Malware Analysis with CTO and CTO Function Lister

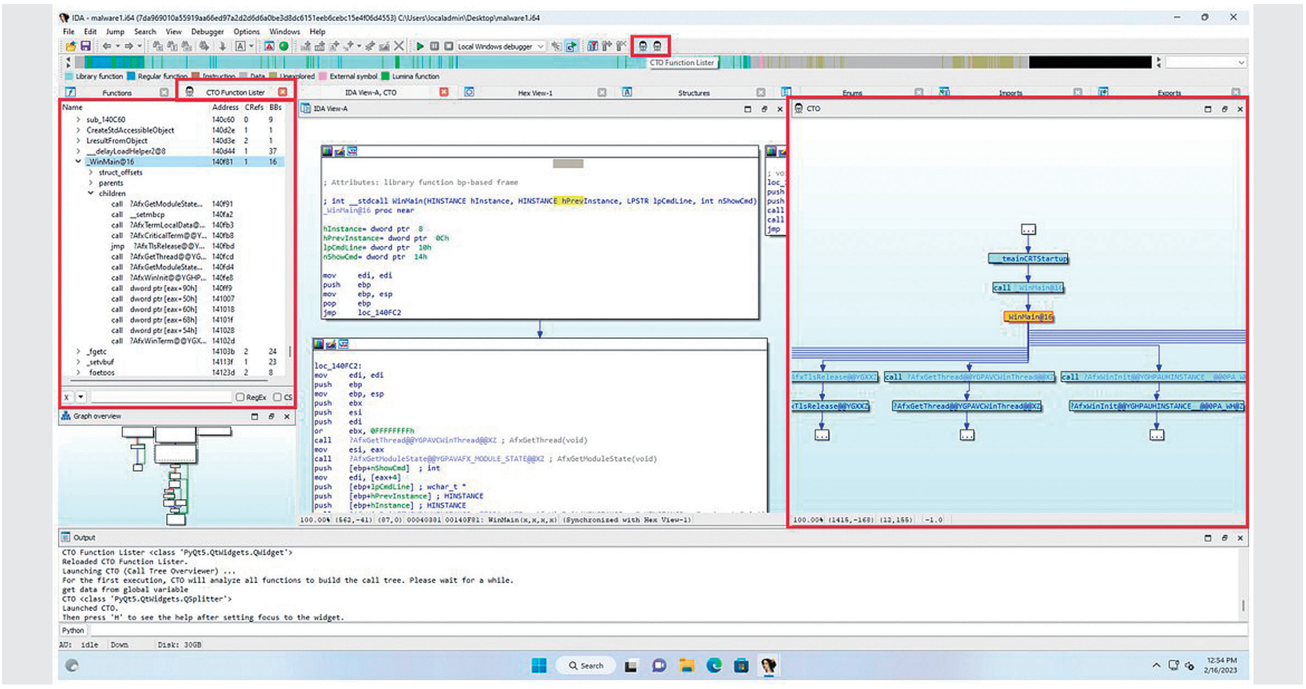
At the Virus Bulletin conference in 2021 (VB2021 localhost), I presented tools called CTO and CTO Function Lister[*1]. I have continued to improve the tools since then by adding new functionality. In this article, I explain what sort of malware analysis tasks these tools are applicable to, along with an in-the-wild malware sample.

The malware sample I use here is selfmake3, which downloads and executes a RAT called SpiderPig, and it has been used in targeted attacks. The SHA256 hash value appears below.

```
7DA969010A55919AA66ED97A2D2D6D6A0BE3D8DC6151EEB6CEBC15E4F06D4553
```

2.1 Startup and Initial Windows

Both CTO and CTO Function Lister are IDA Pro[*2] plugins. They can be launched from Plugins in the Edit menu, toolbar buttons, or shortcut keys. In Figure 1, you will see icons that look like a middle-aged man on the far right of the IDA window toolbar. These are the CTO and CTO Function Lister icons. When clicked, CTO Function Lister appears on the left side of the window, and CTO on the right. The CTO tool is mainly used for visualizing function call parent-child relationships. The main purpose of CTO Function Lister is to extract and retain a list of functions and notable characteristics of each function, and to search for the information via filters. In the figure, you can see that each tool is synchronized with the address of the "_WinMain" function (more precisely, the MFC AfxWinMain function) displayed in IDA's disassembly view (IDA View-A) and is displaying information for that address.



```
Name                          Address  CRefs  BBs
> sub_140C60                  140c60   0      9
> CreateStdAccessibleObject   140d2e   1      1
> LresultFromObject           140d3e   2      1
> __delayLoadHelper2@8        140d44   1      37
v _WinMain@16                 140f81   1      16
    > struct_offsets
    > parents
    v children
        call  ?AfxGetModuleState...   140f91
        call  __setmbcp               140fa2
        call  ?AfxTermLocalData@...   140fb3
        call  ?AfxCriticalTerm@@Y...  140fb8
        jmp   ?AfxTlsRelease@@Y...    140fbd
        call  ?AfxGetThread@@YG...    140fcd
        call  ?AfxGetModuleState...   140fd4
        call  ?AfxWinInit@@YGHP...    140fe8
        call  dword ptr [eax+90h]     140ff9
        call  dword ptr [eax+50h]     141007
        call  dword ptr [eax+60h]     141018
        call  dword ptr [eax+68h]     14101f
        call  dword ptr [eax+54h]     141028
        call  ?AfxWinTerm@@YGX...     14102d
> _fgetc                      14103b   2      24
> _setvbuf                    14113f   1      23
> foetoos                     14123d   2      8
```

```
; Attributes: library function bp-based frame

; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
_WinMain@16 proc near

hInstance= dword ptr  8
hPrevInstance= dword ptr  0Ch
lpCmdLine= dword ptr  10h
nShowCmd= dword ptr  14h

mov     edi, edi
push    ebp
mov     ebp, esp
pop     ebp
jmp     loc_140FC2
```

```
loc_140FC2:
mov     edi, edi
push    ebp
mov     ebp, esp
push    ebx
push    esi
push    edi
or      ebx, 0FFFFFFFFh
call    ?AfxGetThread@@YGPAVCWinThread@@XZ ; AfxGetThread(void)
mov     esi, eax
call    ?AfxGetModuleState@@YGPAVAFX_MODULE_STATE@@XZ ; AfxGetModuleState(void)
push    [ebp+nShowCmd]  ; int
mov     edi, [eax+4]
push    [ebp+lpCmdLine] ; wchar_t *
push    [ebp+hPrevInstance] ; HINSTANCE
push    [ebp+hInstance] ; HINSTANCE
```

```
100.00% (562,-41) (87,0) 00040381 00140F81: WinMain(x,x,x,x) (Synchronized with Hex View-1)
```

```
__tmainCRTStartup
call _WinMain@16
_WinMain@16
AfxTlsRelease@@YGXXZ    call ?AfxGetThread@@YGPAVCWinThread@@XZ    call ?AfxWinInit@@YGHPAUHINSTANCE__@@0PA_W
xTlsRelease@@YGXXZ    ?AfxGetThread@@YGPAVCWinThread@@XZ    ?AfxWinInit@@YGHPAUHINSTANCE__@@0PA_W@Z
100.00% (1416,-168) (12,155) -1.0
```

```
CTO Function Lister <class 'PyQt5.QtWidgets.QWidget'>
Reloaded CTO Function Lister.
Launching CTO (Call Tree Overviewer) ...
For the first execution, CTO will analyze all functions to build the call tree. Please wait for a while.
get data from global variable
CTO <class 'PyQt5.QtWidgets.QSplitter'>
Launched CTO.
Then press 'H' to see the help after setting focus to the widget.
```

```
AU:  idle   Down    Disk: 30GB
```

