

기 술 문 서

LD_PRELOAD와 공유 라이브러리를 사용한 libc 함수 후킹

정지훈
binoopang@is119.jnu.ac.kr



Abstract

libc에서 제공하는 API를 후킹 해 본다. 물론 이 방법을 사용하면 다른 라이브러리에서 제공하는 API들도 후킹 할 수 있다.

여기서 제시하는 방법은 리눅스 후킹에서 가장 기본적인 방법이 될 것이기 때문에 후킹의 워밍업이라고 생각하고 읽어보자 :D

Content

1. 목적	1
1.1. 아이디어	1
2. Hooker 제작	2
2.1. 실제 libc 함수의 포인터 획득	2
2.2. Hooker 공유 라이브러리 생성	6
3. Hooker 테스트	10
3.1. 테스트 환경	10
3.2. LD_PRELOAD를 사용한 Hooker 로드	10
4. 마치며	12
참고문헌	13

1. 목적

이 문서에서는 공유라이브러리와 LD_PRELOAD를 사용하여 API후킹을 시도할 것이다. 기본적인 아이디어와 기술이 간단해서 쉽게 사용할 수 있는 기술일 것이다.

1.1. 아이디어

리눅스에서 프로세스가 로드될 때 여러 가지 라이브러리들을 로드한다. 그중 'libc'는 C언어에서 제공되는 API들이 들어있는 라이브러리이다. 여기서 LD_PRELOAD를 사용해서 강제로 라이브러리를 프로세스에 먼저 로드시키고 만약 그 라이브러리에 'libc'에 들어있는 함수와 동일한 이름의 함수가 존재할 경우 프로세스는 'libc'가 아닌 LD_PRELOAD를 통해 로드된 라이브러리의 함수를 호출하게 된다. 이것은 라이브러리의 로드 순서의 문제이다.

따라서 우리는 LD_PRELOAD를 사용해서 'libc'함수와 동일한 형식의 후킹함수를 로드시켜서 그 함수를 먼저 실행한 다음 원래 'libc'의 함수로 흐름을 넘겨줄 것이다.

2. Hooker 제작

2.1. 실제 'libc' 함수의 포인터 획득

후킹의 기본은 함수 포인터이다. 먼저 함수 포인터를 사용해서 실제 'libc'의 함수 주소를 가지고 있다가 우리의 함수가 일을 마치거나 필요시에 실제 'libc' 함수를 호출해야 한다.

그렇다면 어떻게 실제 'libc' 함수 주소를 얻어낼 수 있을까? gdb를 사용하여 getuid 주소를 찾아 보자.

```
Breakpoint 1, 0x080483f2 in main ()
Current language: auto; currently asm
(gdb) p getuid
$1 = {<text variable, no debug info>} 0xb7e43a50 <getuid>
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /mnt/ntfs/my_documents/document/c/hook/test

Breakpoint 1, 0x080483f2 in main ()
(gdb) p getuid
$2 = {<text variable, no debug info>} 0xb7ea5a50 <getuid>
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /mnt/ntfs/my_documents/document/c/hook/test

Breakpoint 1, 0x080483f2 in main ()
(gdb) p getuid
$3 = {<text variable, no debug info>} 0xb7e26a50 <getuid>
(gdb) █
```

[그림 1] gdb를 사용한 getuid 함수 주소 출력

gdb를 사용해서 'getuid' 함수의 주소를 출력해 보았다. [그림 1]에서 총 3번에 걸쳐서 실행하여 각각 주소를 출력하였지만 모두 결과가 다른 것을 볼 수 있다. 이것은 실행할 때마다 라이브러리의 로드되는 주소가 약간씩 달라지기 때문에 함수의 주소도 조금씩 다른 것이다.

2.1.1. 메모리 스캐너

그렇다면 매번 실행할 때마다 'getuid' 함수를 찾아야 할 것 같다. 여기서는 메모리를 스캔하는 방법을 사용해 보았다.

