The Way of Binary Copy without Permission

written by **hkpco**
http://hkpco.kr/
chanam.park@hkpco.kr

2007. 7

Contents

1. 들어가며 ---------- 3

2. Linux, FreeBSD 란? ---------- 4

3. 방법론 ---------- 5

4. ptrace의 소개 ---------- 5

5. 핵심코드 ---------- 8

6. Let's play hktrace ---------- 13

7. 마치며 ---------- 17

8. 참고자료 ---------- 17

1. 들어가며

유닉스계열 공부를 하던 중 우연히 재미있는 방법을 발견하게 됩니다.
혹시나 여기에 관련된 문서나 코드가 있을까 싶어 많은 검색을 해보았지만 아직까지 공개적으로 작성된 내용이 없다는 사실을 알게 되었습니다. 본 문서에서 소개할 기술은 32bit x86 Linux와 FreeBSD 환경에서 테스트 되었고, 기술에 대한 증명에 초점을 맞추었습니다. 이제부터 일반적으로 read permission이 없는 바이너리에 대한 복사가 불가능하다는 편견을 하나씩 제거해 나갈 것 입니다.

2. Linux, FreeBSD 란?

Linux는 리누스 토발스(Linus Tovals)가 처음 개발한 오픈소스 운영체제입니다.
FreeBSD는 4.4 BSD-Lite2에 기반한 BSD계열의 오픈소스 운영체제입니다.

이번 장에서 각 운영체제의 장점들에 대해 간략히 살펴보고 넘어가겠습니다.
두 운영체제의 공통된 장점들은 다음과 같습니다.

- 라이센스가 자유로움
- 메모리 보호기능이 뛰어남
- 다양한 플랫폼에서 지원이 가능
- 다중 사용자, 다중 처리 시스템을 지원
- 안정적이고 신뢰성이 높으며 이식성이 좋음
- 가볍고 안정적이어서 저사양 컴퓨터에서도 사용이 용이함
- 웹서버, 메일서버, 프린터서버 등 다양한 서비스 제공이 가능
- 소스코드가 공개되어있어 빠른 업데이트와 계속적인 개발이 이루어짐
- 다른 운영체제와는 달리 오픈소스로 인한 커널수준의 강력한 보안패치가 가능
- 무료이며 소스코드가 공개되어있어 프로그래머가 원하는 대로 수정과 보완이 가능

공통된 장점들이 많은데, 두 운영체제에 대한 별개의 장점 또한 존재합니다.

Linux의 대표적인 장점 중 하나는 다음과 같습니다.
- 다양한 배포본으로 취향에 맞는 선택이 가능

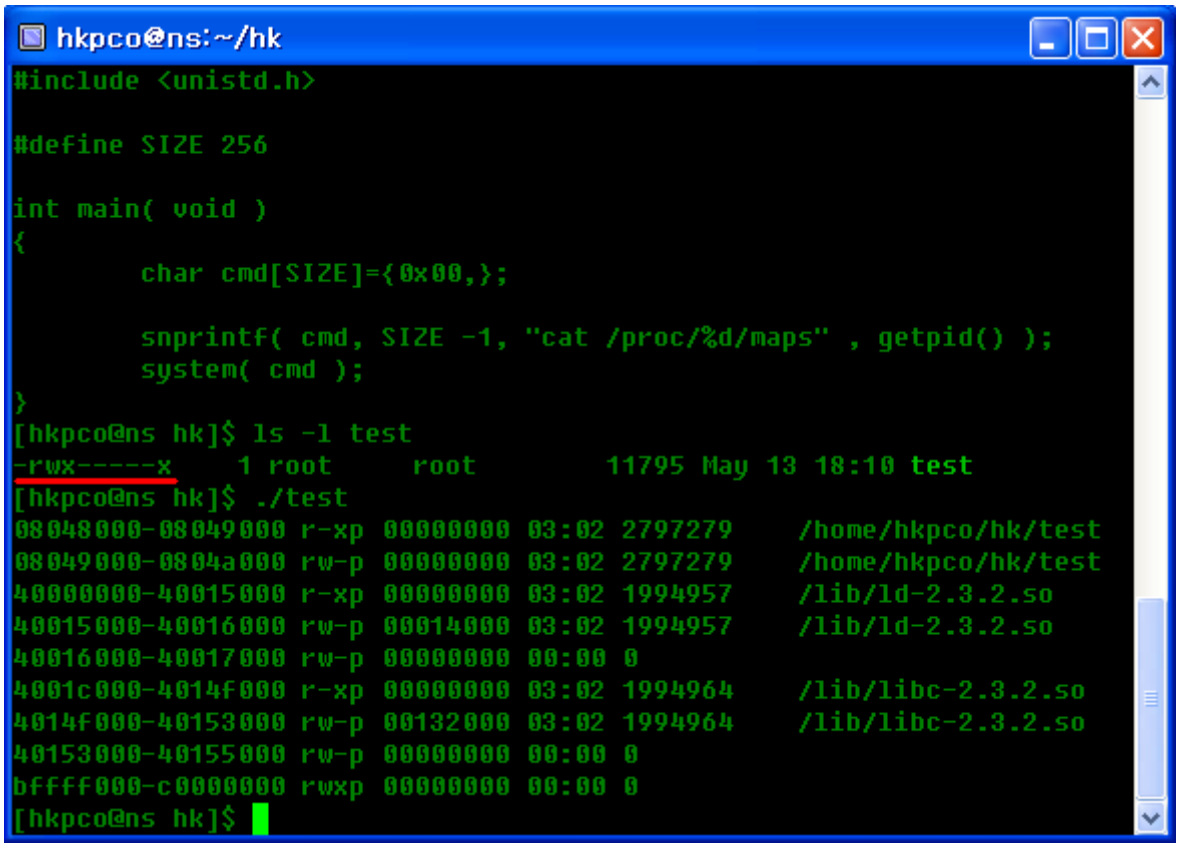
FreeBSD의 대표적인 장점 중 하나는 다음과 같습니다.
- 정통 유닉스 운영체제로, 유닉스 운영체제의 모든 특징을 포함