Figure 1: The CTO and CTO Function Lister Launch Buttons and Display Panels

*1 The presentation given at VB2021 localhost is available at the following URL. CTO (Call Tree Overviewer) yet another function call tree viewer (https://vblocalhost.com/conference/presentations/cto-call-tree-overviewer-yet-another-function-call-tree-viewer/). CTO and CTO Function Lister are also published on my GitHub repository (https://github.com/herosi/CTO).

*2 IDA Pro (https://hex-rays.com/ida-pro/) is a disassembler and decompiler, essential tools for malware analysts. CTO and CTO Function Lister were written using the IDAPython API.

© Internet Initiative Japan Inc.

2.2 Detecting Encryption/Decryption and Encoding/Decoding Routines

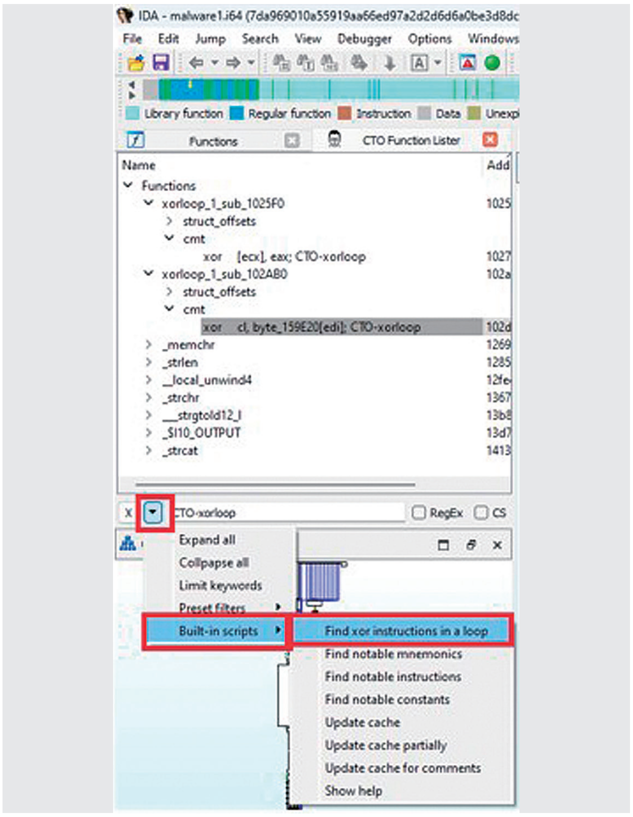
As you no doubt know, malware authors often encrypt or encode communications and config data to make them difficult to detect. In some cases, the authors use existing encryption algorithms such as AES and RC4, and in others, they simply use xor instructions to create custom encodings. In addition to custom encodings that explicitly use xor, many known cryptographic algorithms, including the afore-mentioned, also include xor instructions. Further, loop structures are inevitably needed when cryptographically processing data that is longer than the CPU registers. So, CTO has a built-in command that traverses the functions known to IDA, finds xor instructions, checks if they are in loops, and displays the results. If a function name in the results has not been changed from the default, it is renamed by appending “xorloop_” so that it can easily be found via the function name.

Figure 2 shows how to execute that command. It can be executed from CTO via a shortcut also, but here I show how to execute it from the CTO Function Lister menu.

First, click the dropdown menu button and select “Built-in scripts”, then select “Find xor instructions in a loop”. It depends on the size of the program being analyzed, but with the sample malware (code section size of around 280KB), the command completed in around 2–3 seconds.