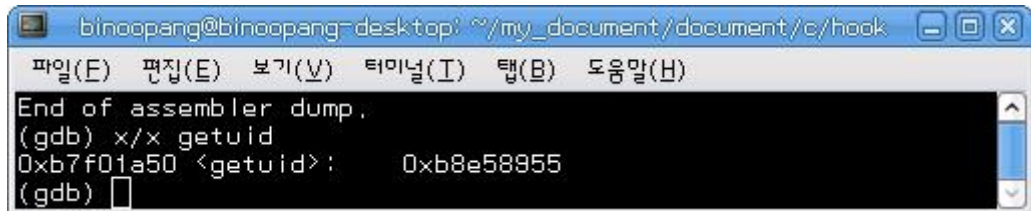
```

1 main()
2 {
3   unsigned int offset;
4   char pattern[] = "Wx55Wx89Wxe5Wxb8";
5   for(offset = 0xb7e00000 ; offset < 0xb7efffff ; offset++)
6   {
7     if(!strncmp((char *)offset, pattern, strlen(pattern)))
8     {
9       printf("found!! 0x%xWn", offset);
10    }
11  }
12 }

```

[코드 1] 메모리 스캔 코드

[그림 1]에서 getuid의 주소가 0xb7e00000 부근에서 나타나는 것을 착안하여 그 부근을 검색하는 것이다. 이때 pattern에 등록되어 있는 'Wx55Wx89Wxe5Wxb8'는 gdb를 사용하여 찾은 getuid 함수의 기계어이다.



[그림 2] getuid() 주소와 기계어코드

[그림 2]와 같은 방법으로 찾을 수 있으며 기계어코드의 의미는 다음과 같다.

```

0xb7f01a50 <getuid+0>: push   %ebp
0xb7f01a51 <getuid+1>: mov    %esp,%ebp
0xb7f01a53 <getuid+3>: mov    $0xc7,%eax

```

[코드 2] 기계어 코드 해석

[코드 2]에서 보여 지는 어셈블리 코드가 [그림 2]에서 찾은 'getuid'의 기계어이다. 조금 더 정확도를 높이고 싶다면 긴 패턴을 사용하면 될 것이다.

```

binoopang@binoopang-desktop: ~/my_document/document/c/hook
파일(E) 편집(E) 보기(V) 터미널(T) 탭(B) 도움말(H)
[ 빈 누 :~/my_document/document/c/hook]$ ./test
found!! 0xb7e28a50
found!! 0xb7e28a70
found!! 0xb7e28a90
found!! 0xb7e28ab0
Segmentation fault
[ 빈 누 :~/my_document/document/c/hook]$

```

[그림 3] 스캐너 실행

[그림 3]에서 처럼 스캐너를 실행하면 총 4개의 함수 주소가 발견된다. 4개의 주소는 순서대로 `getuid`, `geteuid`, `getgid`, `getegid` 이다. 모두 비슷한 패턴을 쓰다보니 한 꺼번에 검색된 것이다.

[그림 3]에서 한 가지 문제점이 있는데 하단 분에 'Segmentation fault'가 발생하였다. 보통 'SIGSEGV' 시그널은 잘못된 메모리에 접근하였을 때 발생한다. 여기서는 아마도 'libc'를 벗어난 다른 메모리 지역에 들어갔을 경우 이렇게 'SIGSEGV' 시그널이 발생하는 것 같다.