3. 방법론

일반적으로 read permission이 없다면 바이너리의 복사가 이루어 지지 않습니다.
대상 바이너리가 다른 사용자 소유이며 읽기권한이 주어지지 않았다면, 해당 바이너리의 내용을 읽어 들일 수 없으므로 복사를 할 수 없습니다. 본 문서에서 소개할 내용은 바이너리 자체의 읽기권한을 이용한 일반적인 복사수행의 편견을 제거해 나갈 것입니다.

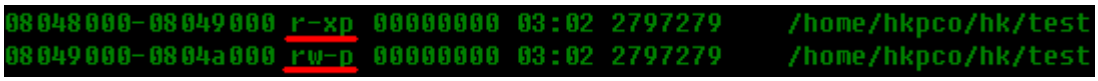
Linux/FreeBSD 환경에서 바이너리가 실행되면 실행구조에 알맞게 가상메모리의 텍스트, 데이터 세그먼트 등에 데이터들이 사상 되게 됩니다. 간단한 테스트를 통하여 일반사용자가 실행권한만이 주어진 프로그램을 실행했을 때의 메모리 사상상태를 살펴보겠습니다.

▶ 작업환경은 Linux system 위주로 진행됩니다.



<그림1> 바이너리 실행 시 메모리 사상상태

<그림1>에서 test.c는 test 바이너리(자기자신)가 실행될 때의 메모리 사상상태를 출력하는 것으로 해당 정보는 /proc/[process id]/maps에 기록되어 있습니다.
(* FreeBSD에서는 /proc/[process id]/map입니다.)



<그림2> text, data segment의 사상상태

<그림1>에서 테스트한 root소유의 프로그램은 일반 사용자에게 읽기권한이 주어지지 않았음에도 불구하고 <그림2>에서 사상 되어있는 메모리 영역의 퍼미션에 모두 읽기권한이 존재하는 것을 확인할 수 있으며 이를 통해 우리는 복사를 위한 조건이 메모리 영역에도 존재한다는 것을 알 수 있습니다. 각 영역은 순서대로 텍스트와 데이터 세그먼트를 뜻하며, 세그먼트란 하나이상의 섹션을 포함한 연속적인 영역으로 섹션을 포괄하는 용어입니다.

바이너리가 실행 될 때 가상메모리에는 실행에 필요한 모든 내용이 적재되게 되고 그 대표적인 영역이 바로 텍스트와 데이터 세그먼트 입니다. 역으로 말해 이 영역의 데이터가 바이너리 자체라고 생각 할 수 있으며 만약 이 영역을 복사 할 수 있다면 바이너리에 읽기권한이 없어도 메모리에 적재되었을 때의 데이터를 가져와 복사가 가능하게 될 것 입니다.

4. ptrace의 소개

Unix 계열의 운영체제에서 제공하는 ptrace() 함수는 프로세스를 추적하는 일을 하며 각종 디버깅 툴에서 필수적으로 사용됩니다. 방법론에서 제시한 기술을 적용시키기 위해서는 ptrace() 함수의 기능이 중요한 역할을 하게됩니다.

아래는 리눅스의 Manual page에서 발췌한 ptrace() 함수의 헤더와 원형입니다.

```
#include <sys/ptrace.h>

long  ptrace(enum __ptrace_request request, pid_t pid, void *addr, void
*data);
```

<그림3> ptrace 함수의 원형

ptrace() 함수의 첫번째 인자에 해당하는 request값에 따라 다양한 작업의 수행이 가능하며 나머지 pid, addr, data의 인자 값들은 request에 따라 유동적으로 사용됩니다. 이제 아래에 나열된 세 종류의 request와 그에 따른 부가적인 인자들의 사용 및 실제 적용 방법을 익혀 나가겠습니다.

