VB2021 localhost

7 - 8 October, 2021 / vblocalhost.com

CTO (CALL TREE OVERVIEWER): YET ANOTHER FUNCTION CALL TREE VIEWER

Hiroshi Suzuki

Internet Initiative Japan, Japan

ABSTRACT

CTO (Call Tree Overviewer) is an *IDA* plug-in designed to display an overview of function call relationships.

When we analyse malware, we tend to lose track of which function we are investigating because there are thousands of functions in a binary file. We also need to find out what path the function we are investigating took to get to this place, but if the function location we are looking into is very deep, it can be tedious to follow the path manually. This is why we need function call trees and path explorers.

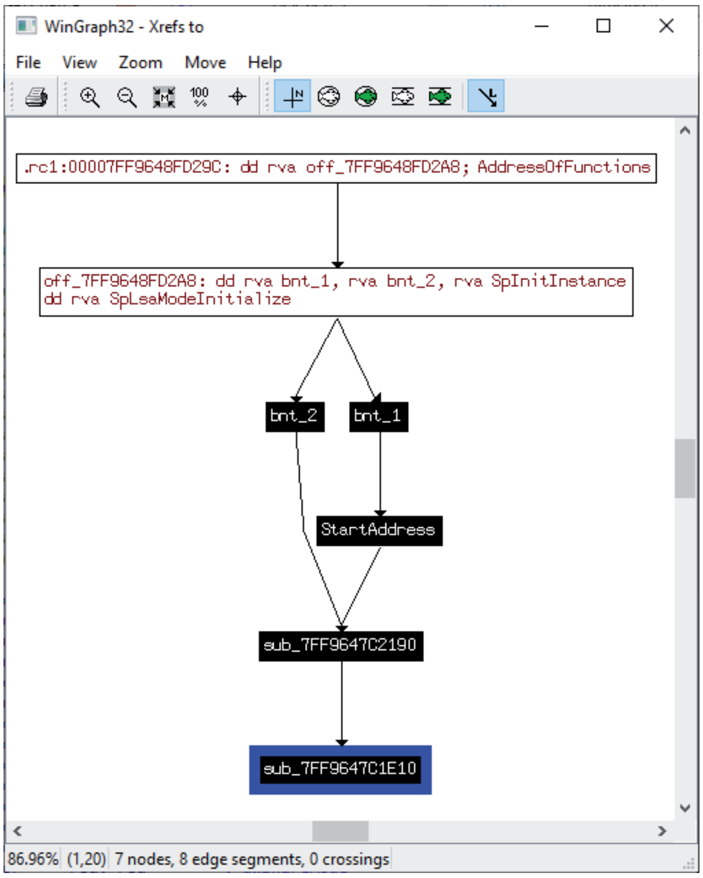
CTO is a field-oriented and practical tool that is designed to solve some disadvantages of *IDA*. It is able to display not only the function call tree as a graphical user interface including internal function calls and API calls, but also referred strings and repeatable comments, which are normally input by a user, if necessary, so that, in one view, you can instantly recognize the relationships between functions and important clues in the currently displayed function.

In addition, it is docked next to the *IDA Pro* disassembly view by default. If you click on a caller or a callee node on the tree graph, the address on *IDA View* will automatically be synchronized with it, and vice versa. By default, static linked libraries, which it is not commonly necessary to look into, and very deep function calls are collapsed to avoid overcomplicating the graph – but you can, of course, dig deeper or filter them out. You can find paths between two given functions as well. Every feature on this tool has its own shortcut key, so that you can handle the tool quickly.

1. MOTIVATION

There are already two features related to function call tree graphs in *IDA Pro*. One is the 'Graphs' or 'Chart' feature, and the other is called 'Proximity Browser'. Given the existence of these, why did I decide to develop this plug-in? Let me explain.

The 'Graphs' or 'Chart' feature does not generate clickable graphs and lacks a path filter feature because it generates a graph using a modified version of WinGraph[1].



```
.rc1:00007FF9648FD29C: dd rva off_7FF9648FD2A8; AddressOfFunctions
```

```
off_7FF9648FD2A8: dd rva bnt_1, rva bnt_2, rva SpInitInstance
dd rva SpLsaModeInitialize
```

Figure 1: Graphs feature.

Meanwhile, 'Proximity Browser' is a more sophisticated feature. You can click nodes and there is a filter feature as well as a path discovery feature.

[1] *Hex-Rays* provides the source code of it at [1].

However, it is still not suitable for grasping the whole picture of the relationships because it tends to follow unnecessary paths and xrefs, and the area per node is large.

Figure 2 shows a comparison of CTO (on the left) with Proximity Browser (on the right) at the same address. As you can see, Proximity Browser traced unnecessary nodes.

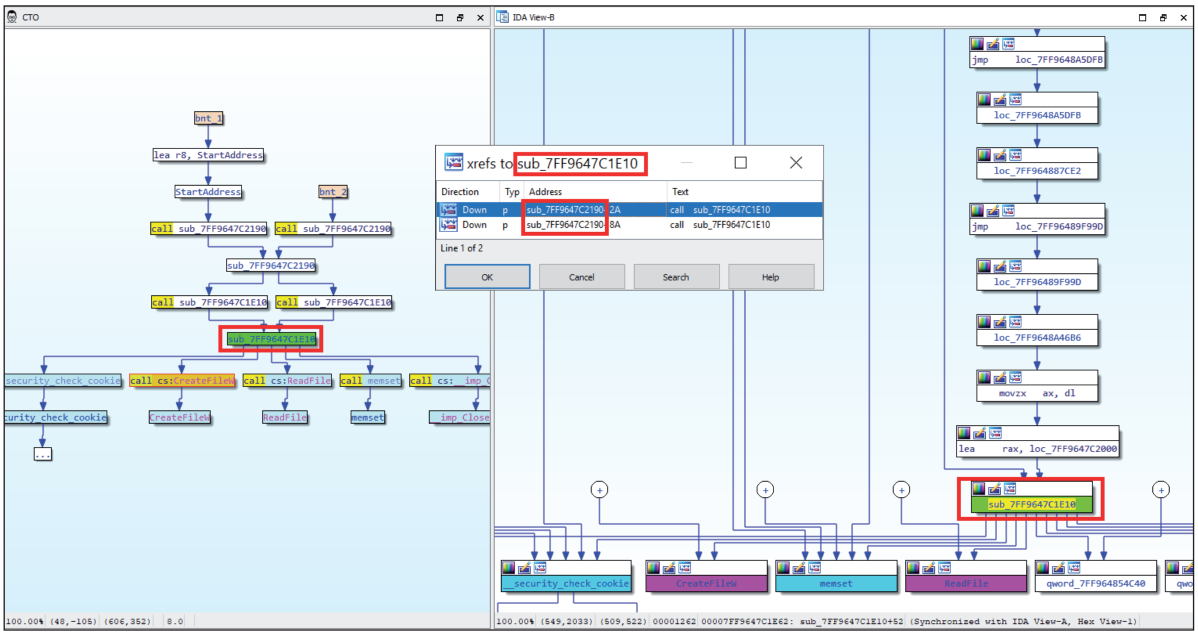


Figure 2: CTO vs Proximity Browser.

In the graph shown in Figure 3, you can also see that Proximity Browser shows four paths, while *IDA*'s xrefs window shows only one xref.

