객체 파일이다. 대부분의 유닉스 시스템이 지원하며, 리눅스에서도 디바이스 드라이버를 서비스에 등록하는 등 동적으로 커널을 확장시키는 기능을 제공한다. KLD는 새로운 하드웨어나 파일시스템 드라이버를 추가할 수 있으며, 시스템 콜과 같은 함수를 추가할 수도 있다. KLD는 자기 자신이 더이상 필요하지 않으면, 메모리 관리를 위해 언로드할 수 있기 때문에, 자원 관리 측면에서 유용한 기능이다.

이러한 특징을 가지는 KLD는 FreeBSD에서도 동일한 기능을 수행하며, 커널과 데이터를 교환하거나, 커널 레벨에서 동작할 수 있는 특징, 그리고 자동 실행이 가능하다는 측면을 이용하여, 루트킷 코드를 KLD에 심어 동작하도록 구성한다. FreeBSD는 리눅스와 동일한 명령어로 모듈을 관리한다. 주요 명령어는 다음과 같다.

- `lsmod`: 현재 커널에 로드된 모듈 목록을 보여주는 명령어로, 일반적으로 `/proc/modules` 파일에 기록된 내용과 동일하다. 해당 명령과 파일은 내부적으로 시스템 콜을 이용하여, 그 결과를 출력하는 구조로 되어있기 때문에, 이미 침해된 시스템의 변조된 시스템 콜로는 해당 루트킷을 탐지할 수 없는 문제점을 가진다.
- `insmod`: 외부 커널 모듈을 로드하는 명령어로, 해당 명령을 수행하면, 모듈 리스트에 해당 모듈이 추가된다. 루트킷은 보통 루트 권한을 획득한 공격자가 해당 명령어를 통해 등록하는 것이 일반적이다.
- `rmmod`: 외부 커널 모듈을 제거하기 위한 명령어이다.

커널 모듈 정보 관리

커널은 모듈 목록 관리를 구조체로 관리한다. 커널 소스의 `'sys/linker.h'`에 해당 구조체의 정보가 기록되어 있다.

```
70 struct linker_file {
71     KOBJ_FIELDS;
72     int refs; /* reference count */
73     int userrefs; /* kldload(2) count */
74     int flags;
75 #define LINKER_FILE_LINKED 0x1 /* file has been fully linked */
76     TAILQ_ENTRY(linker_file) link; /* list of all loaded files */
77     char* filename; /* file which was loaded */
78     char* pathname; /* file name with full path */
79     int id; /* unique id */
80     caddr_t address; /* load address */
81     size_t size; /* size of file */
82     int ndeps; /* number of dependencies */
83     linker_file* deps; /* list of dependencies */
84     TAILQ_HEAD(, common_symbol) common; /* list of common symbols */
85     TAILQ_HEAD(, module) modules; /* modules in this file */
86     TAILQ_ENTRY(linker_file) loaded; /* preload dependency support */
87     int loadcnt; /* load counter value */
88
89     /*
90      * Function Boundary Tracing (FBT) or Statically Defined Tracing (SDT)
91      * fields.
92      */
93     int nenabled; /* number of enabled probes. */
94     int fbt_nentries; /* number of fbt entries created. */
95     void* sdt_probes;
96     int sdt_nentries;
97     size_t sdt_nprobes;
98     size_t sdt_size;
99 };
```

linker_file Structure (sys/linker.h)

linker_file 구조체에는 로드된 모듈의 명칭과, 로드한 모듈의 위치한 경로 정보, 그리고 고유 id 정보, 해당 모듈이 로드된 커널 가상 주소의 시작 위치와 크기, 그리고 더블 링크드 리스트 형태의 모듈 목록을 가지고 있다.