PTRACE_TRACEME
PTRACE_PEEKTEXT
PTRACE_PEEKDATA

PTRACE_TRACEME

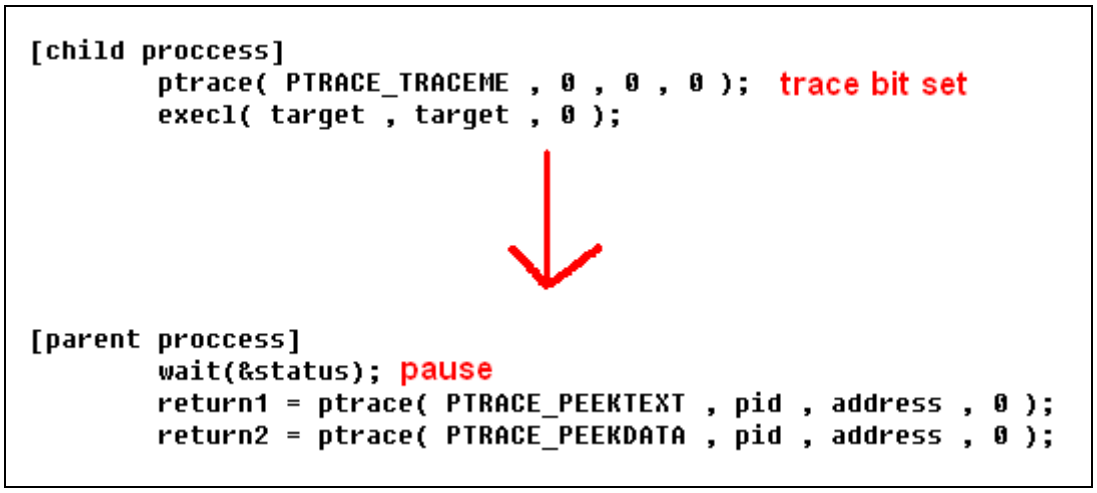
자신의 부모 프로세스에 의해 추적 되어진다는 것을 가리킵니다.

이 프로세스에서의 exec 호출은 부모 프로세스에게 제어권을 주기위해 SIGTRAP 시그널을 발생시킵니다. 시그널이 발생하면 실행을 정지하고 wait함수를 통해 부모에게 통보합니다. 해당 request 사용 시 pid, addr, data인자는 무시됩니다.

PTRACE_PEEKTEXT, PTRACE_PEEKDATA

자식 프로세스 메모리에서 주소값(addr인자)의 word를 읽어서 ptrace 시스템콜의 결과로 반환합니다. 해당 request는 각각 text, data segment에 적용할 목적으로 설계되었지만 리눅스(또는 FreeBSD)는 text와 data 주소 공간을 분리하지 않으므로 현재 두 request는 같습니다. 해당 request 사용 시 data인자는 무시됩니다.

앞서 소개한 ptrace()의 request 들은 본 문서에서 다음과 같은 작업을 위해 사용됩니다.



<그림4> 권한 없는 바이너리 복사에 필요한 ptrace의 수행과정

<그림4>의 작업을 중심으로 한 루틴의 수행과정은 다음과 같습니다.

1. *child process*에서 ptrace의 PTRACE_TRACEME request 사용
2. [1]에 의해 trace bit가 셋팅되어 *parent process*에게 추적 권한이 주어짐
3. execl 함수로 실행된 대상프로그램이 *child process*의 메모리영역에 적재
4. wait상태인 *parent process*가 *child process*에 의해 깬 뒤 PTRACE_PEEKTEXT, PTRACE_PEEKDATA request를 사용하여 ptrace 함수 인자에서 읽어 들인 *child process* 주소영역의 데이터 반환 값을 return1, 2 변수에 저장 뒤 파일에 기록
5. 대상 바이너리의 텍스트/데이터 세그먼트 영역 전체를 복사하기 위하여 텍스트 세그먼트 시작주소(0x08048000)부터 데이터 세그먼트의 마지막 주소까지 Nbyte 씩 증가시키며 [4]의 작업 루틴을 반복

5. 핵심코드

이번 장에서는 이제까지 알아본 지식을 기반으로 만들어진 프로그램의 핵심코드를 살펴보겠습니다. 복사 프로그램은 Linux/FreeBSD 기반으로 만들어졌으며 read permission이 없는 바이너리의 복사를 수행합니다. 작성한 소스코드의 길이가 문서에 기재하기에는 상대적으로 길고 비 효율적이기 때문에 복사 프로그램이 수행하는 핵심적인 코드와 그에 대한 설명을 기술하도록 하겠습니다.