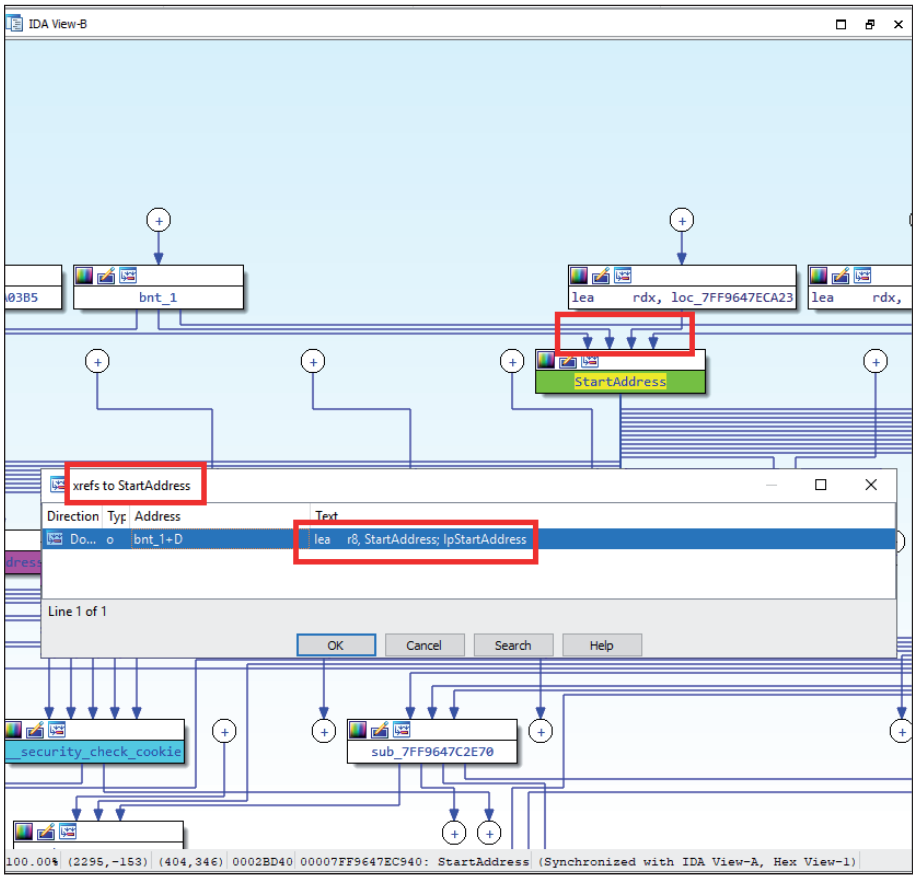

IDA View-B
bnt_1
lea rdx, loc_7FF9647ECA23
lea rdx,
StartAddress
xrefs to StartAddress
Direction
Typ
Address
Text
Do... o bnt_1+D lea r8, StartAddress; lpStartAddress
Line 1 of 1
OK
Cancel
Search
Help
security_check_cookie
sub_7FF9647C2E70
100.00% (2295,-153) (404,346) 0002BD40 00007FF9647EC940: StartAddress (Synchronized with IDA View-A, Hex View-1)

Figure 3: Unnecessary xrefs shown by Proximity Browser.

Furthermore, by default Proximity Browser always displays all types of nodes, including strings or global/static variables, not only functions. Large numbers of nodes makes it difficult to understand the relationships. If we were to dig into several nodes, the graph would be too complex.

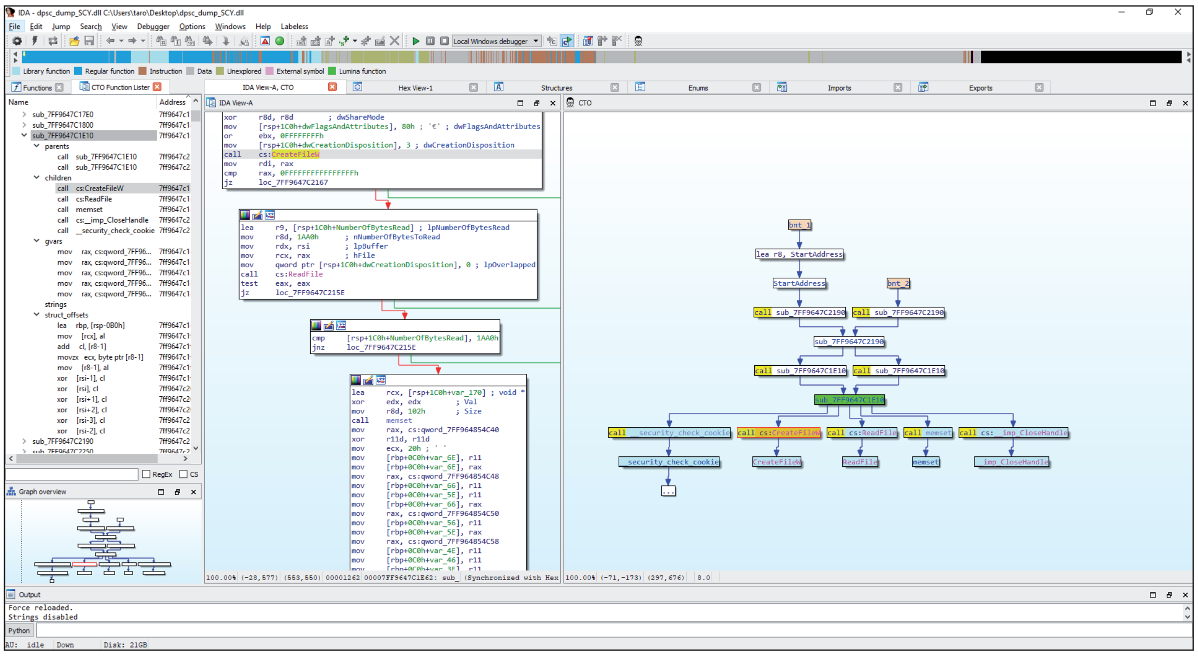
In addition, with Proximity Browser we cannot create a graph to find paths to a pointer on IAT, although it can create paths from a general function pointer. This was also a motivation for me to create the plug-in.

It was these problems that led me to develop CTO.

2. INTRODUCING CTO

2.1. What is CTO?

CTO is an abbreviation for Call Tree Overviewer, which is mainly designed to create a function call tree.



```
Name                                Address
 > sub_7FF9647C17E0                 7ff9647c1
 > sub_7FF9647C1800                 7ff9647c1
 v sub_7FF9647C1E10                 7ff9647c1
   v parents
       call  sub_7FF9647C1E10       7ff9647c2
       call  sub_7FF9647C1E10       7ff9647c2
   v children
       call  cs:CreateFileW         7ff9647c1
       call  cs:ReadFile            7ff9647c1
       call  memset                 7ff9647c1
       call  cs:__imp_CloseHandle   7ff9647c2
       call  __security_check_cookie 7ff9647c2
   v gvars
       mov   rax, cs:qword_7FF96... 7ff9647c1
       mov   rax, cs:qword_7FF96... 7ff9647c1
       mov   rax, cs:qword_7FF96... 7ff9647c1
       mov   rax, cs:qword_7FF96... 7ff9647c1
       mov   rax, cs:qword_7FF96... 7ff9647c1
     strings
   v struct_offsets
       lea   rbp, [rsp-0B0h]        7ff9647c1
       mov   [rcx], al              7ff9647c1
       add   cl, [r8-1]             7ff9647c1
       movzx ecx, byte ptr [r8-1]   7ff9647c1
       mov   [r8-1], al             7ff9647c1
       xor   [rsi-1], cl            7ff9647c1
       xor   [rsi], cl              7ff9647c2
       xor   [rsi+1], cl            7ff9647c2
       xor   [rsi+2], cl            7ff9647c2
       xor   [rsi-3], cl            7ff9647c2
       xor   [rsi-2], cl            7ff9647c2
 > sub_7FF9647C2190                 7ff9647c2
 > sub_7FF9647C2250                 7ff9647c2
```

```
xor     r8d, r8d        ; dwShareMode
mov     [rsp+1C0h+dwFlagsAndAttributes], 80h ; '€' ; dwFlagsAndAttributes
or      ebx, 0FFFFFFFFh
mov     [rsp+1C0h+dwCreationDisposition], 3 ; dwCreationDisposition
call    cs:CreateFileW
mov     rdi, rax
cmp     rax, 0FFFFFFFFFFFFFFFFh
jz      loc_7FF9647C2167

lea     r9, [rsp+1C0h+NumberOfBytesRead] ; lpNumberOfBytesRead
mov     r8d, 1AA0h      ; nNumberOfBytesToRead
mov     rdx, rsi        ; lpBuffer
mov     rcx, rax        ; hFile
mov     qword ptr [rsp+1C0h+dwCreationDisposition], 0 ; lpOverlapped
call    cs:ReadFile
test    eax, eax
jz      loc_7FF9647C215E

cmp     [rsp+1C0h+NumberOfBytesRead], 1AA0h
jnz     loc_7FF9647C215E

lea     rcx, [rsp+1C0h+var_170] ; void *
xor     edx, edx        ; Val
mov     r8d, 102h       ; Size
call    memset
mov     rax, cs:qword_7FF964854C40
xor     r11d, r11d
mov     ecx, 20h ; ' '
mov     [rbp+0C0h+var_6E], r11
mov     [rbp+0C0h+var_6E], rax
mov     rax, cs:qword_7FF964854C48
mov     [rbp+0C0h+var_66], r11
mov     [rbp+0C0h+var_5E], r11
mov     [rbp+0C0h+var_66], rax
mov     rax, cs:qword_7FF964854C50
mov     [rbp+0C0h+var_56], r11
mov     [rbp+0C0h+var_5E], rax
mov     rax, cs:qword_7FF964854C58
mov     [rbp+0C0h+var_4E], r11
mov     [rbp+0C0h+var_46], r11
```

```
100.00% (-28,577) (553,550) 00001262 00007FF9647C1E62: sub_ (Synchronized with Hex
100.00% (-71,-173) (297,676) 8.0
```

```
Force reloaded.
Strings disabled
```

```
AU:  idle   Down   Disk: 21GB
```