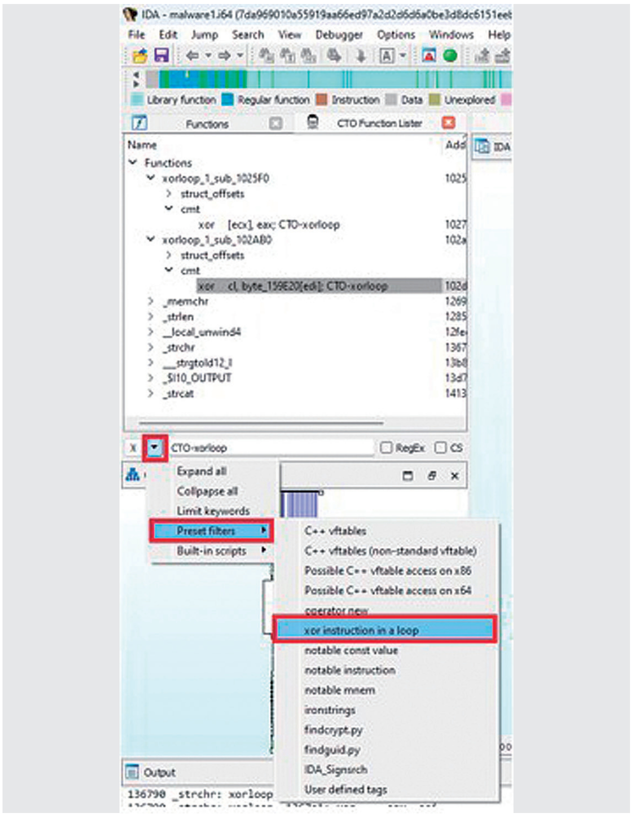
The results are displayed in the Output window, and it can also filter and display only the relevant functions in CTO Function Lister. To do this, open the dropdown menu again and select “Preset filters” and then “xor instruction in a loop” as shown in Figure 3.

This will result in only functions that have an xor instruction inside a loop being listed, as Figure 4 shows. With FLIRT



```
Name                                              Add
Functions
  xorloop_1_sub_1025F0                            1025
    struct_offsets
    cmt
      xor    [ecx], eax; CTO-xorloop              1027
  xorloop_1_sub_102AB0                            102a
    struct_offsets
    cmt
      xor    cl, byte_159E20[edi]; CTO-xorloop    102d
  _memchr                                         1269
  _strlen                                         1285
  __local_unwind4                                 12fe
  _strchr                                         1367
  ___strgtold12_l                                 13b8
  _$I10_OUTPUT                                    13d7
  _strcat                                         1413
```

Figure 2: Executing the “Find xor instructions in a loop” command



```
Name                                              Add
Functions
  xorloop_1_sub_1025F0                            1025
    struct_offsets
    cmt
      xor    [ecx], eax; CTO-xorloop              1027
  xorloop_1_sub_102AB0                            102a
    struct_offsets
    cmt
      xor    cl, byte_159E20[edi]; CTO-xorloop    102d
  _memchr                                         1269
  _strlen                                         1285
  __local_unwind4                                 12fe
  _strchr                                         1367
  ___strgtold12_l                                 13b8
  _$I10_OUTPUT                                    13d7
  _strcat                                         1413
```

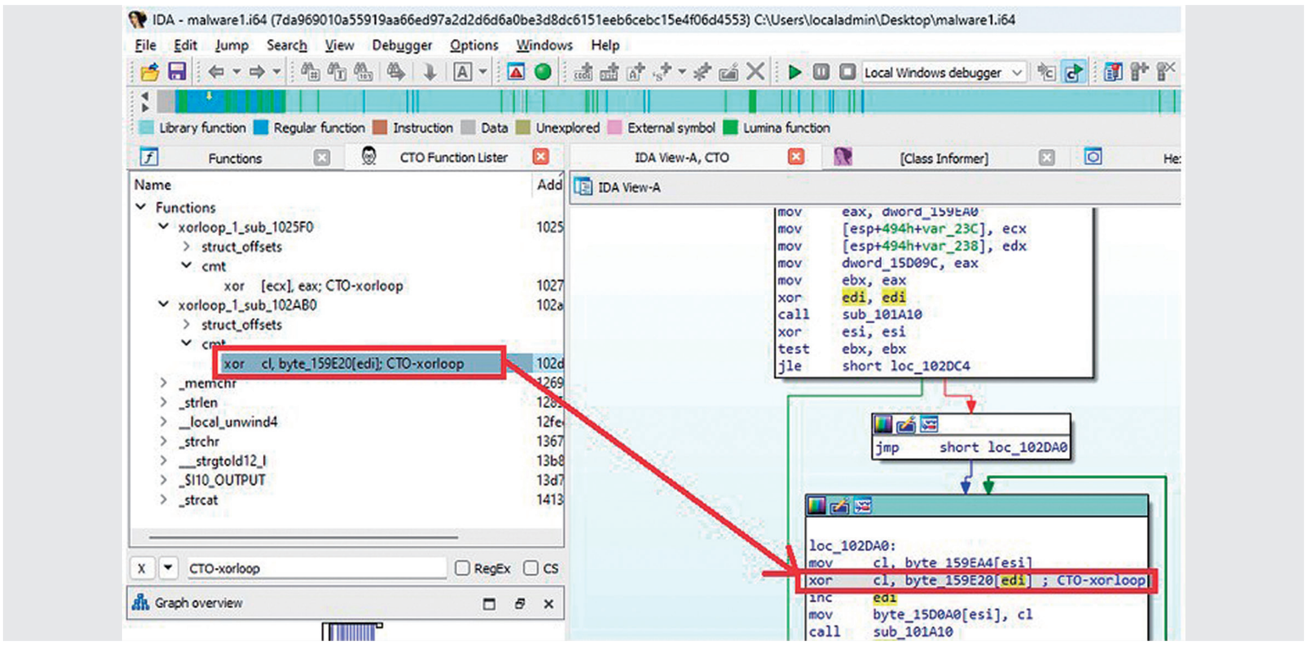
```
136790 _strchr: xorloop
```

Figure 3: Displaying “xor instruction in a loop”

and Lumina[*3], IDA will rename statically linked C (and other language) library functions to an extent. As you can see, all but the first two functions have been named. So, we need to start by looking at the first two functions (sub_1025F0 and sub_102AB0). The aforementioned command adds the comment "CTO-xorloop" to corresponding xor instructions, and these are displayed in CTO Function Lister using a filter. They are in the cmt subtrees. By clicking a line in CTO Function Lister, you can jump to the corresponding address in IDA's disassembly view to inspect the surrounding code.

Looking at the code surrounding the two functions obtained using the command above, one is a routine used to decrypt the payload downloaded from a malicious server, and the other is a routine used to decrypt C&C config data (host name, IP address, etc.) that is hard-coded into the sample. You can find key code blocks like this instantly with these tools.