2.1.2. libc가 로드된 주소 검색

[그림 3]의 문제점을 해결하기 위해서는 실제 'libc'가 로드된 주소를 정확히 알 필요가 있다. 많은 검색을 통해서 'dl_iterate_phdr()'를 알게 되었고 아래와 같은 코드를 작성할 수 있었다.

```

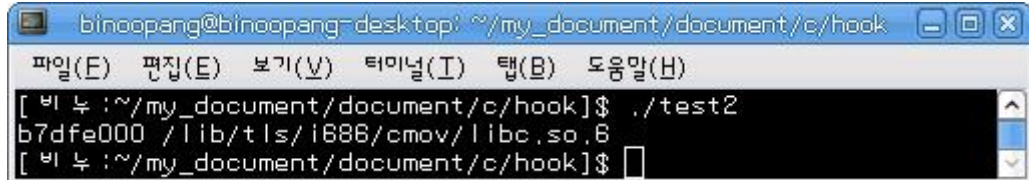
1 #define _GNU_SOURCE
2 #include <stdio.h>
3 #include <link.h>
4
5 static int print_callback(struct dl_phdr_info *info,
6     size_t size,
7     void *data)
8 {
9     if((strstr(info->dlpi_name, "libc")))
10         printf("%08x %sWn", info->dlpi_addr, info->dlpi_name);
11     return 0;
12 }
13
14 int main()
15 {
16     dl_iterate_phdr(print_callback, NULL);
17     return 0;
18 }

```

[코드 3] 'libc' 라이브러리 시작 주소 검색 루틴

[코드 3]에서 'dl_iterate_phdr'은 콜백함수를 사용하며 콜백함수는 로드되는 라이브러리 횃수만큼 호출되어서 라이브러리 정보를 구조체에 담는다 여기서 우리가 필요한 항목은 라이브러리 이름과 시작주소이다.

[코드 3]의 9번 줄에서 'strstr()'을 사용하여 라이브러리 이름에 'libc'가 들어가 있으면 해당 라이브러리의 시작 주소를 가져오는 형식이다.



[그림 4] 로드된 libc 라이브러리 주소 획득

참고로 여기서 사용된 함수는 'GNU 확장 함수'¹⁾이기 때문에 '#define _GNU_SOURCE'를 정의해 주어야 한다.

여기까지 해서 메모리 스캔하는데 필요한 작업이 완료 되었다. 실제 완성된 스캔코드는 아래와 같다.

```

1 void search_getuid_offset(void)
2 {
3 printf("[Hooker] Searching getuid offsetWn");
4 char pattern[] = "Wx55Wx89Wxe5Wxb8";
5 for(offset = libc_start ; offset < 0xb7ffffff ; offset++)
6 {
7 if(!strncmp((char *)offset, pattern, strlen(pattern)))
8 {
9 printf("[Hooker] Found offset : 0x%xWn", offset);
10 break;
11 }
12 }
13 if(offset == (unsigned long)NULL)
14 {
15 printf("[Hooker] Can not found offsetWn");
16 exit(0);
17 }
18}
    
```

[코드 4] 메모리 스캔 코드

1) GNU 확장 함수는 표준함수가 아니기 때문에 꼭 _GNU_SOURCE를 정의해 주어야 한다. 그렇지 않으면 컴파일시에 에러가 발생한다.

[코드 4]에서 5번 줄의 'libc_start' 변수는 'dl_iterate_phdr'의 콜백함수에 의해 얻어낸 'libc'가 로드된 주소이다. 우리는 검색되는 4개의 함수 중 첫 번째 함수만 필요하므로 첫 번째 함수가 호출되면 'break'를 사용하여 for문을 빠져 나온다.

2.2. Hooker 공유 라이브러리 생성

스캔 코드를 기반으로 이제 공유 라이브러리를 생성해야 한다. 여기서는 'getuid' 함수를 후킹할 것이기 때문에 공유 라이브러리에서 메인함수는 'getuid()' 이름으로 해야한다. 그래야 실제 어플리케이션에서 'getuid()'를 호출하면 우리의 라이브러리에 존재하는 'getuid()'를 호출한다.

```
1 // getuid 후킹함수
2 int getuid()
3 {
4     int ret;
5     if(signal(SIGSEGV, sig_usr) == SIG_ERR)
6         fprintf(stderr, "Can not catch signalWn");
7
8     dl_iterate_phdr(scan_callback, NULL);
9     printf("[Hooker] libc start address : 0x%xWn", libc_start);
10    search_getuid_offset();
11    orig_getuid = offset;
12    printf("[hooker] Call Orig getuid!!Wn");
13    ret = orig_getuid();
14    printf("[hooker] Return value of getuid : %dWn", ret);
15    return ret+1;
16 }
```

[코드 5] getuid() 후킹 함수

[코드 5]는 'getuid()' 후킹함수 이다. 이 함수가 흔히 우리가 프로그래밍할 때 사용하는 main()의 역할을 하게 된다. 8번 줄에서 가장먼저 'libc'의 로드주소를 찾은 다음 10번 줄에서 실제 'libc'의 getuid함수 주소를 찾는다. 찾아낸 주소는 11번 줄에서 함수 포인터에 넣어지고 13번 줄에서 호출되어 변수에 저장된다.