Figure 4: CTO (Call Tree Overviewer).

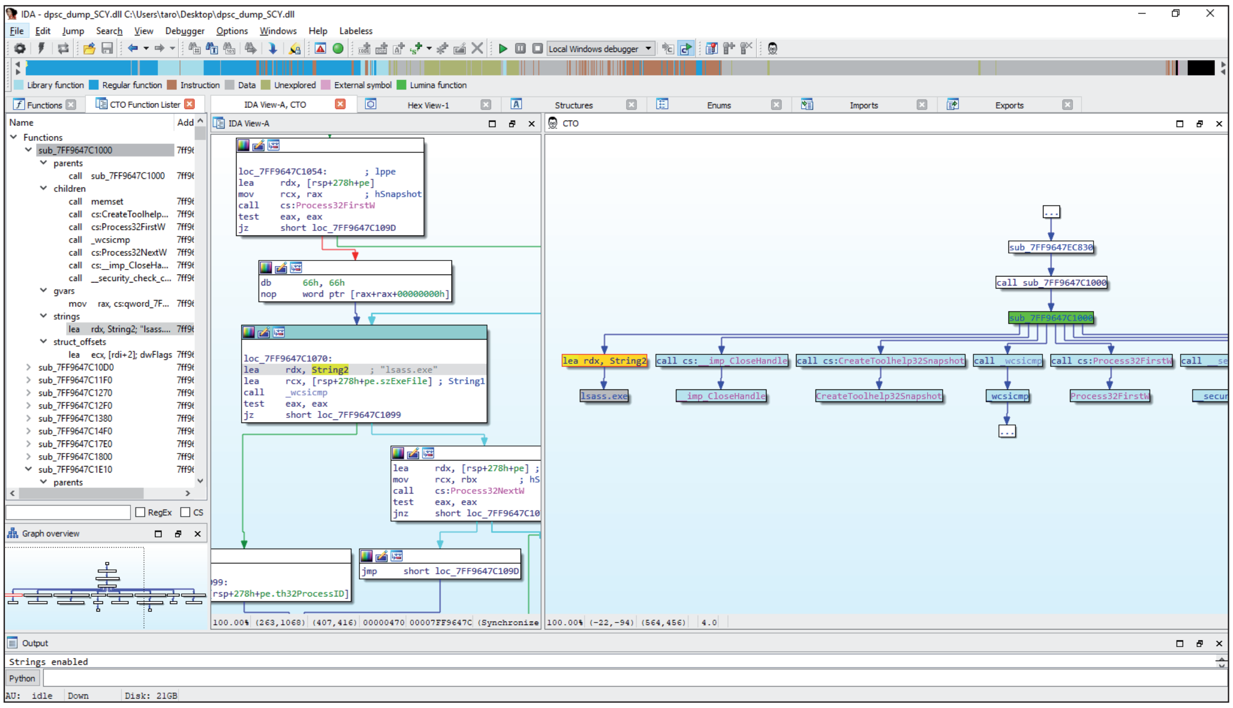
I implemented not only a function call tree creation feature, but several malware analysis techniques, such as creating a function summary from API calls and strings to rapidly identify the function's role, finding paths based on the target address, and creating a tree with least nodes.

This is the current list of elements that CTO can extract from functions for a function summary:

- Function calls / function pointer references
    - General functions
    - APIs
    - Static linked libraries
    - Unresolved indirect calls
- Global static variables references
- Strings references
- (Possible) structure offset references
- Repeatable comments
- Specific general comments (with keywords for third-party tool corroboration)

Of these, only function calls are displayed as nodes by default. This is because, to be a practical tool, I think the most important point is to keep the graph simple but sufficient. If you want to grasp function relationships, you will not need such detailed information. That is why, by default, CTO displays only function call related nodes.

On the other hand, if you display all types of nodes and cut other parent and grandchild nodes, you will be able to focus on a function. Depending on your requirements, CTO can easily hide and unhide additional nodes with a shortcut key.



```
loc_7FF9647C1054:        ; lppe
lea     rdx, [rsp+278h+pe]
mov     rcx, rax         ; hSnapshot
call    cs:Process32FirstW
test    eax, eax
jz      short loc_7FF9647C109D

db      66h, 66h
nop     word ptr [rax+rax+00000000h]

loc_7FF9647C1070:
lea     rdx, String2     ; "lsass.exe"
lea     rcx, [rsp+278h+pe.szExeFile] ; String1
call    _wcsicmp
test    eax, eax
jz      short loc_7FF9647C1099

lea     rdx, [rsp+278h+pe] ;
mov     rcx, rbx         ; hS
call    cs:Process32NextW
test    eax, eax
jnz     short loc_7FF9647C10

jmp     short loc_7FF9647C109D
```

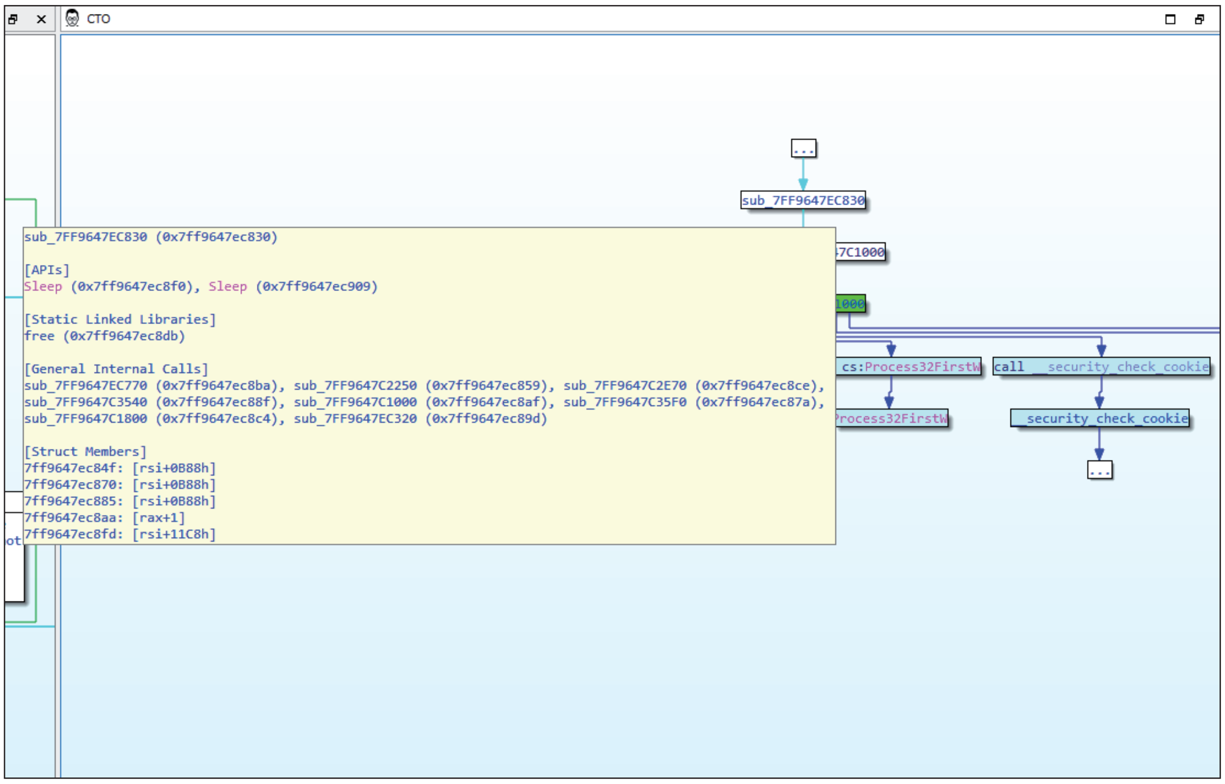
```
Strings enabled
```

Figure 5: A string as a node.