Here we looked at custom xor-based encodings as an example, but encryption algorithms such as AES and hash algorithms such as SHA256 and MD5 often have characteristic magic values[*4] and tables. Third-party scripts and plugins for detecting such characteristics, such as findcrypt[*5] and IDA Signsrch[*6], have been released. CTO Function Lister recognizes the results of these as well and can filter and display results based on them. Using these tools in combination lets you efficiently discover encryption/decryption and encoding/decoding routines, and quickly check the surrounding code.



```
Name                                        Add
Functions
  xorloop_1_sub_1025F0                      1025
    struct_offsets
    cmt
        xor    [ecx], eax; CTO-xorloop      1027
  xorloop_1_sub_102AB0                      102a
    struct_offsets
    cmt
        xor    cl, byte_159E20[edi]; CTO-xorloop   102d
  _memchr                                   1269
  _strlen                                   128
  __local_unwind4                           12fe
  _strchr                                   1367
  ___strgtold12_l                           13b8
  _$I10_OUTPUT                              13d7
  _strcat                                   1413
```

CTO-xorloop

```
mov     eax, dword_159EA0
mov     [esp+494h+var_23C], ecx
mov     [esp+494h+var_238], edx
mov     dword_15D09C, eax
mov     ebx, eax
xor     edi, edi
call    sub_101A10
xor     esi, esi
test    ebx, ebx
jle     short loc_102DC4

jmp     short loc_102DA0

loc_102DA0:
mov     cl, byte_159EA4[esi]
xor     cl, byte_159E20[edi] ; CTO-xorloop
inc     edi
mov     byte_15D0A0[esi], cl
call    sub_101A10
```

Figure 4: Results of "xor instruction in a loop" and the Surrounding Code

*3 FLIRT and Lumina are part of IDA's functionality. Both can use pattern matching to detect and rename known functions. FLIRT refers to a local database, while Lumina refers to a cloud-based database. With Lumina, IDA users push function information to the cloud first. After that, when a function with the same pattern is found in a binary analyzed by another user, the function name on the Lumina server is applied to it.

*4 A magic value is a specific string or numerical value that is used as a marker to uniquely identify a header or footer of a certain format.

*5 FindCrypt (https://github.com/you0708/ida/tree/master/idapython_tools/findcrypt) is a third-party script for IDA Pro that uses pattern matching to detect tables and magic values of well-known encryption and hash algorithms. Several implementations of FindCrypt exist. The one referred to here is implemented in Python and is easy to extend, so this is the one I use.

*6 IDA Signsrch (https://sourceforge.net/projects/idasignsrch/) is a third-party plugin for IDA Pro that is used to detect cryptographic and hash algorithms in the same way as FindCrypt. While it is similar to FindCrypt, both tools sometimes differ in scope, so more than one of them is used at times.

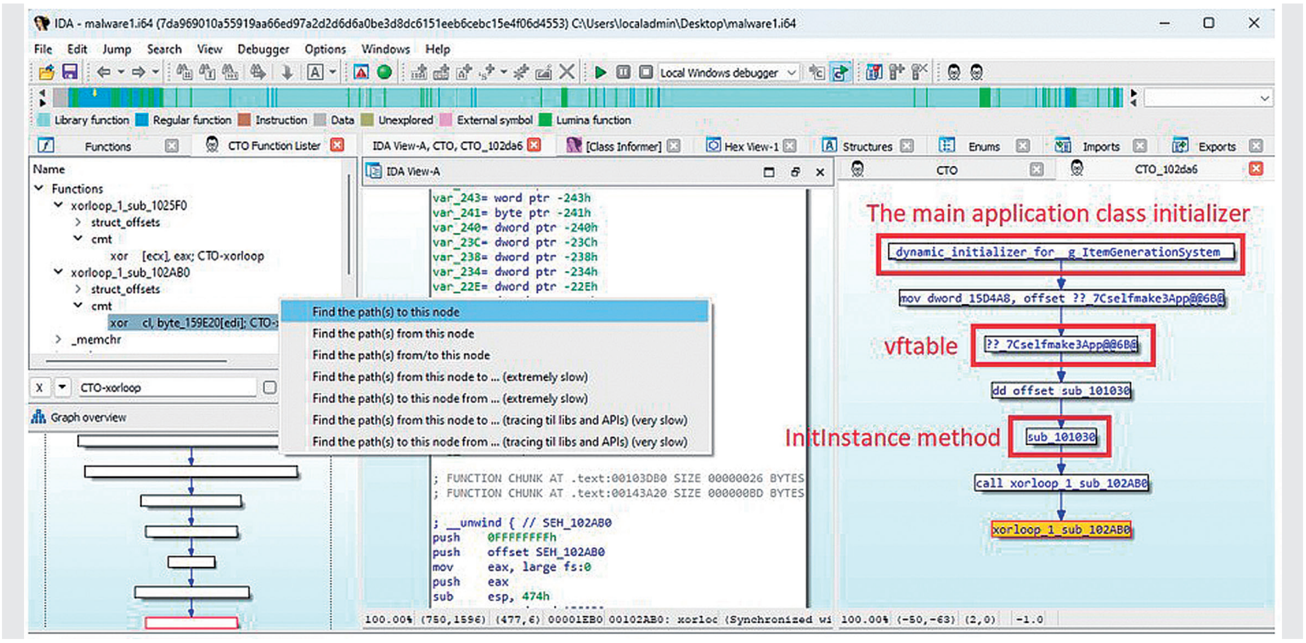
© Internet Initiative Japan Inc.

2.3 Path Exploration

CTO can display paths to or from an address. Figure 5 shows the result of right-clicking an xor instruction found using CTO Function Lister and selecting "Find the path(s) to this node". The results appear in a call tree graph on the right side of the window.