가장 마지막 줄에 'return ret+1'이 있는데 이렇게 되면 실제 uid에 1을 더한 uid가 리턴될 것이다.

```

1 /* -- API Hooking by bin00pang -- */
2
3 #define _GNU_SOURCE
4 #include <stdio.h>
5 #include <link.h>
6 #include <stdlib.h>
7 #include <string.h>
8 #include <signal.h>
9
10 // 실제 getuid의 주소를 담을 변수
11 unsigned long offset = (unsigned long)NULL;
12
13 unsigned long libc_start = (unsigned long)NULL;
14
15 // 실제 getuid()의 진입점을 가리킬 함수 포인터
16 int (*orig_getuid)(void) = NULL;
17
18 void search_getuid_offset(void);
19
20 // SIGSEGV 시그널 핸들러
21 static void sig_usr()
22 {
23     printf("[Hooker] Received SIGSEGV signal .. quit .. \n");
24     exit(0);
25 }
26
27 // libc가 로드된 주소를 찾기위한 루틴
28 static int scan_callback(struct dl_phdr_info *info,
29     size_t size,
30     void *data)
31 {
32     if(strstr(info->dlpi_name, "libc"))
33         libc_start = (unsigned int)info->dlpi_addr;
34     return 0;
35 }
36
37
38 // getuid 후킹함수
39 int getuid()

```

```

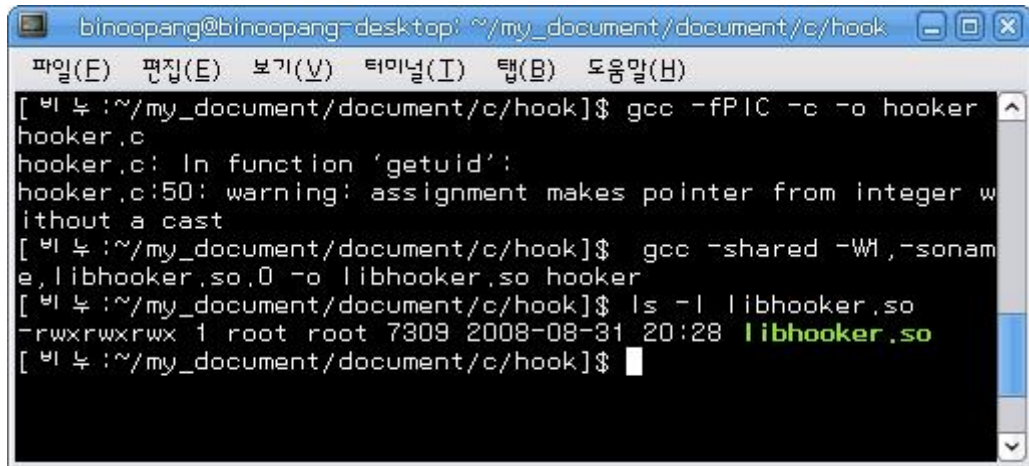
40 {
41     int ret;
42     if(signal(SIGSEGV, sig_usr) == SIG_ERR)
43         fprintf(stderr, "Can not catch signalWn");
44
45     dl_iterate_phdr(scan_callback, NULL);
46     printf("[Hooker] libc start address : 0x%xWn", libc_start);
47     search_getuid_offset();
48     orig_getuid = offset;
49     printf("[hooker] Call Orig getuid!!Wn");
50     ret = orig_getuid();
51     printf("[hooker] Return value of getuid : %dWn", ret);
52     return ret+1;
53 }
54
55 // 실제 getuid의 진입점 찾는 루틴
56 void search_getuid_offset(void)
57 {
58     printf("[Hooker] Searching getuid offsetWn");
59     char pattern[] = "Wx55Wx89Wxe5Wxb8";
60     for(offset = libc_start ; offset < 0xb7ffffff ; offset++)
61     {
62         if(!strncmp((char *)offset, pattern, strlen(pattern)))
63         {
64             printf("[Hooker] Found offset : 0x%xWn", offset);
65             break;
66         }
67     }
68
69     if(offset == (unsigned long)NULL)
70     {
71         printf("[Hooker] Can not found offsetWn");
72         exit(0);
73     }
74 }

