Invoke-PSImage利用分析

嘶吼roartalk / 2018-01-02 16:33:01 / 浏览数 2885 安全技术 技术讨论 顶(0) 踩(0)

0x00 前言

最近在github看见一个有趣的项目：Invoke-PSImage，在png文件的像素内插入powershell代码作为payload(不影响原图片的正常浏览)，在命令行下仅通过一行powersh

这是一种隐写(Steganography)技术的应用，我在之前的文章对png的隐写技术做了一些介绍，可供参考：

《隐写技巧——PNG文件中的LSB隐写》

《隐写技巧——利用PNG文件格式隐藏Payload》

本文将结合自己的一些心得对Invoke-PSImage进行分析，介绍原理，解决测试中遇到的问题，学习脚本中的编程技巧，提出自己的优化思路

Invoke-PSImage地址：

https://github.com/peewpw/Invoke-PSImage

0x01 简介

本文将要介绍以下内容：

- 脚本分析
- 隐写原理
- 实际测试
- 编程技巧
- 优化思路

0x02 脚本分析

1、参考说明文件

https://github.com/peewpw/Invoke-PSImage/blob/master/README.md

(1) 选取每个像素的两个颜色中的4位用于保存payload

(2) 图像质量将受到影响

(3) 输出格式为png

2、参考源代码对上述说明进行分析

(1) 像素使用的为RGB模式，分别选取颜色分量中的G和B的低4位(共8位)保存payload

(2) 由于同时替换了G和B的低4位，故图片质量会受影响

补充：

LSB隐写是替换RGB三个分量的最低1位，人眼不会注意到前后变化，每个像素可以存储3位的信息

猜测Invoke-PSImage选择每个像素存储8位是为了方便实现(8位=1字节)，所以选择牺牲了图片质量

(3) 输出格式为png，需要无损

png图片为无损压缩(bmp图片也是无损压缩)，jpg图片为有损压缩。所以在实际测试过程，输入jpg图片，输出png图片，会发现png图片远远大于jpg图片的大小

(4) 需要注意payload长度，每个像素保存一个字节，像素个数需要大于payload的长度

0x03 隐写原理

参照源代码进行举例说明(跳过读取原图片的部分)

1、修改像素的RGB值，替换为payload

代码起始位置：

https://github.com/peewpw/Invoke-PSImage/blob/master/Invoke-PSImage.ps1#L110

对for循环做一个简单的修改，假定需要读取0x73，将其写入第一个像素RGB(0x67,0x66,0x65)

（1）读取payload

代码：

```
$paybyte1 = [math]::Floor($payload[$counter]/16)
```

说明：

`$payload[$counter]/16`表示`$payload[$counter]/0x10`

即取0x73/0x10，取商，等于0x07

所以，$paybyte1 = 0x07

代码：

```
$paybyte2 = ($payload[$counter] -band 0x0f)
```

说明：

即0x73 & 0x0f，结果为0x03

所以，$paybyte2 = 0x03

代码：

```
$paybyte3 = ($randb[($counter+2)%109] -band 0x0f)
```

说明：

作随机数填充，$paybyte3可忽略

注：

原代码会将payload的长度和图片的像素长度进行比较，图片多出来的像素会以同样格式被填充成随机数

（2）向原像素赋值，添加payload

原像素为`RGB(0x62,0x61,0x60)`

代码：

```
$rgbValues[($counter*3)] = ($rgbValues[($counter*3)] -band 0xf0) -bor $paybyte1
```

说明：

即0x60 & 0xf0 | 0x07

所以，$rgbValues[0] = 0x67

代码：

```
$rgbValues[($counter*3+1)] = ($rgbValues[($counter*3+1)] -band 0xf0) -bor $paybyte2
```

说明：

即0x61 & 0xf0 | 0x03

所以，$rgbValues[1] = 0x63

代码：

```
$rgbValues[($counter*3+2)] = ($rgbValues[($counter*3+2)] -band 0xf0) -bor $paybyte3
```

说明：

随机数填充，可忽略

综上，新像素的修改过程为：

R： 高位不变，低4位填入随机数
G： 高位不变，低4位填入payload的低4位
B： 高位不变，低4位填入payload的高4位

2、读取RGB，还原出payload

对输出做一个简单的修改，读取第一个像素中的payload并还原

取第0个像素的代码如下：

```
sal a New-Object;
Add-Type -AssemblyName "System.Drawing";
$g= a System.Drawing.Bitmap("C:\1\evil-kiwi.png");
$p=$g.GetPixel(0,0);
$p;
```

还原payload，输出payload的第一个字符，代码如下：

```
$o = [math]::Floor(($p.B -band 15)*16) -bor ($p.G -band 15);
[math]::Floor(($p.B -band 15)*16) -bor ($p.G -band 15));
```

0x04 实际测试

使用参数：

```
Invoke-PSImage -Script .\test.ps1 -Image .\kiwi.jpg -Out .\evil-kiwi.png
```

test.ps1: 包含payload，例如"start calc.exe"

kiwi.jpg： 输入图片，像素数量需要大于payload长度

evil-kiwi.png: 输出图片路径

脚本执行后会输出读取 图片解密payload并执行的代码

实际演示略

0x05 优化思路

结合前面的分析，选择替换RGB中两个分量的低4位保存payload，会在一定程序上影响图片质量，可参照LSB隐写的原理只替换三个分量的最低位，达到人眼无法区别的效果

当然，该方法仅是隐写技术的一个应用，无法绕过Win10 的AMSI拦截

在Win10 系统上测试还需要考虑对AMSI的绕过

0x06 小结

本文对Invoke-PSImage的代码进行分析，介绍加解密原理，分析优缺点，提出优化思路，帮助大家更好的进行学习研究

上一篇：AlphaJump - 如何用机器... 下一篇：Misc 总结 ----隐写术之电...
1. 0 条回复
   - 动动手指，沙发就是你的了！

目录

RSS 关于社区 友情链接 社区小黑板