Although this graph shows the relationships between each function address and the code and data that refer to it, this is, unfortunately, not a perfect execution path. The reason for this is that even if a function contains a function pointer, it is not necessarily called right away. For example, the function pointer might be stored in a register or on a heap chunk, with the function called much later on. Indirect calls are often used in mechanisms like C＋＋ vftables. To find exactly where a function is executed, you have to track down the class instance, find all the code that refers to it, and find all the code that retrieves a function pointer from the vftable and executes it. It makes the code quite complicated. So, whenever code accesses a function pointer, CTO extracts the address and builds a parent-child relationship graph like this. This is still useful enough, though.

In the example here, a function called dynamic initializer is the first node of the path. This function is processed by the initterm[*7] function in the CRT (C-Runtime). Reading the code reveals that this malware is written using MFC. MFC applications must declare their main application class as a global variable. This declaration causes initterm to call the constructor of the main application class, encapsulated in a dynamic initializer, and the class instance is stored in a global variable.


IDA - malware1.i64 (7da969010a55919aa66ed97a2d2d6d6a0be3d8dc6151eeb6cebc15e4f06d4553) C:\Users\localadmin\Desktop\malware1.i64
File
Edit
Jump
Search
View
Debugger
Options
Windows
Help
Local Windows debugger
Library function
Regular function
Instruction
Data
Unexplored
External symbol
Lumina function
Functions
CTO Function Lister
IDA View-A, CTO, CTO_102da6
[Class Informer]
Hex View-1
Structures
Enums
Imports
Exports
IDA View-A
CTO
CTO_102da6
Name
Functions
xorloop_1_sub_1025F0
struct_offsets
cmt
xor [ecx], eax; CTO-xorloop
xorloop_1_sub_102AB0
struct_offsets
cmt
xor cl, byte_159E20[edi]; CTO-
_memchr
CTO-xorloop
Graph overview
Find the path(s) to this node
Find the path(s) from this node
Find the path(s) from/to this node
Find the path(s) from this node to ... (extremely slow)
Find the path(s) to this node from ... (extremely slow)
Find the path(s) from this node to ... (tracing til libs and APIs) (very slow)
Find the path(s) to this node from ... (tracing til libs and APIs) (very slow)
The main application class initializer
vftable
InitInstance method

```
var_243= word ptr -243h
var_241= byte ptr -241h
var_240= dword ptr -240h
var_23C= dword ptr -23Ch
var_238= dword ptr -238h
var_234= dword ptr -234h
var_22E= dword ptr -22Eh

; FUNCTION CHUNK AT .text:00103DB0 SIZE 00000026 BYTES
; FUNCTION CHUNK AT .text:00143A20 SIZE 000000BD BYTES

; __unwind { // SEH_102AB0
push    0FFFFFFFFh
push    offset SEH_102AB0
mov     eax, large fs:0
push    eax
sub     esp, 474h
```

```
dynamic_initializer_for__g_ItemGenerationSystem_
mov dword_15D4A8, offset ??_7Cselfmake3App@@6B@
??_7Cselfmake3App@@6B@
dd offset sub_101030
sub_101030
call xorloop_1_sub_102AB0
xorloop_1_sub_102AB0
```

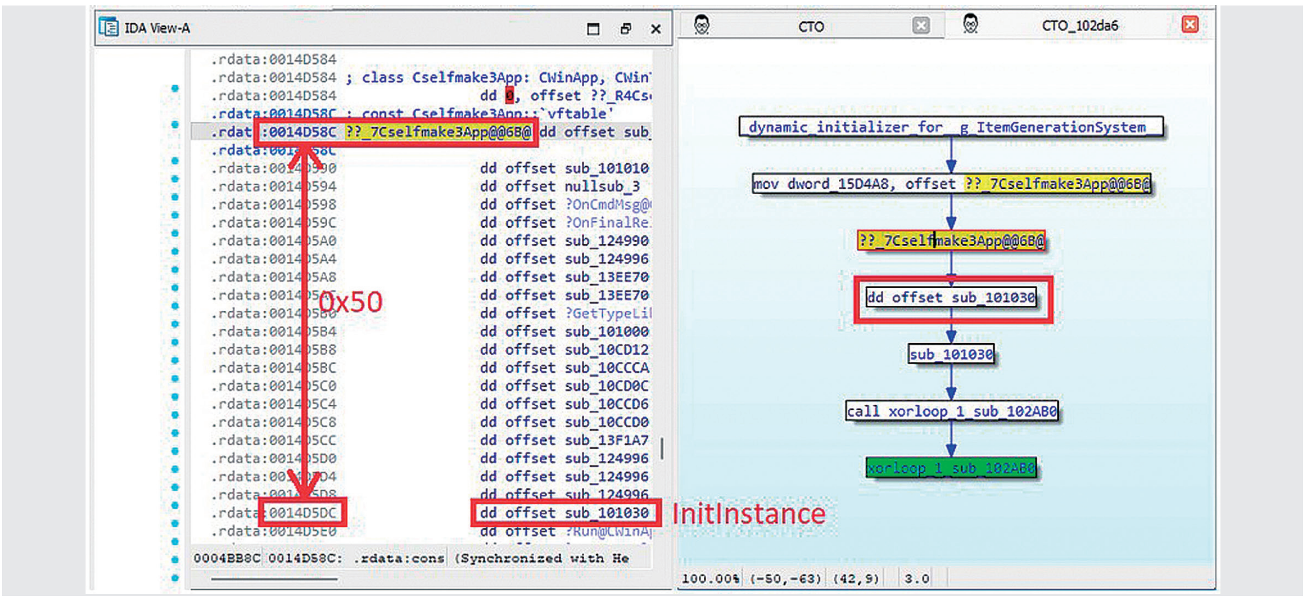
```
100.00% (750,1596) (477,6) 00001EB0 00102AB0: xorloc (Synchronized wi 100.00% (-50,-63) (2,0) -1.0
```

Figure 5: Path Exploration

[*7] initterm (https://learn.microsoft.com/cpp/c-runtime-library/reference/initterm-initterm-e) is a function that initializes global objects within the CRT before executing the main function. When initterm is called within the CRT, it takes global variables as its first and second arguments, so these are relatively easy to find even if IDA does not recognize this function. initterm executes the function pointers between the addresses specified by its two arguments in sequence. Each function pointer is encapsulated in dynamic initializer code (https://learn.microsoft.com/cpp/c-runtime-library/crt-initialization). Within that code, the global object's constructor is executed, and the class instance is stored in a global variable.

global variable. The function name displayed in the panel is automatically assigned by Lumina[*8], and clearly the part of the name following "for" is wrong. On the other hand, the path shown by CTO indicates there is access to a vftable with the class name Cselfmake3App in the constructor code. It can also be confirmed from the class inheritance hierarchy, which can be obtained from Class Informer[*9], that this class inherits the CWinApp class. These facts make it clear that Cselfmake3App is the main application class of this malware..