```

[코드 6] Hooker 전체 코드

[코드 6]은 전체적인 Hooker의 코드이다. 생각보다 길지 않은 것을 볼 수 있다.(물론 더 짧게 만들 수도 있다.:D)

이제 실제 [코드 6]을 공유 라이브러리로 만들어보자. 공유 라이브러리는 아래와 같은 과정으로 만들어진다.



```
binoopang@binoopang-desktop: ~/my_document/document/c/hooker
파일(E) 편집(E) 보기(V) 터미널(T) 탭(B) 도움말(H)
[비누:~/my_document/document/c/hooker]$ gcc -fPIC -c -o hooker
hooker.c
hooker.c: In function 'getuid':
hooker.c:50: warning: assignment makes pointer from integer without a cast
[비누:~/my_document/document/c/hooker]$ gcc -shared -Wl,-soname,libhooker.so.0 -o libhooker.so hooker
[비누:~/my_document/document/c/hooker]$ ls -l libhooker.so
-rwxrwxrwx 1 root root 7309 2008-08-31 20:28 libhooker.so
[비누:~/my_document/document/c/hooker]$
```

[그림 5] 공유 라이브러리 생성

생성되는 대략적인 순서는 다음과 같다.

먼저 오브젝트 파일로 컴파일 한다.

```
$ gcc -fPIC -c -o hooker hooker.c
```

공유 라이브러리로 컴파일 한다.

```
$ gcc -shared -Wl,-soname,libhooker.so.0 -o libhooker.so hooker
```

이와같은 과정을 거치면 [그림 5]와 같이 'libhooker.so'라는 공유라이브러리가 생성된다. 이제 모든 후킹의 준비가 완료되었다.

3. Hooker 테스트

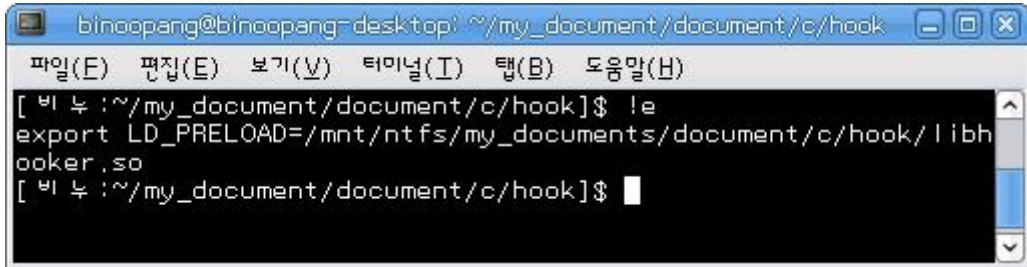
3.1. 테스트 환경

Hooker의 테스트 환경은 다음과 같다.