핵심코드의 두 반복문은 각각 텍스트 세그먼트와 데이터 세그먼트를 복사한다는 것 외에는 기능과 목적이 동일하기 때문에 중복되는 부분은 제외하고 분석하도록 하겠습니다.

```
for( i = 0 ; i < size1 - VADR ; i+=ptx_sz , cnt1++ )
{
        dump = ptrace( PTRACE_PEEKTEXT , gcp , (DIFF_T)(VADR +i) , 0 );
        d_buf[cnt1] = dump;

        if( cnt1 == PAGE/ptx_sz -1 )
        {
                write( fd , &d_buf , sizeof(d_buf) );
                memset( d_buf , 0x0 , sizeof(d_buf) );
                cnt1 = -1;
        }
}
if( chk ) write( fd , &d_buf , (size1 % PAGE) );

for( i = 0 ; i < size2 - size1 ; i+=ptx_sz , cnt2++ )
{
        dump = ptrace( PTRACE_PEEKDATA , gcp , (DIFF_T)(size1 +i) , 0 );
        d_buf[cnt2] = dump;

        if( cnt2 == PAGE/ptx_sz -1 )
        {
                write( fd , &d_buf , sizeof(d_buf) );
                memset( d_buf , 0x0 , sizeof(d_buf) );
                cnt2 = -1;
        }
}
```

전체적으로 살펴보면 본 문서에서 소개한 ptrace() 함수의 기능들이 그대로 사용된 것을 알 수 있습니다. 위 소스코드는 두 부분의 반복문(for)으로 구성되어 있는데, 첫번째 반복문은 텍스트 세그먼트의 복사를 위한 것이며, 두번째 반복문은 데이터 세그먼트의 복사를 수행하기 위한 것입니다. 반복문 중간에 위치한 if문은 바이너리가 메모리에 사상된 크기를 얻지 못하고 대상 바이너리 크기로 대체되었을 때, 반복문에서 완료하지 못한 나머지 데이터의 기록을 위한 것이며 자세한 내용은 뒤에서 다루도록 하겠습니다. 또한, 작업의 성능을 위해 변수에 4KB씩 버퍼링 뒤, 저수준 함수(write)를 사용하여 기록하였습니다.

분석하게 될 핵심코드는 자식 프로세스에서 다음과 같은 작업이 수행되었다는 가정 하에 이루어 지며, 데이터 교환을 위한 프로세스간의 통신은 사용되지 않습니다.

```
childp = fork();
if( childp == 0 )
{
        ptrace( PTRACE_TRACEME , 0 , 0 , 0 );
        execl( target , target , NULL );
}
```

앞에서도 설명하였지만 위 코드는 자식 프로세스가 ptrace()의 PTRACE_TRACEME를 수행함으로써 추적권한을 열어주고, 해당 메모리 영역을 대상 프로그램으로 대체시킨 뒤, SIGTRAP 시그널을 발생시켜 부모 프로세스에게 제어권을 넘깁니다.

이제 몇 줄 안되지만 가장 골자가 되는 코드를 한 줄씩 분석해 보겠습니다. 코드 내용을 이해하고 있다면 넘어가도 좋습니다.

```
for( i = 0 ; i < size1 - VADR ; i+=ptx_sz , cnt1++ )
```

text segment 영역의 복사를 위한 부분으로 다음과 같은 변수와 define값이 사용됩니다.

1) #define VADR 0x08048000
2) int i, cnt1;
3) int size1, ptx_sz;

define으로 정의된 VADR 값인 0x08048000은 프로그램 실행 시 적재되는 가상메모리의 시작주소이며 텍스트 세그먼트의 시작주소와 동일합니다.

int형 변수인 i는 텍스트 세그먼트 주소의 마지막까지 카운터 하기 위해 선언된 것이고 또 다른 int형 변수 cnt는 나중에 살펴볼 코드에서 사용되는 변수로써 바이너리 복사의 속도를 위해 반복구문 안의 ptrace() 함수가 특정 메모리를 읽어 반환하는 데이터를 int형 버퍼공간에 저장하고, 4KB(0x1000)가 채워지게 되면 파일에 기록하는 작업을 반복하기 위해 사용되는 카운터 변수입니다. 이에 관한 설명은 다름 라인의 분석에서 더 자세히 다루겠습니다.

size1은 사용자 함수에 의해 값이 할당되는 변수로 텍스트 세그먼트의 마지막 주소를

담고 있으며, 시작 주소인 VADR(0x08048000)을 뺀 값이 텍스트 세그먼트 영역의 크기가 됩니다. 즉, 이 두 값의 차를 이용하여 복사할 메모리 영역의 크기를 구할 수 있습니다.

마지막 ptx_sz는, ptrace() 함수가 선언된 타입의 크기만큼 주소 값의 데이터를 읽어 반환하기 때문에 반복문에서 매번 그 크기만큼 주소 값을 증가시켜 ptrace() 함수 인자로 전달해야 합니다. 그렇기 때문에 sizeof 연산자를 이용하여 얻은 ptrace() 함수의 선언에서 사용되는 타입의 크기를 ptx_sz변수에 저장해서 카운터 변수 i를 ptx_sz값만큼 증가시킵니다.