물리 메모리 분석을 통한 외부 커널 모듈 정보 습득

외부 커널 모듈은 심볼 정보 중 linker_files 심볼을 통해 획득할 수 있다. 해당 심볼의 가상 주소를 물리 주소로 변환하여 확인하면, 첫 번째 외부 모듈의 정보를 획득할 수 있다. FreeBSD의 첫 번째 외부 커널 모듈 정보는 커널 그 자신이므로, 이를 통해 올바르게 외부 모듈 정보에 접근하였는지 확인할 수 있다. 첫 번째 외부 커널 모듈의 정보를 획득하면, linker_file 구조체는 링크드 리스트 형태이므로, 해당 내용을 추적하여 외부 모듈의 로드된 커널의 가상주소 정보를 물리 메모리 주소로 변환하여 기록된 크기만큼 덤프하여 외부 모듈을 덤프할 수 있다. 이러한 습득 방법은 물리 메모리 상의 구조체 정보를 직접 획득함으로써 시스템 콜 후킹과 같은 루트킷의 은닉 기법을 우회하여 데이터를 획득할 수 있다.

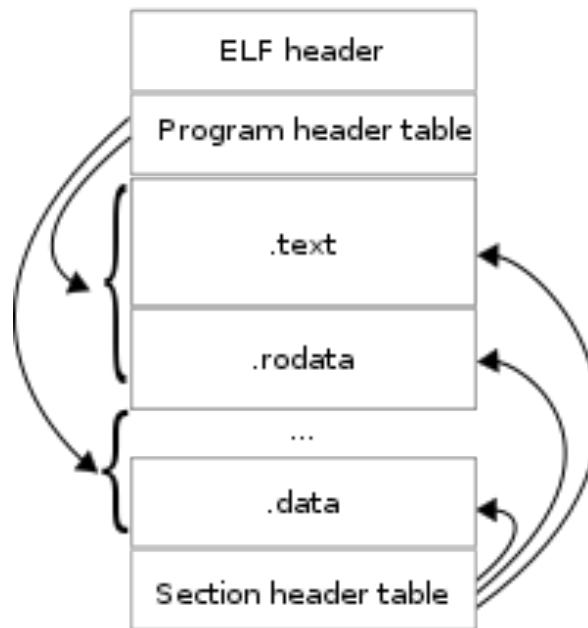
커널 이미지 분석

리눅스의 커널 이미지는 ELF 파일 포맷으로 이루어져있으며, 해당 포맷은 공개된 구조이기 때문에, 문서를 통해 그 내용을 확인할 수 있다. 본 절에서는 해당 파일 포맷에 대한 간단한 소개와 심볼을 추출하는 과정에 대해서만 논하도록 한다.

ELF 파일 포맷

ELF(Executable and Linkable Format) 파일 포맷은 리눅스와 유닉스 운영체제의 실행 파일이나 객체 코드, 공유 라이브러리, 그리고 코어 덤프에도 사용되는 표준 파일 포맷이다. 해당 포맷은 1999년도에 Unix 시스템에서 채용하였으며, 86open 프로젝트에 의해 x86 기반의 유닉스 또는 유닉스와 유사한 운영체제에 표준 바이너리 파일 포맷으로 채용되었다. ELF는 다양한 프로세서나 아키텍처에 구애받지 않는 유연성을 지니고 있으며, 많은 운영체제와 다양한 플랫폼을 지원하는 확장성을 지니고 있다. 또한 ELF 파일 포맷은 임베디드 시스템에서도 사용된다.