In Figure 5, you can see CTO shows us the string ‘lsass.exe’ as a node. If you click the node on CTO, *IDA View* will move to the address where the string is referred. Then you can see the disassembled code around the address. The function clearly indicates that it finds the ‘lsass.exe’ process in the process list.

2.2 Function summary

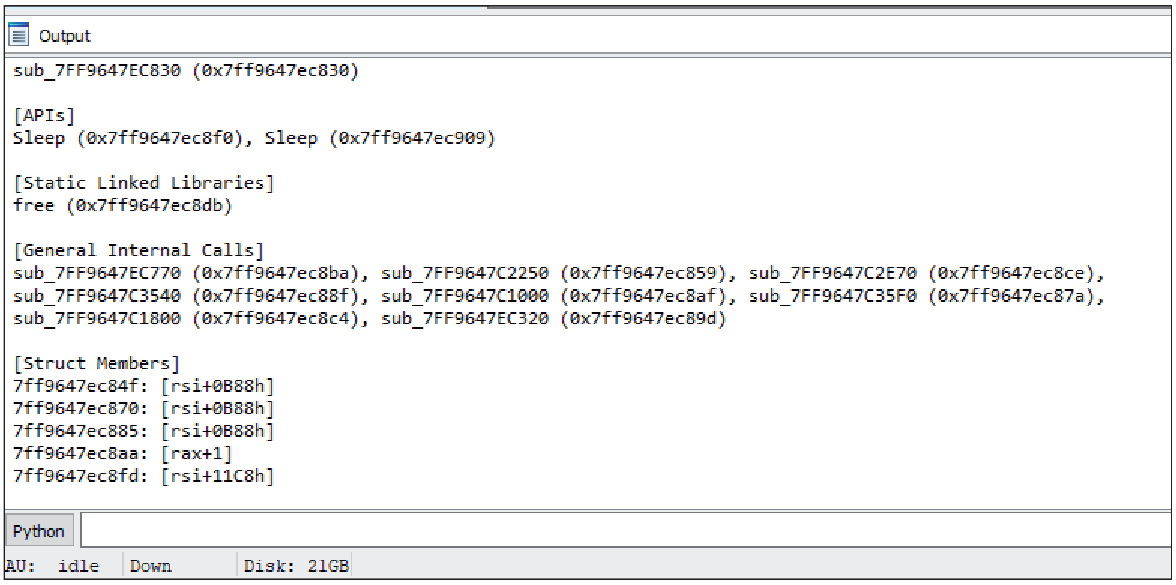
Even if you do not display additional nodes such as strings and global variables, you can still access a function summary as a node hint by pointing the mouse cursor on a function node.



```
sub_7FF9647EC830 (0x7ff9647ec830)

[APIs]
Sleep (0x7ff9647ec8f0), Sleep (0x7ff9647ec909)

[Static Linked Libraries]
free (0x7ff9647ec8db)

[General Internal Calls]
sub_7FF9647EC770 (0x7ff9647ec8ba), sub_7FF9647C2250 (0x7ff9647ec859), sub_7FF9647C2E70 (0x7ff9647ec8ce),
sub_7FF9647C3540 (0x7ff9647ec88f), sub_7FF9647C1000 (0x7ff9647ec8af), sub_7FF9647C35F0 (0x7ff9647ec87a),
sub_7FF9647C1800 (0x7ff9647ec8c4), sub_7FF9647EC320 (0x7ff9647ec89d)

[Struct Members]
7ff9647ec84f: [rsi+0B88h]
7ff9647ec870: [rsi+0B88h]
7ff9647ec885: [rsi+0B88h]
7ff9647ec8aa: [rax+1]
7ff9647ec8fd: [rsi+11C8h]
```

Figure 6: Node hint.

However, a node hint is not clickable. You cannot directly access the information on it. Therefore, I implemented a 'print hint' feature as well, which dumps the information to the output window. Then you can access the information by double-clicking a function name or an address on it.



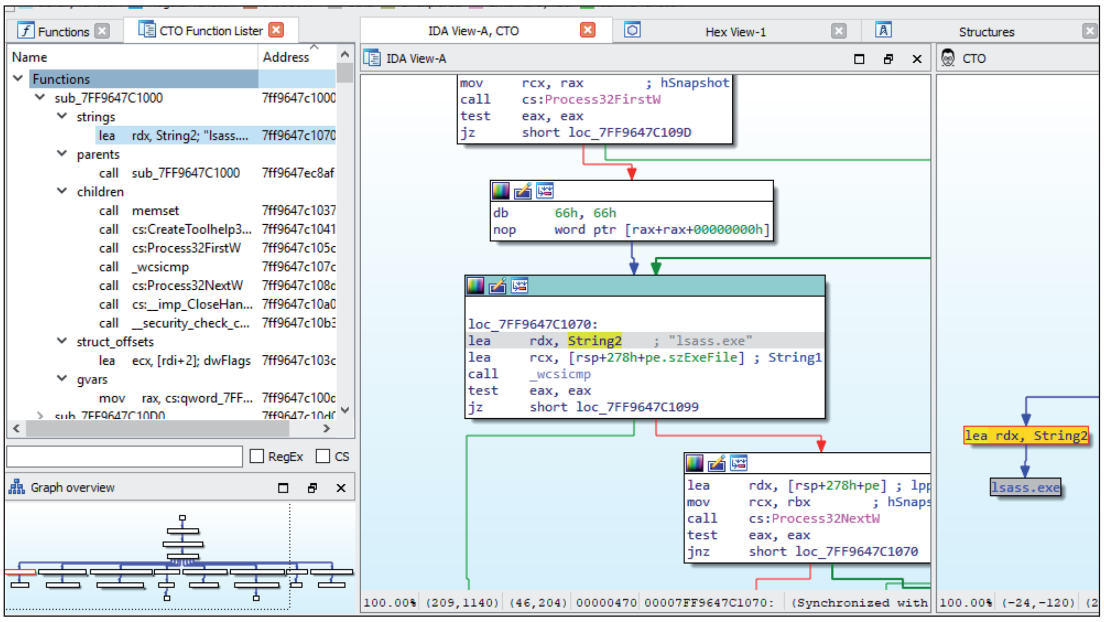
```
sub_7FF9647EC830 (0x7ff9647ec830)

[APIs]
Sleep (0x7ff9647ec8f0), Sleep (0x7ff9647ec909)

[Static Linked Libraries]
free (0x7ff9647ec8db)

[General Internal Calls]
sub_7FF9647EC770 (0x7ff9647ec8ba), sub_7FF9647C2250 (0x7ff9647ec859), sub_7FF9647C2E70 (0x7ff9647ec8ce),
sub_7FF9647C3540 (0x7ff9647ec88f), sub_7FF9647C1000 (0x7ff9647ec8af), sub_7FF9647C35F0 (0x7ff9647ec87a),
sub_7FF9647C1800 (0x7ff9647ec8c4), sub_7FF9647EC320 (0x7ff9647ec89d)

[Struct Members]
7ff9647ec84f: [rsi+0B88h]
7ff9647ec870: [rsi+0B88h]
7ff9647ec885: [rsi+0B88h]
7ff9647ec8aa: [rax+1]
7ff9647ec8fd: [rsi+11C8h]
```

```
AU:  idle   Down     Disk: 21GB
```

Figure 7: Dumped function summary.