Next, Cselfmake3App's vftable connects to a function called sub_101030. CTO extracts and caches access to global variables that exist within functions. In particular, if it finds the string "vftable" or "vtable" at the beginning of a variable name or at the end of the comment attached to its address, it treats the global variable as a vftable, parses the table, and follows certain rules to recognize the function pointer group as belonging to that vftable. Since IDA can recognize RTTI, a string containing "vftable" is added to the comment for this address. So, the vftable analysis is executed when CTO is run for the first time, and thus within CTO, sub_101030 is already recognized as part of this vftable. So, when access to a function belonging to the vftable occurs, CTO can connect this function pointer as a virtual method. Figure 6 shows the IDA screen when the Cselfmake3App vftable node (third from the top, "??_7Cselfmake3App@@6B@") in CTO is clicked. IDA View-A shows a series of function pointers of the vftable. We can see that sub_101030 is located at an offset of 0x50 from the beginning of it. Incidentally, in 32-bit MFC main application classes, there is a virtual method called InitInstance at an offset of 0x50 of the vftable. Hence, sub_101030 is InitInstance.



```
.rdata:0014D584
.rdata:0014D584 ; class Cselfmake3App: CWinApp, CWin
.rdata:0014D584                 dd 0, offset ??_R4Cs
.rdata:0014D58C ; const Cselfmake3App::`vftable'
.rdata:0014D58C ??_7Cselfmake3App@@6B@ dd offset sub_
.rdata:0014D58C
.rdata:0014D590                 dd offset sub_101010
.rdata:0014D594                 dd offset nullsub_3
.rdata:0014D598                 dd offset ?OnCmdMsg@
.rdata:0014D59C                 dd offset ?OnFinalRe
.rdata:0014D5A0                 dd offset sub_124990
.rdata:0014D5A4                 dd offset sub_124996
.rdata:0014D5A8                 dd offset sub_13EE70
.rdata:0014D5AC                 dd offset sub_13EE70
.rdata:0014D5B0                 dd offset ?GetTypeLi
.rdata:0014D5B4                 dd offset sub_101000
.rdata:0014D5B8                 dd offset sub_10CD12
.rdata:0014D5BC                 dd offset sub_10CCCA
.rdata:0014D5C0                 dd offset sub_10CD0C
.rdata:0014D5C4                 dd offset sub_10CCD6
.rdata:0014D5C8                 dd offset sub_10CCD0
.rdata:0014D5CC                 dd offset sub_13F1A7
.rdata:0014D5D0                 dd offset sub_124996
.rdata:0014D5D4                 dd offset sub_124996
.rdata:0014D5D8                 dd offset sub_124996
.rdata:0014D5DC                 dd offset sub_101030
.rdata:0014D5E0                 dd offset ?Run@CWinA
0004BB8C 0014D58C: .rdata:cons (Synchronized with He
```

```
dynamic_initializer_for__g_ItemGenerationSystem_
mov dword_15D4A8, offset ??_7Cselfmake3App@@6B@
??_7Cselfmake3App@@6B@
dd offset sub_101030
sub_101030
call xorloop_1_sub_102AB0
xorloop_1_sub_102AB0
100.00% (-50,-63) (42,9)  3.0
```

Figure 6: MFC Main Application Class vftable and InitInstance Function

*8 As mentioned, Lumina uses names provided by ordinary users, so the accuracy of any given name depends on the skill of the user who created it. Thus, names are often unreliable and should be taken only as a reference. The example here also shows an incorrect name.

*9 Class Informer is a third-party plugin for IDA Pro. It is a tool that can be used to analyze C++ RTTI (Runtime Type Information) and identify class names and the class inheritance hierarchy (https://sourceforge.net/projects/classinformer/). RTTI analysis itself has been possible since IDA 7.0, but I still use this plugin as it remains superior in some respects—e.g., hierarchy display, class search functionality. An improved version that can restore class information on PE32 binaries with the 64-bit version of IDA is also available on my GitHub repository (https://github.com/herosi/classinformer-ida8). I released this because IDA began phasing out 32-bit IDA starting with 8.0 and moving to the 64-bit version only, and the original Class Informer was unable to parse PE32 on the 64-bit version of IDA.

Once the MFC application has processed the main application class constructor within the CRT, as described above, it executes several methods such as InitInstance and Run within the WinMain function (specifically, AfxWinMain). In particular, according to the MFC application document, you must override the InitInstance function[*10], so in many cases, this is effectively the malware's main function. The malware we are looking at here also calls InitInstance (sub_101030), and it is easy to see that the routine (sub_102AB0) to decod the malware config is called from the function by using the CTO call tree graph.

Another feature of CTO's path exploration is the ability to create paths even for global variables (including strings) as long as you have a cross-reference[*11]. IDA also has a feature called Proximity View (or Browser), but it can only be used for functions. This is one advantage of using CTO.