ELF 포맷의 구조는 간단하게 다음과 같은 형태를 가진다.



elf format layout

ELF 헤더는 바이너리가 동작하기 위한 아키텍처 정보와 다음에 분석을 위한 '프로그램 헤더 테이블'의 파일 오프셋 정보(e_phoff)와 섹션 헤더 테이블의 파일 오프셋 정보(e_shoff)와 각 테이블 내부의 엔트리 갯수 및 크기 정보를 가지고 있다.

```
#define EI_NIDENT      16

typedef struct {
    unsigned char    e_ident[EI_NIDENT];
    Elf32_Half       e_type;
    Elf32_Half       e_machine;
    Elf32_Word       e_version;
    Elf32_Addr       e_entry;
    Elf32_Off        e_phoff;
    Elf32_Off        e_shoff;
    Elf32_Word       e_flags;
    Elf32_Half       e_ehsize;
    Elf32_Half       e_phentsize;
    Elf32_Half       e_phnum;
    Elf32_Half       e_shentsize;
    Elf32_Half       e_shnum;
    Elf32_Half       e_shstrndx;
} Elf32_Ehdr;
```

ELF file format header

심볼 정보는 섹션 테이블을 분석하여 획득할 수 있는데, 섹션 엔트리 정보 중 심볼의 주소를 가지는 섹션과 심볼 문자열 테이블을 가지는 섹션에 대한 추출을 해야, 각 가상주소에 어떠한 심볼이 위치하는지 확인할 수 있다.

```
typedef struct {
    Elf32_Word    sh_name;
    Elf32_Word    sh_type;
    Elf32_Word    sh_flags;
    Elf32_Addr    sh_addr;
    Elf32_Off     sh_offset;
    Elf32_Word    sh_size;
    Elf32_Word    sh_link;
    Elf32_Word    sh_info;
    Elf32_Word    sh_addralign;
    Elf32_Word    sh_entsize;
} Elf32_Shdr;
```

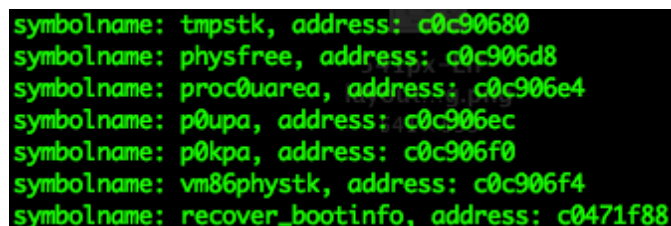
Section header

각 섹션의 타입은 섹션 엔트리 정보 중 두 번째 데이터인 sh_type 를 통해 획득할 수 있다. 타입은 WORD 형태로 표현되어 있으며, 이 중 심볼 정보인 SHT_SYMTAB(0x02)과 심볼에 매칭되는 심볼 문자열 테이블을 가진 SHT_STRTAB(0x03) 정보를 획득해야 한다.

```
typedef struct {
    Elf32_Word    st_name;
    Elf32_Addr    st_value;
    Elf32_Word    st_size;
    unsigned char st_info;
    unsigned char st_other;
    Elf32_Half    st_shndx;
} Elf32_Sym;
```

Symbol Table Entry Structure

심볼 정보는 위의 구조체와 같은 형태로 연속적으로 배치되어 있으며, 섹션 헤더의 사이즈 정보나, 심볼 구조체 전체가 0x00 인 경우를 체크하여 그 끝을 확인할 수 있다. 구조체 정보 중 st_name은 심볼 문자열 테이블 목록 중 몇 번째 인덱스인지에 대한 정보를 담고 있으며, st_value를 통해 커널 메모리 상의 가상 주소를 표현하고 있다. 심볼 문자열 테이블은 각각의 인덱스 번호와 문자열의 쌍으로 이루어져 있으며, 이를 맵으로 구성하여 체크하면, 각 심볼 정보의 가상 주소와 해당 가상 주소에 맵핑되는 심볼 문자열 정보를 확인할 수 있다.



```
symbolname: tmpstk, address: c0c90680
symbolname: physfree, address: c0c906d8
symbolname: proc0uarea, address: c0c906e4
symbolname: p0upa, address: c0c906ec
symbolname: p0kpa, address: c0c906f0
symbolname: vm86phystk, address: c0c906f4
symbolname: recover_bootinfo, address: c0471f88
```

추출한 심볼 정보

volafunx(volafox for Unix)

앞의 구조를 토대로 구현된 본 저자의 분석 도구는 다음의 기능을 가진다.