예를 들어, ptrace() 함수가 4바이트 int형으로 선언되어 있다고 하면 메모리상의 값을 4바이트씩 읽어 반환할 것이므로 0x08048000 주소를 인자로 주었다고 가정하였을 때, 한번에 0x08048000 , 0x08048001 , 0x08048002 , 0x08048003 의 주소에 해당하는 값(4byte)을 반환할 것입니다. 그러므로 텍스트 세그먼트를 카운터 하는 변수 i는 반복문을 한번 거칠 때 마다 4바이트씩 증가되어야 합니다. 그래서 결국, ptx_sz변수의 값은 sizeof를 이용하여 얻은 ptrace() 함수의 선언타입 크기가 되는 것입니다.

```
dump = ptrace( PTRACE_PEEKTEXT , gcp , (DIFF_T)(VADR +i) , 0 );
```

실질적인 복사를 위한 부분으로 몇 가지 변수와 define값이 사용됩니다.

```
1) #ifdef __linux__
   #define PTX long
   #define DIFF_T void *

   #elif __FreeBSD__
   #define PTX int
   #define PTRACE_PEEKTEXT PT_READ_I
   #define DIFF_T caddr_t

2) #define VADR 0x08048000
3) PTX dump;
4) int gcp, i;
```

#ifdef, #elif 매크로를 이용하여 현재 시스템(Linux or FreeBSD)을 먼저 식별한 뒤,

시스템에 따른 정의(define)값을 선언합니다. FreeBSD에서는 PTRACE_PEEKTEXT라는 이름 대신 PT_READ_I를 사용하기 때문에 소스코드를 컴파일하는 시스템이 FreeBSD일 경우 본 프로그램의 define 값인 PTRACE_PEEKTEXT를 PT_READ_I의 값으로 정의하며, 여기서 PTRACE_PEEKTEXT와 PT_READ_I는 각 시스템에서 선언된 이름만 다를 뿐 기능은 동일합니다.

두번째 인자에 주어진 gcp의 값은 자식프로세스의 PID(Process ID)이며 ptrace() 에서 해당 PID의 가상메모리를 사용하겠다는 뜻이 됩니다.

세번째 인자인 (VADR +i)는 텍스트 세그먼트 복사를 위해 시작주소부터 변수 i의 값을 더하여 Nbyte씩 증가시키며 세그먼트의 마지막 주소까지 ptrace()의 인자로 전달합니다.

네번째 인자는 PTRACE_PEEKTEXT를 사용할 때에는 무시되므로 0(null)을 할당해 줍니다. 그리고, ptrace()를 이용하여 메모리를 읽은 반환값을 dump변수에 저장합니다.

여기서 dump의 원형이 PTX로 선언되어 있고 시스템에 따라 long형과 int형으로 구분하는 이유는, Linux상에서 ptrace() 함수의 원형은 long으로 정의되어 있으며 FreeBSD에서는 int형으로 정의되어 있기 때문에 각 시스템에 따른 ptrace() 함수의 선언타입을 따르기 위해서 입니다(DIFF_T도 동일). 그래서 Linux는 long, FreeBSD는 int로 PTX를 정의하면 dump변수가 각 시스템에 정의된 PTX의 형태로 선언되게 되고 결과적으로 ptrace()의 반환형태와 일치하도록 변수를 선언할 수 있습니다.

```
d_buf[cnt1] = dump;

if( cnt1 == PAGE/ptx_sz -1 )
{
        write( fd , &d_buf , sizeof(d_buf) );
        memset( d_buf , 0x0 , sizeof(d_buf) );
        cnt1 = -1;
}
```

메모리 영역의 복사를 위한 루틴으로 write() 함수만을 이용해서 기능 구현이 가능하지만 큰 용량의 바이너리의 경우 그 시간이 오래 걸리며 한번에 Nbyte의 기록을 위한 함수를 수 천번 이상 호출하는 것은 자원적인 면에서도 매우 비효율적입니다. 궁극적으로 함수호출의 횟수를 최소화 하여 작업수행 속도를 증가시키기 위해 일정 크기의 버퍼에 결과를 저장한 뒤 버퍼가 다 채워지게 되면 write() 함수를 호출하여 파일에 기록합니다.

그럼 이제 이 루틴에 대한 코드를 알아 보겠습니다. 사용되는 값들은 다음과 같습니다.