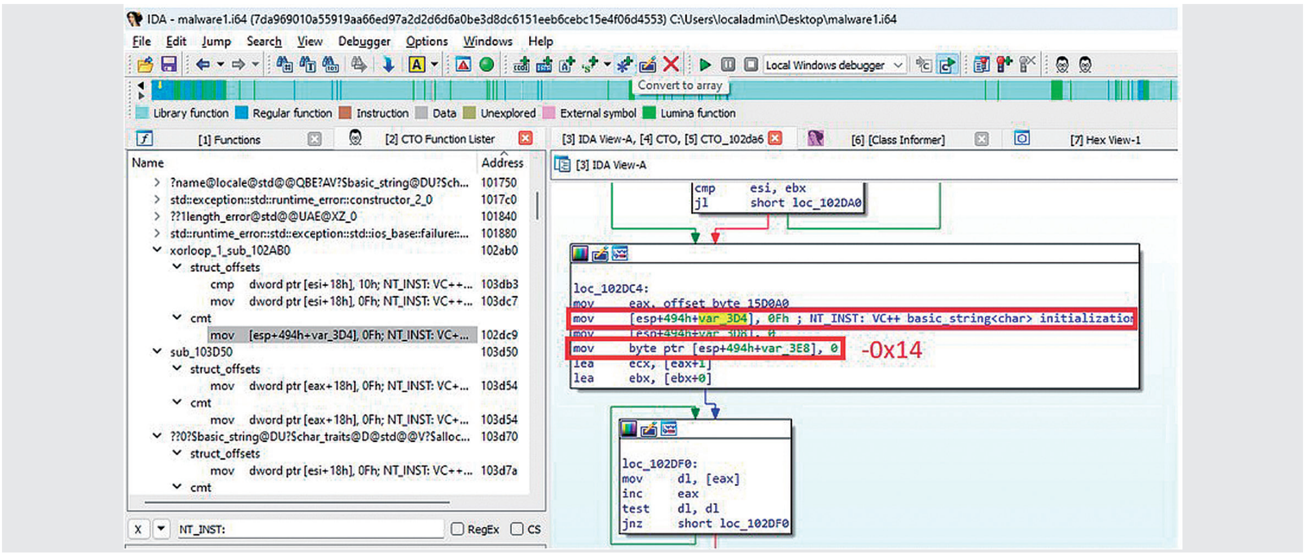
Note that in order to use the CTO Function Lister features as described here, you first need to run CTO.

2.4 Detecting std::string / std::wstring

A lot of malware written in C++ uses std::string and std::wstring for string manipulation. The constructors and some methods of these classes are expanded inline, which can make it hard to determine that these classes are being used at first glance. But because the code that initializes the class layout uses a distinctive initial value, they can be detected with a few simple pattern matching albeit with a few false positives.

These classes can be found by selecting "Built-in scripts", "Find notable instructions" from the CTO Function Lister drop-down menu introduced earlier. You can also select "Preset filters", "Notable instruction" from the drop-down menu to filter the results of this command.

As an example, we'll look at std::string as used in the code that parses the config data decoded by the malware. Figure 7 shows the initialization code for std::string detected by CTO. In the first red box in the figure, the stack variable is



```
Name                                                          Address
>  ?name@locale@std@@QBE?AV?$basic_string@DU?$ch...           101750
>  std::exception::std::runtime_error::constructor_2_0        1017c0
>  ??1length_error@std@@UAE@XZ_0                              101840
>  std::runtime_error::std::exception::std::ios_base::failure::...  101880
v  xorloop_1_sub_102AB0                                       102ab0
   v  struct_offsets
         cmp   dword ptr [esi+18h], 10h; NT_INST: VC++...     103db3
         mov   dword ptr [esi+18h], 0Fh; NT_INST: VC++...     103dc7
   v  cmt
         mov   [esp+494h+var_3D4], 0Fh; NT_INST: VC+...       102dc9
v  sub_103D50                                                 103d50
   v  struct_offsets
         mov   dword ptr [eax+18h], 0Fh; NT_INST: VC+...      103d54
   v  cmt
         mov   dword ptr [eax+18h], 0Fh; NT_INST: VC+...      103d54
v  ??0?$basic_string@DU?$char_traits@D@std@@V?$alloc...      103d70
   v  struct_offsets
         mov   dword ptr [esi+18h], 0Fh; NT_INST: VC++...     103d7a
   v  cmt

NT_INST:
```

```
cmp     esi, ebx
jl      short loc_102DA0

loc_102DC4:
mov     eax, offset byte_15D0A0
mov     [esp+494h+var_3D4], 0Fh ; NT_INST: VC++ basic_string<char> initializatio
mov     [esp+494h+var_3D8], 0
mov     byte ptr [esp+494h+var_3E8], 0
lea     ecx, [eax+1]
lea     ebx, [ebx+0]

loc_102DF0:
mov     dl, [eax]
inc     eax
test    dl, dl
jnz     short loc_102DF0
```

Figure 7: Detecting std::string

*10 The following URL describes the methods that can be overridden and methods that must be overridden when deriving an application class from CWinApp. Only InitInstance is required (https://learn.microsoft.com/cpp/mfc/overridable-cwinapp-member-functions).

*11 Cross-references, also known as xrefs, are one of IDA's key features. This feature lists code and data referring to a specific address. There are two types: xrefs from and xrefs to. IDA can display and use both, and they are thus collectively called cross-references. CTO also uses cross-references to create parent-child relationships.