- 프로세스 목록 출력/덤프 - 커널 프로세스 구조체를 가리키는 심볼 정보를 토대로 프로세스 체인을 따라가며 목록을 추출한다. 추출한 프로세스 구조체 정보를 가지고 실제 프로세스의 물리 메모리 영역을 덤프한다.
- 외부커널모듈 목록 출력/덤프 - LKM 목록을 추출하고, 해당 영역을 덤프한다. 로드된 커널 모듈은 커널의 메모리 영역에 위치하므로, 별도의 CR3 레지스터 정보가 필요하지 않다.
- 시스템 콜 목록 출력/후킹 여부 탐지 - 심볼 정보를 토대로 시스템 콜 테이블 구조체를 분석하여 정보를 표현한다. 커널 이미지 내부의 시스템 콜 자체 심볼과 가상주소를 이용하여, 물리 메모리에서 추출한 테이블 구조체의 가상 주소와 매칭, 시스템 콜 후킹 여부를 탐지한다.

분석 테스트된 시스템은 FreeBSD 7,8 버전이며, 타 버전에서도 가능할 것으로 생각된다. 본 도구를 작동시키기 위해선 파이션이 설치된 인텔 기반의 모든 운영체제에서 구동할 수 있다.

대부분의 기능은 Mac OS X 물리 메모리 분석 도구인 volafox 를 토대로 구성되어 있으며, 두 모듈이 안정화되는데로 하나의 프레임워크로 재통합하는 과정을 거칠 예정이다.

프로세스 목록 출력

앞 절의 정보를 토대로 커널 이미지 내부의 심볼 정보에서 프로세스 링크 체인을 관리하는 심볼의 가상 주소에 접근하여 구조체 정보를 수집한다. 프로세스 체인을 토대로 분석하며, 시스템 콜 후킹을 통한 프로세스 은닉을 우회하여 결과물을 수집할 수 있다.

```
n0fate:volafunx n0fate$ python volafunx.py -i FreeBSD.vmem -s kernel -o proc_info
[+] Memory Image: FreeBSD.vmem
[+] Kernel Image: kernel
[+] Information: proc_info

-- process list --
list_entry_next  pid      ppid      process name  username
c1946000         0         0      kernel
c1945aa0         1         0      initl      root
c19452a8         2         0      g_event
c1945000         3         0      g_upl
...
c1ea0d48        907        11      getty      root
c1ea0aa0        908        11      getty      root
c1c6dd48        909        11      getty      root
c0d99b58        923        902      bash       root
```

Memory Forensics for FreeBSD

proc 구조체 내부에는 부모 프로세스의 PID인 PPID가 존재하는 것이 아닌 부모 프로세스의 proc 구조체의 위치를 커널 가상 주소로 표현하고 있기 때문에, 해당 체인을 추적하여 부모의 PID를 획득하여야 한다. 사용자 이름은 proc에 존재하지 않으며, proc 구조체 내부의 프로세스 그룹 정보인 pgrp의 포인터를 따라가서 해당 구조체 내부의 session 구조체를 추적하여, 프로세스를 실행한 사용자의 이름을 획득할 수 있다. 이외에도 프로세스 구조체에는 스레드 체인이나, 프로세스 실행 시간 정보와 같은 다양한 정보를 가지고 있으므로, 도구의 기능 확장을 통해 많은 정보를 수집할 수 있다.