1) #define PAGE 0x1000

2) PTX dump, d_buf[PAGE/sizeof(PTX)];

3) char ch[PAGE+1];
4) int cnt1;
5) int fd;
6) int ptx_sz;

ptrace() 함수를 이용하여 메모리상의 데이터를 리턴받은 dump변수값을 PTX형의 d_buf 변수에 저장합니다. ptrace() 함수 선언타입의 크기만큼 값을 반환하기 때문에 그 값을 저장하는 d_buf의 변수타입도 ptrace() 함수와 같은 형태로 선언하여 줍니다.

cnt1은 반복문을 통해 d_buf 변수의 배열을 증가시켜 배열전체에 값을 저장할 수 있도록 하는 역할을 합니다. if문을 통하여 cnt1값이 (PAGE/ptx_sz -1)과 동일한가를 체크할 때 PAGE값에 ptx_sz를 나누어 준 것은 ptrace() 함수가 한번에 ptx_sz크기의 메모리를 읽어서 반환하기 때문입니다. cnt1변수의 카운터가 Nbyte 증가하면 ptrace() 함수와 동일한 선언타입인 d_buf의 배열도 증가하게 되는데, 배열 하나가 ptx_sz크기를 차지하게 되므로 4096(PAGE)번의 카운터를 수행하면, 총 (4096 x ptr_sz) 만큼의 메모리를 읽어오게 되고 결과적으로 ptx_sz만큼을 나누어주어야 합니다. 그리고 PAGE/ptx_sz에 -1을 해 준 것은 카운터변수가 0부터 시작하기 때문입니다.

여기서 버퍼링 변수 배열의 총 크기를 4KB(PAGE/sizeof(PTX)) 만큼 선언한 이유는 바이너리가 적재될 때의 가상 메모리 공간이 page(4KB) 단위로 할당되어 관리되기 때문입니다. 예를 들어, 바이너리 사상 시 가상 메모리의 공간이 3000byte가 필요할 때에 커널은 4096byte(4KB)를 할당하고 관리합니다. 이는 커널이 4KB(page)단위로 메모리를 관리하기 때문인데, 이러한 시스템의 메커니즘은 상당히 효율적인 방법으로써 자세한 내용은 커널의 메모리 관리를 살펴 보시기 바랍니다.

버퍼 변수공간이 모두 채워지면 write() 함수를 이용하여 파일에 기록한 뒤, memset() 함수를 사용해서 배열을 Null 값으로 초기화 합니다. 그리고 카운터변수를 -1로 초기화 해주어야 다음 반복구문에서 카운터 변수(cnt1)의 값이 0부터 시작 할 수 있게 됩니다.

```
if( chk ) write( fd , &d_buf , (size1 % PAGE) );
```

두 반복문 사이에 있는 위 코드에 대하여 알아보겠습니다. 다음과 같습니다.

1) #define PAGE 0x1000
2) int chk;
3) int fd, size1, size2;
4) PTX d_buf[PAGE/sizeof(PTX)];

해당 작업은 특정 조건이 만족될 때에만 수행되는데, chk 변수의 값을 체크하여 write() 함수 수행유무를 결정하게 됩니다. 여기서 chk 변수는 메모리에 사상된 크기를 얻어오는 역할을 하는 사용자 함수의 반환 값으로 할당 됩니다. 이 사용자 함수는 메모리 사상 정보가 담긴 파일에 접근 할 수 있다면 0을, 그렇지 않으면 1을 반환합니다.

사용자 함수의 반환 값(chk)이 1이면 프로그램은 메모리의 사상 정보를 얻어올 수 없다고 판단하여 대상 바이너리의 사이즈로 값을 대체합니다. 이 때, size1과 size2변수에 그 크기를 저장하게 되는데 여기에 대한 세부적인 내용은 다음 라인의 분석을 통하여 설명하도록 하겠습니다.

```
for( i = 0 ; i < size2 - size1 ; i+=ptx_sz , cnt2++ )
```

data segment의 복사를 수행하는 역할을 하는 반복문입니다.
해당 코드의 기본적인 작업은 앞서 설명한 텍스트 세그먼트 부분과 동일하며, 여기서는 반복문의(텍스트 세그먼트의 반복문도 해당) 또 다른 부가적인 역할을 설명하도록 하겠습니다. 사용되는 값들은 아래와 같습니다.

1) int i, cnt2;
2) int size1, size2;
3) int ptx_sz;

해동 코드는 앞서 설명한 텍스트 세그먼트의 복사 수행 루틴과 동일한 역할을 하지만 위에서 언급한 chk 변수의 값이 `1`일 경우(메모리 사상정보를 얻어올 수 없을 경우)에는 size1과 size2값이 각각 대상 바이너리의 크기를 가지므로 첫 번째 반복문에서 size1에 할당된 대상 바이너리의 크기로 인해 모든 복사작업이 끝나게 되기 때문에 두 번째(데이터 세그먼트) 반복문의 작업수행이 필요하지 않게 됩니다. 이것은 미리 바이너리의 사이즈를 size1, size2 변수에 각각 할당해 주었으므로 두 번째 반복문에서 size2 - size1값이 `0`이 되게 되어 해당 부분을 수행하지 않고 건너 뛸 수 있게 됩니다.