2.3 CTO function lister

You can also use a helper list widget called 'CTO function lister' to access a function summary. The widget can be used as a standalone tool as well, but it is designed to help CTO. You can access the same information as a node hint. You can also use a regex filter on it to easily find information that you want to know, like which function calls CreateProcessW or which function accesses a structure offset.



```
Name                                   Address
Functions
  sub_7FF9647C1000                     7ff9647c1000
    strings
      lea   rdx, String2; "lsass....   7ff9647c1070
    parents
      call  sub_7FF9647C1000           7ff9647ec8af
    children
      call  memset                     7ff9647c1037
      call  cs:CreateToolhelp3...      7ff9647c1041
      call  cs:Process32FirstW         7ff9647c105c
      call  _wcsicmp                   7ff9647c107c
      call  cs:Process32NextW          7ff9647c108c
      call  cs:__imp_CloseHan...       7ff9647c10a0
      call  __security_check_c...      7ff9647c10b3
    struct_offsets
      lea   ecx, [rdi+2]; dwFlags      7ff9647c103c
    gvars
      mov   rax, cs:qword_7FF...       7ff9647c100c
  sub_7FF9647C10D0                     7ff9647c10d0
```

```
mov     rcx, rax          ; hSnapshot
call    cs:Process32FirstW
test    eax, eax
jz      short loc_7FF9647C109D

db      66h, 66h
nop     word ptr [rax+rax+00000000h]

loc_7FF9647C1070:
lea     rdx, String2     ; "lsass.exe"
lea     rcx, [rsp+278h+pe.szExeFile] ; String1
call    _wcsicmp
test    eax, eax
jz      short loc_7FF9647C1099

lea     rdx, [rsp+278h+pe] ; lpp
mov     rcx, rbx          ; hSnaps
call    cs:Process32NextW
test    eax, eax
jnz     short loc_7FF9647C1070
```

```
100.00% (209,1140) (46,204) 00000470 00007FF9647C1070: (Synchronized with
100.00% (-24,-120) (2
```

Figure 8: CTO function lister.

2.4 Keep the graph simple

As I mentioned earlier, Proximity Browser and the Chart feature on *IDA* show us unnecessary xrefs such as unwind information and static linked libraries. They increase the number of nodes and make it difficult to understand the relationships due to the complexity.

In order to keep the graph simple, CTO does not show unnecessary xrefs, although you can check all xrefs with a shortcut key.

CTO also stops tracing if it finds a function related to a static linked library, although you can still dig into it manually by expanding a collapsed node.

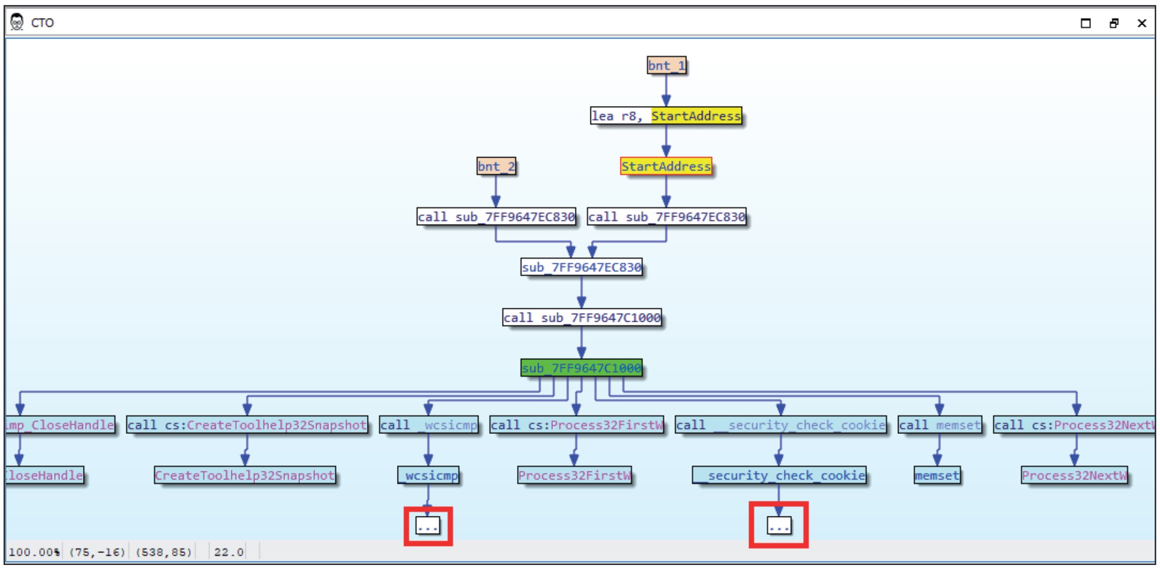


Figure 9: Exceeded nodes.

You can see two '...' nodes in Figure 9. They are called the path exceeded nodes. You can double-click them to expand if you want to dig into them.

In addition, CTO omits many parent nodes not related to the paths to the target address, although you can easily include all the omitted nodes with a shortcut key. If you compare Figure 10 with Figure 9, you can see many functions under the parent nodes. CTO omits them by default because if you want to check only paths related to the target function, they are unnecessary. However, if you want to include them, you can easily display the nodes with a shortcut key.

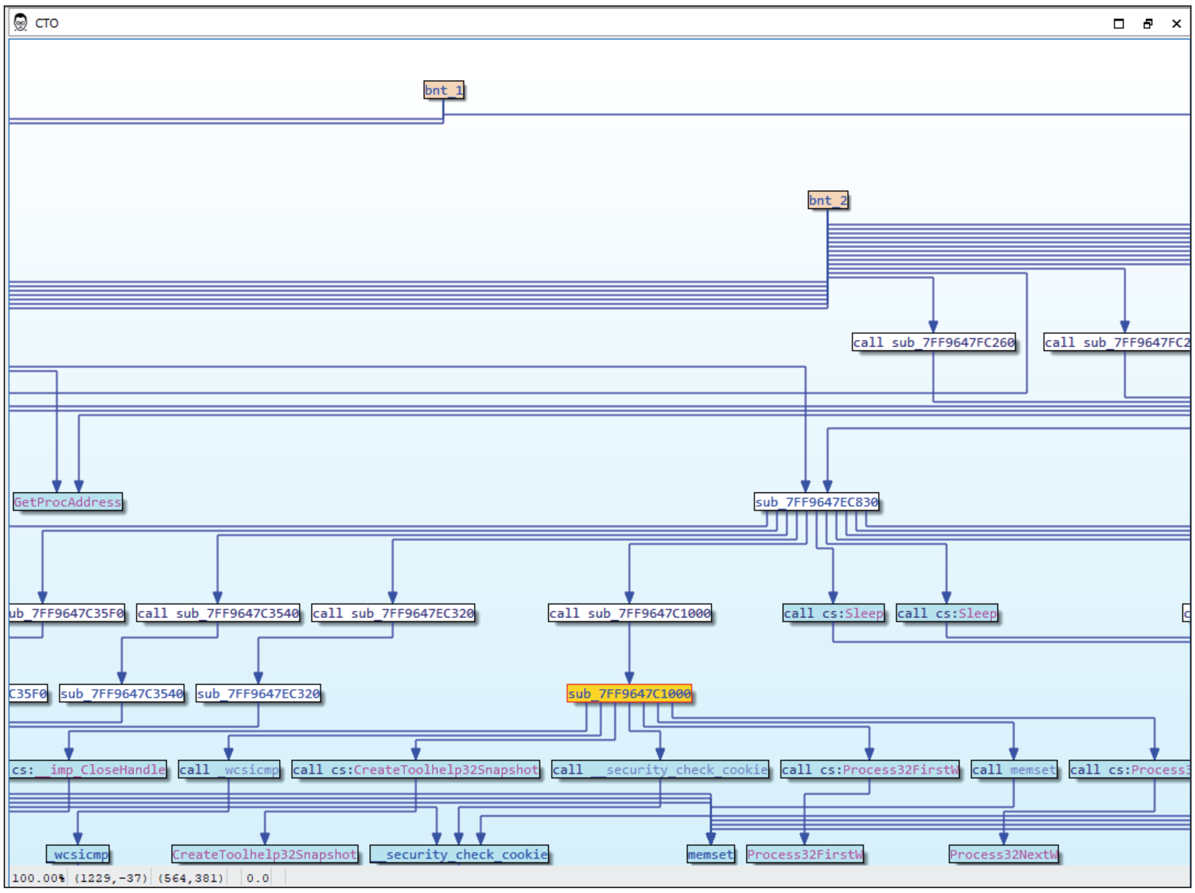


Figure 10: Showing child nodes in parent functions.