프로세스 덤프

앞의 프로세스 목록 출력 기능을 토대로 추출할 프로세스의 CR3 레지스터 정보를 가지고 주소 변환 테이블을 구성하여, 프로세스가 사용하는 가상 주소 영역을 물리 메모리 상의 주소로 변환하여 덤프를 수행한다. 각 VM_MAP_ENTRY 마다 하나의 파일을 생성하며, 파일에는 가상 주소의 시작 주소와 끝 주소를 명시한다.

```
n0fate:volafunx n0fate$ python volafunx.py -i FreeBSD.vmem -s kernel -x 923[+] Memory Image: FreeBSD.vmem
[+] Kernel Image: kernel
[+] Dump PID: 923
[+] Dump Process ID: 923
[-] process name: bash
[-] vmSPACE: c194b3a0
[-] start: 1000
[-] end: bfc00000
[-] VAD CR3: c63a6000
[-] PHYS CR3: 32e0000
[+] END VAD DUMP LIST
[+] VAD DUMP START
[-] [DUMP] Image Name: bash-8048000-80da000
[-] [DUMP] Image Name: bash-80da000-80de000
[-] [DUMP] Image Name: bash-80de000-8100000
[-] [DUMP] Image Name: bash-280da000-2810a000
[-] [DUMP] Image Name: bash-2810a000-2810c000
[-] [DUMP] Image Name: bash-2810c000-2811f000
[-] [DUMP] Image Name: bash-2811f000-2815c000
[-] [DUMP] Image Name: bash-2815c000-2815f000
[-] [DUMP] Image Name: bash-2815f000-28167000
[-] [DUMP] Image Name: bash-28167000-28168000
[-] [DUMP] Image Name: bash-28168000-2825d000
[-] [DUMP] Image Name: bash-2825d000-2825e000
[-] [DUMP] Image Name: bash-2825e000-2835a000
[-] [DUMP] Image Name: bash-2835a000-28360000
[-] [DUMP] Image Name: bash-28360000-28376000
[-] [DUMP] Image Name: bash-28400000-28500000
[-] [DUMP] Image Name: bash-bf8e0000-bfc00000
[+] VAD DUMP COMPLETE
n0fate:volafunx n0fate$
```

외부 커널 모듈 목록 출력

커널 이미지의 심볼 정보를 토대로 외부 커널 모듈의 단일 링크 체인을 따라 외부 커널 모듈 정보를 수집한다. 이렇게 수집한 목록 정보는 시스템 콜 후킹을 우회하여, 정보를 수집하므로, 외부 커널 모듈 은닉을 피해 정보를 수집할 수 있다.

```
n0fate:volafunx n0fate$ python volafunx.py -i FreeBSD.vmem -s kernel -o lkm_info
[+] Memory Image: FreeBSD.vmem
[+] Kernel Image: kernel
[+] Information: lkm_info

== loaded kernel driver information ==
id      refs    urefs   name   file path          lkm address      lkm size
1       1        0      kernel /boot/kernel/kernel c0400000          b6dfe0
n0fate:volafunx n0fate$
```

외부 커널 모듈 덤프

앞선 외부 커널 모듈 정보를 토대로 해당 커널의 가상 주소 영역을 덤프하여 하나의 파일로 출력한다. 실제 커널 모듈을 덤프하였기 때문에, IDA와 같은 디스어셈블리 도구를 통해 해당 내용을 분석할 수 있다. 단, 각 함수가 물리 메모리 상의 실제 함수의 주소 또는 시스템 콜 함수의 주소를 가리키기 때문에, 분석 시 이 점에 유의하여, 시스템 콜의 심볼 정보와 비교하는 과정이 필요하다.

```
n0fate:volafunx n0fate$ python volafunx.py -i FreeBSD.vmem -s kernel -m 1
[+] Memory Image: FreeBSD.vmem
[+] Kernel Image: kernel
[+] Dump Module: 1
[+] Find kernel module, offset: c0400000, size: b6dfe0
[+] Dump file name: kernel-c0400000-c0f6dfe0
[+] Module Dump Complete
n0fate:volafunx n0fate$
```