- 커널 버전 : 2.6.24-19-generic
- gcc 버전 : 4.2.3

3.2. LD_PRELOAD를 사용한 Hooker 로드

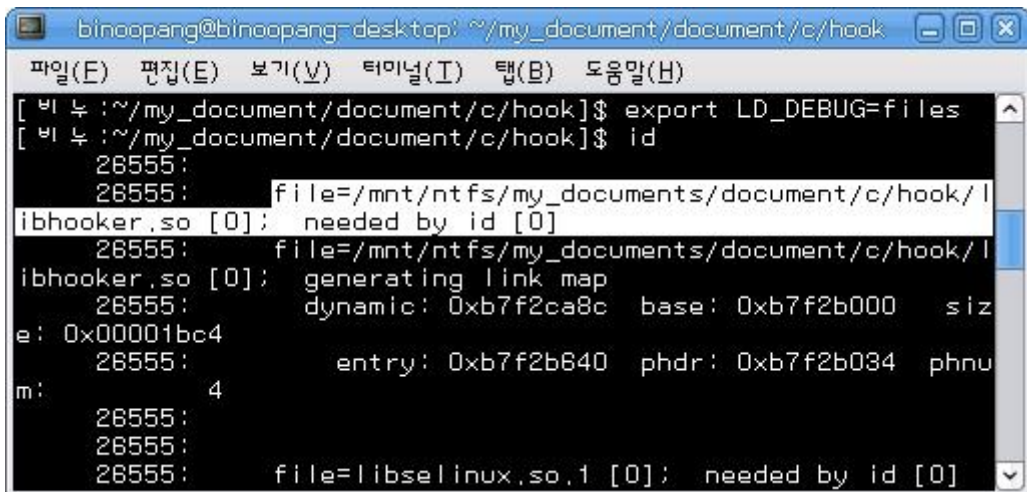
우리가 만든 Hooker를 사용하기 위해서는 일단 라이브러리가 프로세스에 로드되어야 한다. 이 작업은 환경변수인 LD_PRELOAD가 쉽게 해준다.



```
binoopang@binoopang-desktop: ~/my_document/document/c/hook
파일(E) 편집(E) 보기(V) 터미널(I) 탭(B) 도움말(H)
[ 빈 누 :~/my_document/document/c/hook]$ !e
export LD_PRELOAD=/mnt/ntfs/my_documents/document/c/hook/libhooker.so
[ 빈 누 :~/my_document/document/c/hook]$
```

[그림 6] LD_PRELOAD 익스포트

[그림 6]과 같이 LD_PRELOAD의 값을 libhooker.so의 절대경로로 설정한 다음 export 시키면 이제 준비가 끝난 것이다.

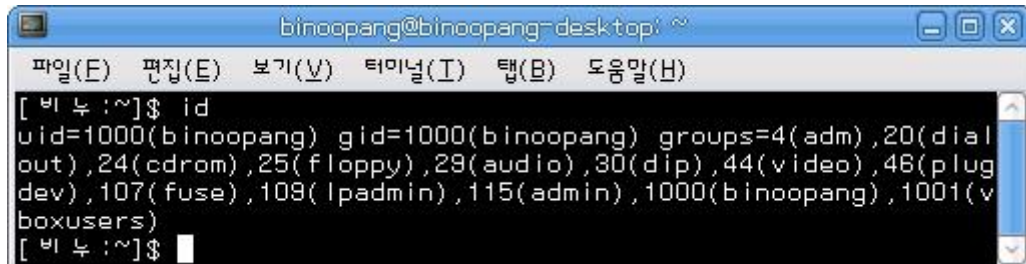


```
binoopang@binoopang-desktop: ~/my_document/document/c/hook
파일(E) 편집(E) 보기(V) 터미널(I) 탭(B) 도움말(H)
[ 빈 누 :~/my_document/document/c/hook]$ export LD_DEBUG=files
[ 빈 누 :~/my_document/document/c/hook]$ id
26555:
26555: file=/mnt/ntfs/my_documents/document/c/hook/libhooker.so [0]; needed by id [0]
26555: file=/mnt/ntfs/my_documents/document/c/hook/libhooker.so [0]; generating link map
26555: dynamic: 0xb7f2ca8c base: 0xb7f2b000 size: 0x00001bc4
26555: entry: 0xb7f2b840 phdr: 0xb7f2b034 phnum: 4
26555:
26555:
26555: file=libselinux.so.1 [0]; needed by id [0]
```