CTO has several options such as 'Go to start node' and 'Jump to a node' in order to avoid to getting lost in a graph with many nodes. The former can immediately go back to the start node that you chose when you created the CTO graph. The latter can find a node from a list using a keyword such as a partial function name, a partial address and a node type, and so on, and jump to the node.

You can also get the opposite side of a node by double-clicking the edge, which is an arrow shape, like in *IDA View*. And you can get next-to-node information by placing the cursor on an arrow or a node since the hint will also be displayed.

2.5 Synchronizing with IDA View

To be a field-oriented tool, I think usability is important. Let me explain.

For usability, CTO will be docked next to *IDA* disassembly view (or *IDA View*) by default. You can check code around the address in *IDA View* as soon as you click a node corresponding to the address on CTO and vice versa, since CTO and *IDA* will be synchronized with each other.

2.6 Callee-Caller mode

As you have seen in CTO's graphs so far, you can check each caller address in a function as well as the callee functions. Caller addresses help you rapidly understand what role their parent function has. Of course, you can omit caller nodes if they are unnecessary, as in Figure 11. Then the graph will be simpler. You can toggle the modes with a shortcut key.

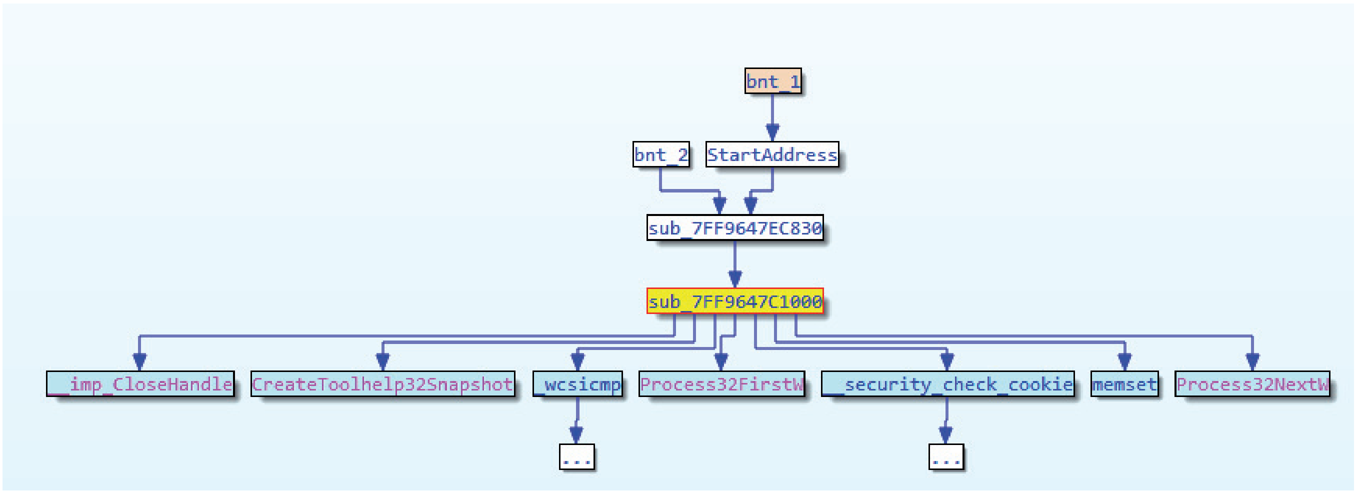


Figure 11: Pure function call graph.

2.7 Shortcuts redirecting to IDA

You can access several *IDA* shortcuts, which are often likely to be used while you are focusing CTO's window, for example, renaming a function name, checking xrefs, applying a structure to an offset and so on, by redirecting the key input to *IDA*. You do not have to click on the *IDA* screen every time you want to use them.

2.8 Function call paths graph

Although *IDA*'s Proximity Browser can create a function call paths graph, it cannot create a graph based on an API pointer on the import table like CreateFileA or CreateProcessW. Since it is very important for reverse engineers to check important APIs, making such graphs was one of my motivations to create the new tool.

CTO creates a call paths graph as a subgraph in a new tab so that you can keep the current graph you are looking into while you are checking other paths or nodes. Figure 12 shows a graph based on CreateFileW.

2.9 Third-party tool corroboration

CTO collects specific comments. If a tool outputs its result as a comment, CTO can recognize it. Currently, it collects the outputs of the tools below:

- findcrypt.py [2]
- ironstrings [3]

I will extend this in the future.

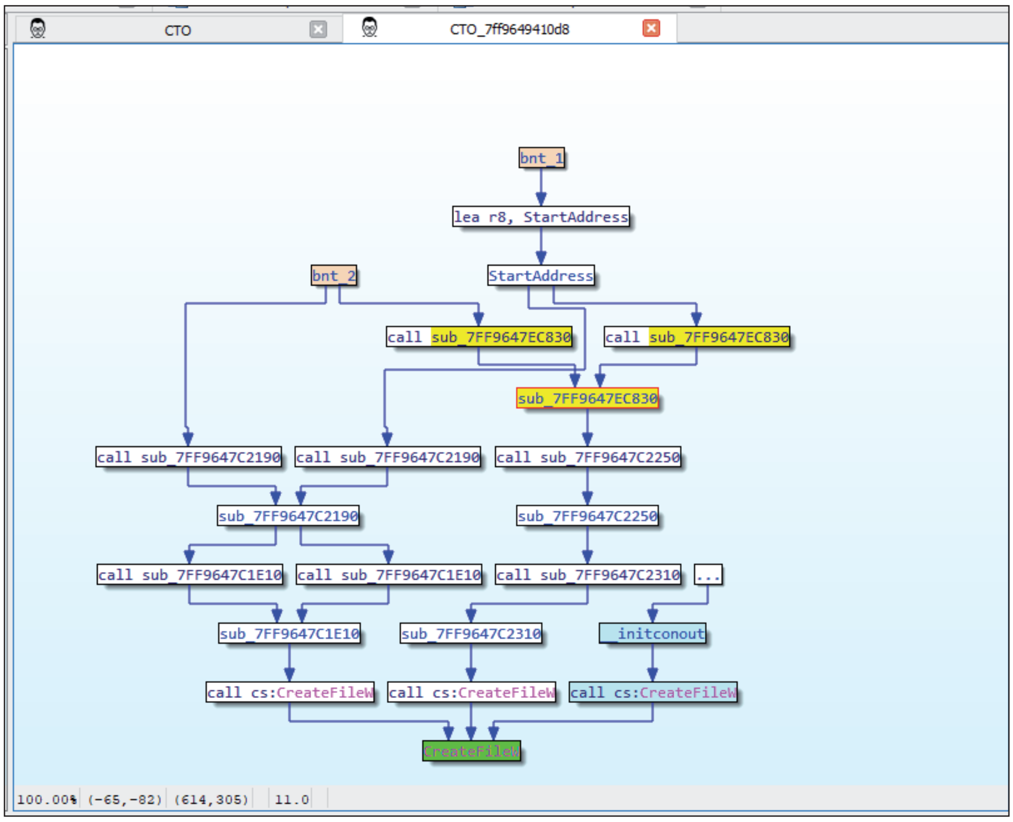
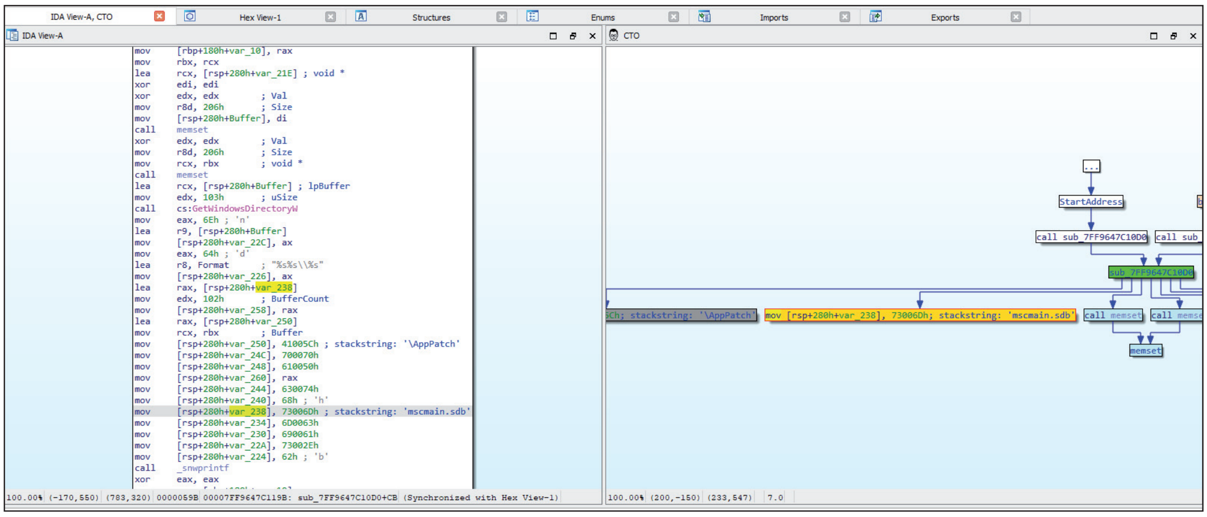