시스템 콜 목록 출력

시스템 콜 테이블을 가리키는 심볼 정보를 토대로 시스템 콜 구조체를 덤프하여 분석한 결과를 출력한다. 커널 이미지 내부의 심볼 정보와 매칭하여, 각 시스템 콜 번호의 함수가 어떠한 시스템 콜 함수인지 함수명을 표현한다.

```
n0fate:volafunx n0fate$ python volafunx.py -i FreeBSD.vmem -s kernel -o syscall_info
-= syscall list -=
count  arg      function  hooking detection
0       0       nosys     valid function
1       1       sys_exit  valid function
2       0       fork      valid function
3       3       read      valid function
4       3       write     valid function
```

시스템 콜 후킹 탐지

시스템 콜 후킹 탐지는 악성코드 분석에 꼭 필요한 기능으로 앞의 시스템 콜 추출을 수행하며 시스템 콜 함수의 주소가 바뀌었는지 체크한다. 함수의 포인터 주소가 변경된 경우는 함수 명이 해당 주소로 표현되며, 'system call hooking possible'이라는 메시지를 표시한다. 시스템 콜 함수의 갯수가 많기 때문에 분석 시에는 해당 명령 뒤에 `grep 'hooking'` 을 통해 시스템 콜 후킹이 발생된 함수만을 파악할 수 있다.

Appendix

volafox (<http://code.google.com/p/volafox>)

저자가 개발한 Mac OS X 물리 메모리 분석 도구로 운영체제의 기본 정보와 마운트 장치 정보, 그리고 프로세스 목록 출력 및 덤프, KEXT 목록 출력 및 덤프, 시스템 콜 목록 출력 및 시스템 콜 후킹 탐지 기능을 갖춘 디지털 포렌식 도구이다. 파이썬으로 개발되어 있으며, 지속적인 업데이트를 수행하고 있다.

Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A

인텔에서 제공하는 32비트 및 64비트 아키텍처에서 지원하는 다양한 기능에 대한 설명이 잘 정리된 문서이다. 해당 문서를 통해 가상 메모리에서 물리 메모리로의 주소 변환 방식을 파악할 수 있다.

An error 0xBAADC0DE occurred (<http://baadc0de.blogspot.com>)

윈도우 메모리 포렌식에 대해 꾸준히 연구한 고려대학교 디지털 포렌식 연구센터(Digital Forensic Research Center)의 한지성님이 운영하는 블로그로 윈도우 기반의 물리 메모리 분석에 대해 체계적으로 정리되고 있는 블로그이다. 또한 저자가 직접 윈도우 환경의 물리 메모리 수집 도구 중 가장 빠른 FastDD를 개발하기도 하였다. 물리 메모리 포렌식에 관심이 있는 분들은 꼭 찾아보길 권고하는 사이트

FreeBSD/Linux Cross Reference (<http://fxr.watson.org>)

리눅스 및 FreeBSD 그리고 Mac OS X의 커널인 XNU 커널 소스의 크로스 레퍼런스를 제공하는 사이트로, 분석 시스템이 별도로 갖춰지지 않은 상황에서 커널 소스 분석 시 매우 유용하게 사용할 수 있다.

Detecting Loadable Kernel Module(LKM) Rootkit

LKM 루트킷에 대한 기본적인 이해를 할 수 있는 문서, 한글로 되어 있어 해당 내용을 처음 접하는 분들에게 유용하다.

Designing BSD Rootkits

FreeBSD 운영체제의 루트킷을 개발하는 방법에 대해 정리된 책으로 350 페이지 정도의 작은 분량에 루트킷 개발에 대한 내용을 잘 담아두었다. 단 페이지 분량 탓인지 그 내용이 깊지 않고 인터넷 검색을 통해서 대부분 얻을 수 있는 정보이기 때문에, 좀더 편하게 BSD의 루트킷에 대한 정보를 획득하고 싶다면, 한번쯤 읽어볼만한 책이라 할 수 있다.

Executable and Linkable Format(ELF)

리눅스와 유닉스에서 사용되는 바이너리 파일 포맷인 ELF 파일 포맷에 대한 구조를 설명한 문서