[그림 7] 디버그모드로 실행

[그림 7]은 'id'명령어를 디버그모드로 실행한 것이다. 보면 가장 먼저 로드하는 라이브러리

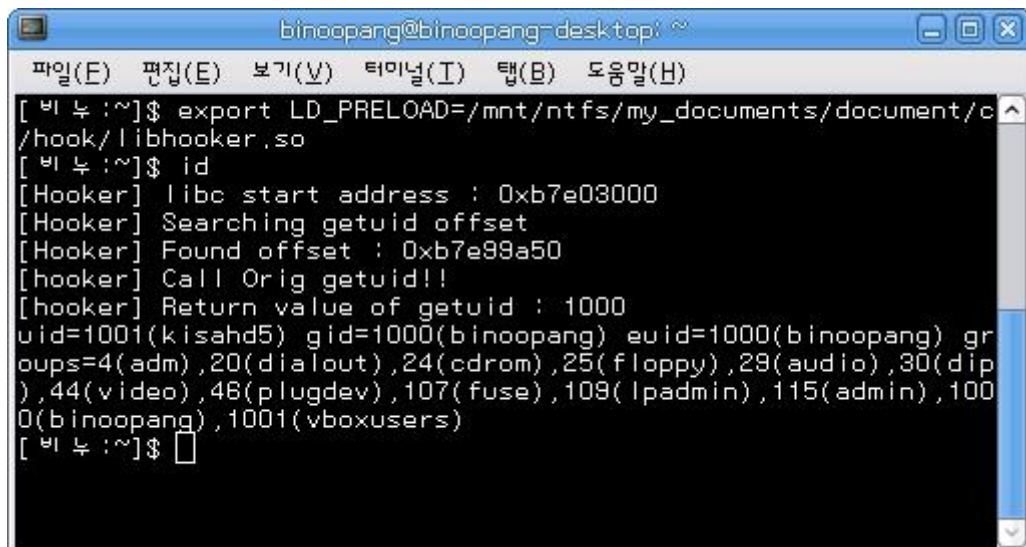
가 바로 방금 우리가 만든 'libhooker.so'라는 것을 확인할 수 있다. 먼저 정상적인 'id'의 결과를 확인해 보자.



```
binoopang@binoopang~$ id
uid=1000(binoopang) gid=1000(binoopang) groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),107(fuse),109(lpadmin),115(admin),1000(binoopang),1001(vboxusers)
binoopang@binoopang~$
```

[그림 8] 정상적인 id의 결과

'libhooker.so'를 로드하기 전에는 정상적인 id결과가 출력된다. 실제 'binoopang'의 uid는 1000이다. 이제 후킹을 한 다음의 결과를 보자.



```
binoopang@binoopang~$ export LD_PRELOAD=/mnt/ntfs/my_documents/document/c/hook/libhooker.so
binoopang@binoopang~$ id
[Hooker] libc start address : 0xb7e03000
[Hooker] Searching getuid offset
[Hooker] Found offset : 0xb7e99a50
[hooker] Call Orig getuid!!
[hooker] Return value of getuid : 1000
uid=1001(kisahd5) gid=1000(binoopang) euid=1000(binoopang) groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),107(fuse),109(lpadmin),115(admin),1000(binoopang),1001(vboxusers)
binoopang@binoopang~$
```

[그림 9] 후킹 후의 id 결과

[그림 9]에서 정상적으로 getuid가 후킹되었고 원래 1000이었던 uid가 1001로 바뀌어 출력된 것을 확인할 수 있다.

3. 마치며

이 문서에서 제시한 후킹 방법은 매우 간단하다.(사실 걸어가다 생각나서 해본 것이다.) 이 방법에는 후킹으로써는 제한되는 부분이 많이 있다. 특히 라이브러리를 사용하는 방법이기 때문에 제거 하는 것이 너무 쉽다.

다음에는 다른 방법으로 후킹 하는 방법에 대해서 써보야겠다.

참고문헌

- [1] Linux Manpage, "ld_iterate_phdr" http://linux.die.net/man/3/ld_iterate_phdr