Figure 12: Find paths to CreateFileW.

In Figure 13 you can see that CTO recognizes a stack string identified by ironstrings as a node.



```
mov     [rbp+180h+var_10], rax
mov     rbx, rcx
lea     rcx, [rsp+280h+var_21E] ; void *
xor     edi, edi
xor     edx, edx        ; Val
mov     r8d, 206h       ; Size
mov     [rsp+280h+Buffer], di
call    memset
xor     edx, edx        ; Val
mov     r8d, 206h       ; Size
mov     rcx, rbx        ; void *
call    memset
lea     rcx, [rsp+280h+Buffer] ; lpBuffer
mov     edx, 103h       ; uSize
call    cs:GetWindowsDirectoryW
mov     eax, 6Eh ; 'n'
lea     r9, [rsp+280h+Buffer]
mov     [rsp+280h+var_22C], ax
mov     eax, 64h ; 'd'
lea     r8, Format      ; "%s%s\\%s"
mov     [rsp+280h+var_226], ax
lea     rax, [rsp+280h+var_238]
mov     edx, 102h       ; BufferCount
mov     [rsp+280h+var_258], rax
lea     rax, [rsp+280h+var_250]
mov     rcx, rbx        ; Buffer
mov     [rsp+280h+var_250], 41005Ch ; stackstring: '\AppPatch'
mov     [rsp+280h+var_24C], 700070h
mov     [rsp+280h+var_248], 610050h
mov     [rsp+280h+var_260], rax
mov     [rsp+280h+var_244], 630074h
mov     [rsp+280h+var_240], 68h ; 'h'
mov     [rsp+280h+var_238], 73006Dh ; stackstring: 'mscmain.sdb'
mov     [rsp+280h+var_234], 6D0063h
mov     [rsp+280h+var_230], 690061h
mov     [rsp+280h+var_22A], 73002Eh
mov     [rsp+280h+var_224], 62h ; 'b'
call    _snwprintf
xor     eax, eax
```

```
100.00% (-170,550) (783,320) 0000059B 00007FF9647C119B: sub_7FF9647C10D0+CB (Synchronized with Hex View-1)
```

Figure 13: Tool corroboration.

3. HOW CTO WORKS, INSIDE CTO

3.1 Core structures of CTO

In order to synchronize with *IDA View*, CTO utilizes two hooks:

- UI hooks (UI_Hooks class)
- View hooks (View_Hooks class)

UI hooks and view hooks are in ida_kernwin.py.

CTO also inherits the graph viewer class (GraphViewer class) in ida_graph.py.

Since *IDA*'s GUI-related APIs are in these two modules, if you want to create a GUI-based *IDA* plug-in, you should look into them first.

3.2 UI_Hooks class

You can catch all UI events, which are called 'actions' by *Hex-Rays*, by inheriting this class. You can observe actions not only on your widget, but even actions on *IDA*. CTO inherits this class to update node information on CTO in real time by overriding the preprocess_action(), postprocess_action() and updating_actions() methods. CTO checks a difference between a pre- and a post-action on *IDA* and updates the information if it is needed.

For example, CTO observes the 'MakeName' action, which is called when a function or a variable name is renamed. Since there are many kinds of actions, you should check them by executing the get_registered_actions() API on ida_kernwin.py in advance. Or, you can also check by dumping the first argument of the preprocess_action() method when it is executed.

You can refer to the examples at [4] and [5], or check the CTO source code (see sync_ui.py).

3.3 View_Hooks class

CTO utilizes this class to synchronize a CTO node with the address in *IDA View*. CTO inherits the class and overrides the view_loc_changed() method to observe location change events. If the address on *IDA View* is changed, CTO colours the corresponding node on CTO. On the other hand, if a different node on CTO is selected, CTO changes the location on *IDA View* with the jumpto() API.

You can refer to an example at [6], or check the CTO source code (see sync_ui.py).

3.4. GraphViewer class

Inheriting the GraphViewer class is the simplest way to create a custom graph like CTO. You can create a graph by overriding the AddNode() method to add a box called a 'node', and the AddEdge() method to add an arrow connector called an 'edge'.

You can also hook many events on your widget. For example:

- Keyboard events
    - OnViewKeydown()
- Mouse events
    - OnClick() for click events
    - OnDblClick() for double-click events
    - OnPopup() for right-click events
    - OnHint() for on-mouse-over events for nodes
    - OnEdgeHint() for on-mouse-over events for edges

Note that you will need to implement the OnGetText() method to use this class. Although it is not defined, strictly speaking it is commented out in the super class, but it is mandatory.

You can refer to an example at [6], or check the CTO source code (see cto.py).

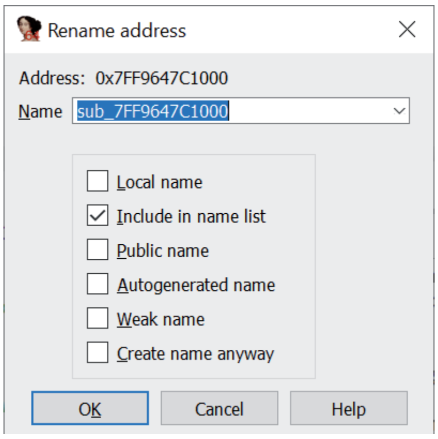
3.5 jumpto API

Anyone who has ever scripted on *IDA* should know about the jumpto() API. And anyone should know the first argument is an ea. However, how many people know the second argument and the arguments after that? And how many people know there is one more definition of the jumpto() API? They are very important for handling *IDA*'s action context. Let me explain about what the action context is and how it works.

As I mentioned earlier, CTO redirects several shortcut keys to *IDA*. For example, if you press 'N', CTO issues the 'MakeName' action to *IDA*, then *IDA* shows us a dialog for renaming. However, the dialog depends on the cursor location.

For example, let's assume you want to rename a function name manually. First, you would click the function name and press the 'N' key. Second, a 'Rename address' dialog would show up and you would be able to rename it by inputting a new function name. However, if a script or a plug-in wants to move the cursor on the function name programmatically, it will be a slightly different story.

If you use the jumpto() API with the first address of a function, you can go to the function head. And if you press 'N' after that, a 'MakeName' action will be executed and you may get the 'Rename address' dialog like the one shown in Figure 14.