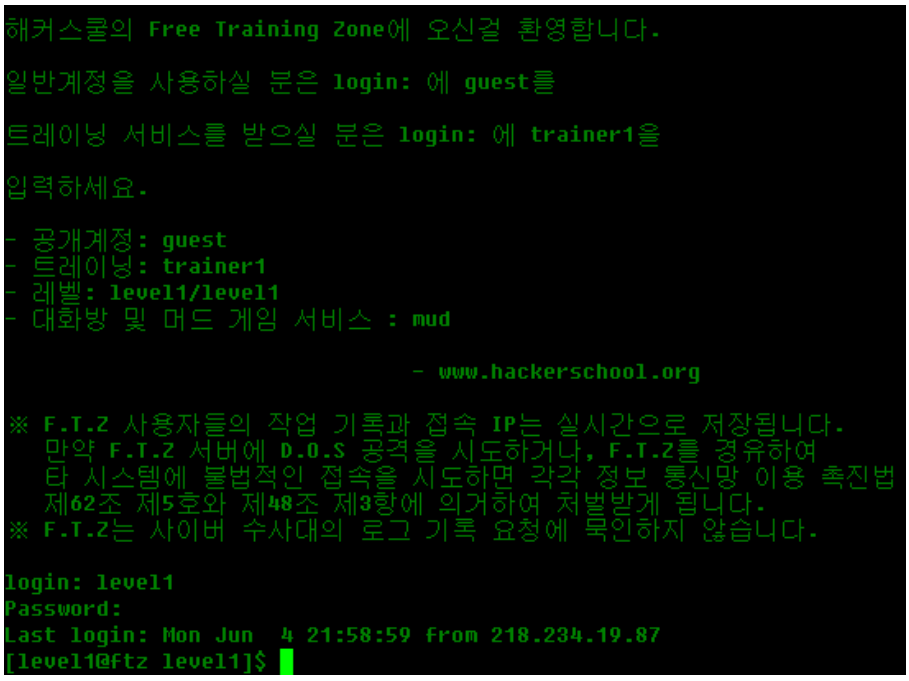
6. Let's play hktrace

문서의 기술을 토대로 만들어진 프로그램인 hktrace를 이용하여 실제환경에 적용시켜 보겠습니다. 대상 서버는 시스템 분야 워게임 서비스인 해커스쿨(http://hackerschool.org/)의 Free Training Zone(이하 FTZ)입니다. FTZ는 국내의 가장 잘 알려진 워게임 서비스 중 하나이며 리눅스의 기초를 익히는 `Trainer Service`와 실전 문제들을 풀어보는 `Hacking Zone`으로 구성되어 있습니다. level1~level20의 단계가 있으며 문제를 풀고 목표 권한을 획득한 뒤 /bin/my-pass 바이너리를 실행시켜 다음 레벨의 패스워드를 알아내는 방식입니다.

테스트 서버의 정보는 다음과 같습니다.

[level1@ftz hk]$ cat /etc/redhat-release Red Hat Linux release 9 (Shrike)
[level1@ftz hk]$ uname -a Linux ftz.hackerschool.org 2.4.32 #2 SMP 2006. 01. 17. (화) 00:18:19 KST i686 i686 i386 GNU/Linux

FTZ서비스를 이용하기 위해 ftz.hackerschool.org에 접속합니다.



<그림5> *FTZ(ftz.hackerschool.org)에 접속한 모습*

아래는 문제를 풀었을 경우 다음 레벨의 패스워드를 출력해 주는 /bin/my-pass입니다.

```
[level1@ftz level1]$ ls -l /bin/my-pass
-rwxr-x--x    1 root     root        13919 12월  9  2003 /bin/my-pass
```

<그림6> /bin/my-pass의 바이너리 정보

<그림6>을 통하여 /bin/my-pass는 root소유의 프로그램으로 크기는 13919byte이며 root를 제외한 사용자들에게는 실행권한만이 주어진 것을 확인할 수 있습니다. 그 외에 바이너리의 생성날짜, 하드링크 수를 알 수 있으며, 본 문서에서 다루는 중요한 정보는 대상 바이너리의 권한, 소유자, 크기 입니다.

/bin/my-pass는 읽기 권한이 없기 때문에 cp, strings명령이 허용되지 않습니다.

```
[level1@ftz hk]$ cp /bin/my-pass hk-pass
cp: cannot open `/bin/my-pass' for reading: 허가 거부됨
[level1@ftz hk]$ strings /bin/my-pass
strings: /bin/my-pass: 허가 거부됨
```

<그림7> cp, strings명령의 실패

문서의 이론을 기반으로 만들어진 hktrace를 이용하여 방법론에 대한 증명을 해보겠습니다. 대상 바이너리는 <그림6>에서 살펴본 /bin/my-pass입니다.

```
[level1@ftz hk]$ ./hktrace /bin/my-pass

        ==================== hktrace ====================
        hktrace is no-read permission binary copying tool
        target program is needed execute-permission only
        available system is Linux and FreeBSD
        create filename is *.hk

        made by ChanAm, Park
        hkpco@korea.com
        =================================================

        [+] target name -  /bin/my-pass
        [+] information -  -rwxr-x--x    1 root     root        13919 12월  9  2003 /bin/my-pass
        [+] system info -  Linux 2.4.32

        [+] 0x8048000-0x8049000 copy ok.
        [+] 0x8049000-0x804a000 copy ok.

        !! [my-pass.hk] created.
        !! [my-pass.hk] -rwx------    1 level1   level1       8192  8월 15 12:26 my-pass.hk
```

<그림8> hktrace의 실제 사용

<그림8>에서 성공적으로 hktrace가 수행된 것을 볼 수 있습니다. 읽기권한이 주어지지 않은 root소유의 바이너리인 /bin/my-pass를 현재 디렉토리에 my-pass.hk라는 이름으로 복사 하였습니다.