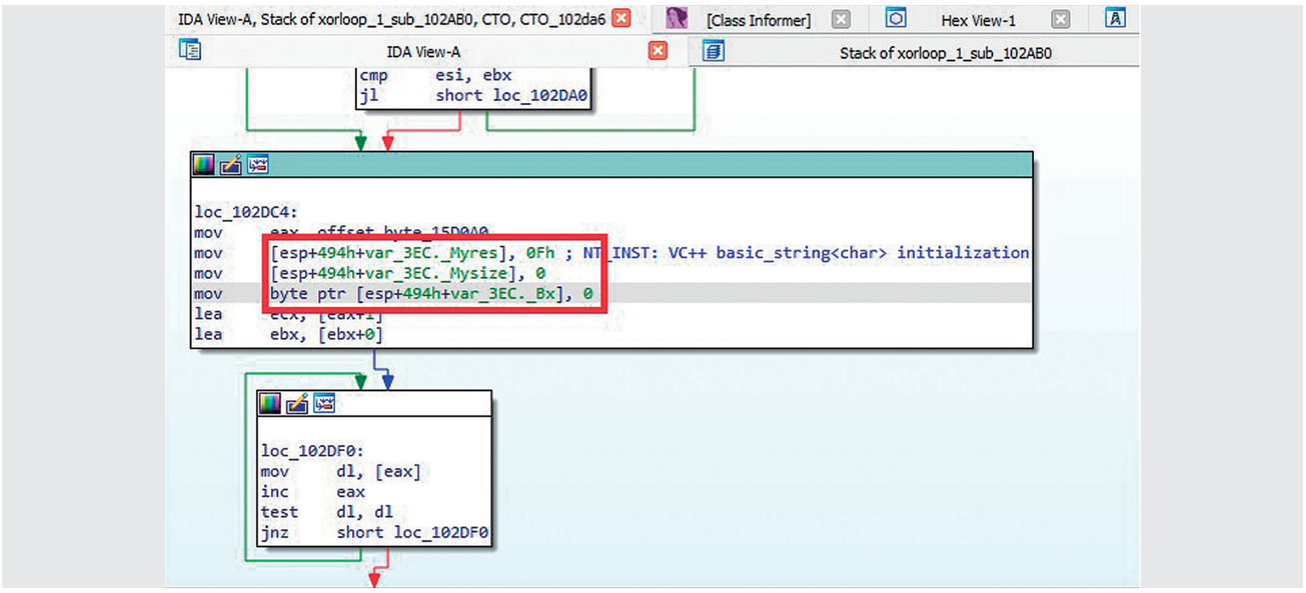
initialized with the immediate value 0xf. This is part of the initialization code for std::string that has been used in Visual Studio for many years. Two instructions below (second red box) is the code that initializes the beginning of the buffer (position -0x14 from the address initialized with 0xf above) with a 1-byte NULL character. When these instructions appear in a set like this, I consider this a use of std::string and apply that structure.

The class layout of std::string is undocumented, and we have determined there to be several patterns depending on which version of Visual Studio is used. On the other hand, I found the malware we are looking at here was compiled using Visual Studio 2008. So, loading the appropriate structure for that version and applying it to the top of the std::string instance on the stack results in a nice, clean recognition of std::string as shown in Figure 8.

2.5 CTO / CTO Function Lister in Practice

At GCC 2023 Singapore[*12] in February 2023, my colleague and I delivered a training course on malware analysis using the IDA plugins discussed here. At the end, we had people randomly form teams of four to six and presented them with characteristic functions and code obtained from the malware sample discussed here in CTF format, and asked them to analyze the malware in some game-like exercises.

While the participants were students that had been specially selected from various countries, many of them had no experience with IDA or reverse engineering, so we had



IDA View-A, Stack of xorloop_1_sub_102AB0, CTO, CTO_102da6
[Class Informer]
Hex View-1
IDA View-A
Stack of xorloop_1_sub_102AB0

```
cmp     esi, ebx
jl      short loc_102DA0

loc_102DC4:
mov     eax, offset byte_15D0A0
mov     [esp+494h+var_3EC._Myres], 0Fh ; NT_INST: VC++ basic_string<char> initialization
mov     [esp+494h+var_3EC._Mysize], 0
mov     byte ptr [esp+494h+var_3EC._Bx], 0
lea     ecx, [eax+1]
lea     ebx, [ebx+0]

loc_102DF0:
mov     dl, [eax]
inc     eax
test    dl, dl
jnz     short loc_102DF0
```

Figure 8: Applying the Structure to std::string and Recognizing Member Variables

*12 GCC (Global Cybersecurity Camp) (https://gcc.ac/) is a training program for students selected from eight, mainly Asian, countries (as of February 2023). GCC 2023 was the fifth such camp and was held in Singapore in February, and we presented there on the content in this chapter (https://gcc.ac/gcc_2023/lectures/#re-verse-engineering-malware-written-in- c-with-ida-and-semi-automated-scripts). Hoping to contribute to the industry, IIJ has continued to provide training at this event since its inception.

© Internet Initiative Japan Inc.

to give them a brief lecture before starting the CTF exercise. Yet by using techniques like those presented here to save time, the best teams were able to finish most of the malware analysis in around an hour and a half. Over two-thirds of the teams got through most of the important parts in around three hours. This exercise was solely about reverse engineering, so we did not give the teams the executable file itself. They only received an IDA database with the file loaded in. What the malware does is simple, so it is easy to get an overall idea of what's happening under the hood once it is executed, but I deliberately made things harder for the participants because the ability to properly dissect malware by reading the code is also crucial. Even under these conditions, the students used the tools and techniques presented to flesh out their understanding, and it was exciting to see them develop their skills so quickly.

2.6 Final Thoughts

Aside from what I have described here, CTO and CTO Function Lister also implement a range of features that I needed based on past malware analysis. I plan to continue implementing new ideas, such as automation, going forward. I hope these tools prove useful in your malware analysis endeavors.

Hiroshi Suzuki
Malware & Forensic Analyst, Office of Emergency Response and Clearinghouse for Security Information, Advanced Security Division, IIJ
As a member of IIJ-SECT (IIJ's CSIRT), Mr. Suzuki is engaged in internal and customer incident response. He is primarily a malware analyst and forensic investigator. Drawing on the insight and knowledge from this work, he has spoken at international conferences including Black Hat (USA, Europe, Asia), Virus Bulletin, and FIRST TC, as well as at a range of domestic organizations including Japan's National Center of Incident Readiness and Strategy for Cybersecurity (NISC), the Ministry of Internal Affairs and Communications, the Ministry of Justice, IPA, and the National Institute of Advanced Industrial Science and Technology (AIST). He also delivers training courses for experts and students at domestic and international conferences and training programs, including Black Hat USA, FIRST (Annual, TC), Global Cybersecurity Camp, MWS, Japan's National Security Camp, and Cyber Colosseo. He was the first Japanese trainer to be selected for Black Hat USA, where he has given trainings on incident response using forensic investigation and malware analysis. He has dedicated over 17 years to these areas.