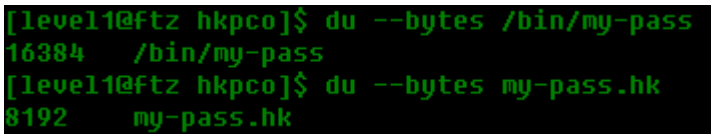
앞서 언급한 것처럼 /bin/my-pass는 해당 레벨의 문제를 풀고 새로운 권한을 획득하여 다음 레벨의 패스워드를 알아내기 위해 사용됩니다. 그래서 각 레벨별 패스워드를 출력해주기 위하여 /bin/my-pass에 모든 레벨의 패스워드가 포함되어 있다는 것을 알 수 있습니다. 그렇다면 strings명령을 이용하여 해당 바이너리의 문자열들을 볼 수 있을 것이고, 출력된 문자열들 중에 각 레벨별 패스워드가 존재할 것입니다. 하지만 /bin/my-pass에는 읽기권한이 주어지지 않아(<그림6>참고) strings명령으로 바이너리를 읽어 문자열을 추출 할 수 없기 때문에 hktrace로 복사된 my-pass.hk를 이용해야 합니다.

```
[level1@ftz hk]$ strings my-pass.hk
/lib/ld-linux.so.2
libc.so.6
printf
geteuid
getuid
system
_IO_stdin_used
__libc_start_main
__gmon_start__
GLIBC_2.0
PTRh
clear
Level1 Password is "level1".
Level2 Password is "hacker or cracker".
Level3 Password is "can you fly?".
Level4 Password is "suck my brain".
Level5 Password is "what is your name?".
Level6 Password is "what the hell".
Level7 Password is "come together".
Level8 Password is "break the world".
Level9 Password is "apple".
Level10 Password is "interesting to hack!".
Level12 Password is "it is like this".
Level13 Password is "have no clue".
Level14 Password is "what that nigga want?".
Level15 Password is "guess what".
Level16 Password is "about to cause mass".
Level17 Password is "king poetic".
Level18 Password is "why did you do it".
Level19 Password is "swimming in pink".
Level20 Password is "we are just regular guys".
clear Password is "i will come in a minute".
```

<그림9> my-pass.hk의 strings결과

<그림7>에서 cp, strings명령을 시도했을 때에는 /bin/my-pass에 읽기권한이 없기 때문에 실패 하였지만, hktrace로 복사된 바이너리인 my-pass.hk는 strings명령을 이용하여 성공적으로 문자열을 추출하였으며, 그 결과 <그림9>와 같이 FTZ의 모든 레벨 패스워드를 열람할 수 있었습니다.

그런데 여기서 두 바이너리(/bin/my-pass, my-pss.hk)의 사이즈가 서로 다른 것을 확인할 수 있습니다. du명령을 이용하여 크기를 비교해 보겠습니다.



<그림10> du명령을 이용한 바이너리 사이즈 비교

<그림10>에서 /bin/my-pass, my-pass.hk의 크기는 각각 16384byte, 8192byte인 것을 볼 수 있습니다. 바이너리의 실행에는 문제가 없지만, 동일한 복사본인 my-pass.hk의 크기가 원본(/bin/my-pass)과 다른 이유는 실행 전후의 바이너리 구조가 다르기 때문인데, hktrace가 실행에 맞게 메모리에 사상 된 후의 데이터를 추출하기 때문에 결과적으로 원본과 크기가 다르게 되는 것입니다.

7. 마치며

지금까지 권한 없는 바이너리의 복사에 관한 방법론과 실제 적용을 알아보았습니다.
이것은 전혀 새로운 기법이 아니며 유사한 개념으로 gdb, strace 등의 각종 디버깅 툴에서 묵시적으로 사용되고 있던 것입니다. 본 문서에서 소개한 권한 없는 바이너리의 접근 기술이 허용되는 것은 어쩌면 유닉스 계열 시스템의 권한 체계에 대한 디자인상의 오류일 수도 있습니다.

8. 참고자료

[1] 리눅스 문제 분석과 해결 - 에이콘
[2] Linux Manual Page
[3] Freebsd Manual Page
[4] http://lxr.linux.no/source/kernel/ptrace.c