Rename address

Address: 0x7FF9647C1000

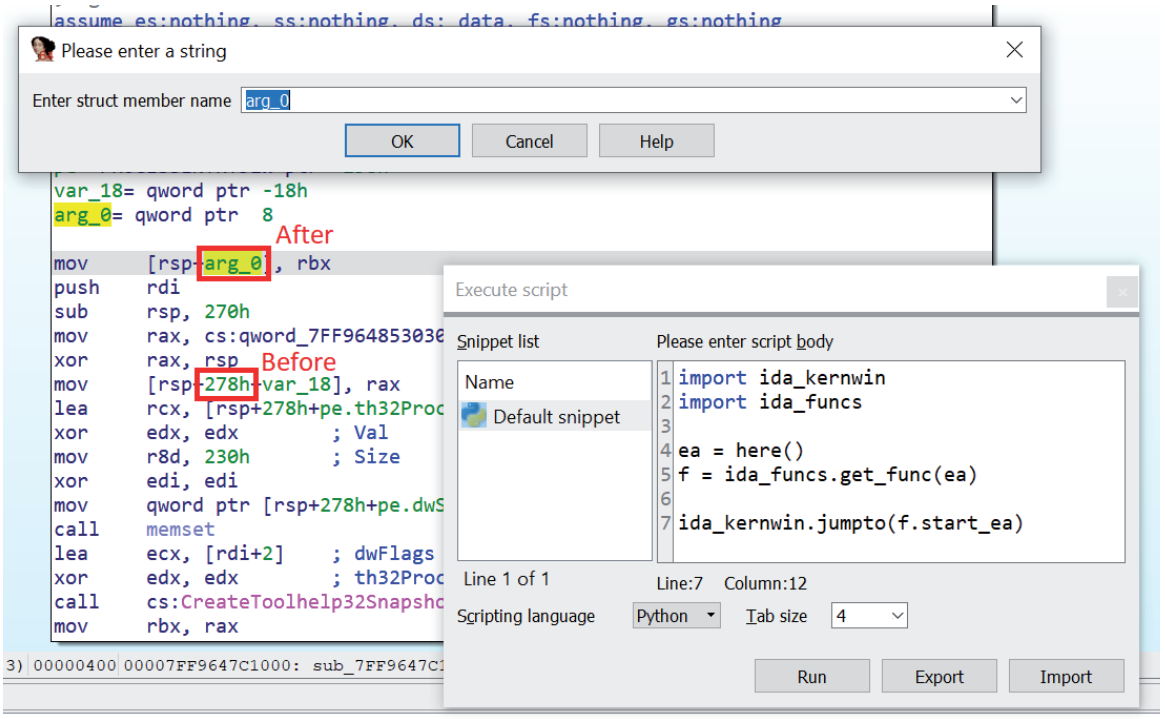
Name: sub_7FF9647C1000

- [ ] Local name
- [x] Include in name list
- [ ] Public name
- [ ] Autogenerated name
- [ ] Weak name
- [ ] Create name anyway

Figure 14: Rename address dialog.

However, if a keyboard cursor points on a renameable string, *IDA* would show us a different behaviour. For example, Figure 15 illustrates *IDA* showing us a renaming struct member dialog instead of showing a 'Rename address' dialog in spite of the cursor at the first instruction of the function. Let me explain why this occurred.

Before the script execution, the cursor was located in the middle of a line, in Figure 15, that was the offset from the stack pointer, which is an integer 0x278, on the sixth instruction of the function. After the script execution, the cursor moved to the first instruction of the function. However, the horizontal location of the cursor was not changed with the jumpto() API with only the first argument. Therefore, the cursor unexpectedly pointed to 'arg_0'. As a result, after the 'N' key was pressed, *IDA* showed a renaming struct member dialog, not the 'Rename address' dialog.



Please enter a string

Enter struct member name: arg_0

```
assume es:nothing, ss:nothing, ds: data, fs:nothing, gs:nothing
var_18= qword ptr -18h
arg_0= qword ptr  8

mov     [rsp+arg_0], rbx
push    rdi
sub     rsp, 270h
mov     rax, cs:qword_7FF964853030
xor     rax, rsp
mov     [rsp+278h+var_18], rax
lea     rcx, [rsp+278h+pe.th32Proc
xor     edx, edx            ; Val
mov     r8d, 230h           ; Size
xor     edi, edi
mov     qword ptr [rsp+278h+pe.dwS
call    memset
lea     ecx, [rdi+2]        ; dwFlags
xor     edx, edx            ; th32Proc
call    cs:CreateToolhelp32Snapsho
mov     rbx, rax
3) 00000400 00007FF9647C1000: sub_7FF9647C1
```

Execute script

Snippet list: Default snippet

Please enter script body

```
import ida_kernwin
import ida_funcs

ea = here()
f = ida_funcs.get_func(ea)

ida_kernwin.jumpto(f.start_ea)
```

Line 1 of 1 Line:7 Column:12

Scripting language: Python Tab size: 4

Figure 15: IDA changes the behaviour depending on the cursor location.

Thus, *IDA* will change its behaviour depending on the cursor position, even though the same action is issued. This is the action context. You will need to pay attention to it. And the two variants of jumpto() APIs handle this.

According to ida_kernwin.py, there are two definitions of jumpto() APIs:

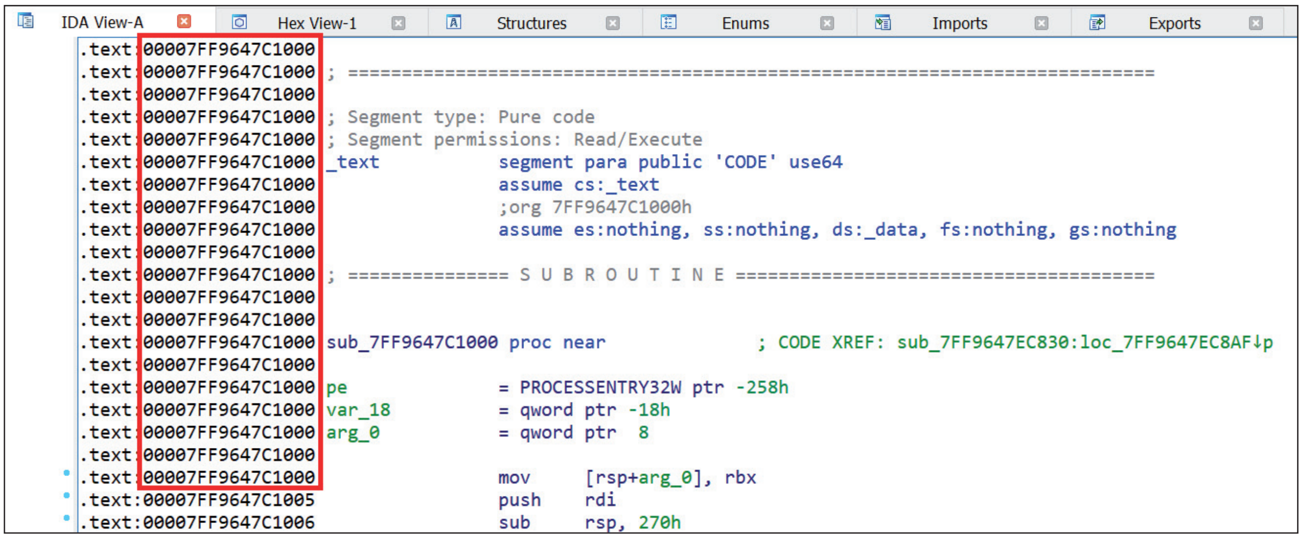
jumpto(ea, opnum=-1, uijmp_flags=0x0001) -> bool

jumpto(custom_viewer, place, x, y) -> bool

The former is the one we usually use. But there is one more important argument. That is the second argument 'opnum', which is an operand number. If you specify an opnum, you can easily tweak the horizontal location of the cursor. This is useful if you want to move the cursor onto an operand.

If you want to move the cursor vertically and/or horizontally, you can use the second definition of the API.

For example, a head of the address in a function usually has multiple lines in the same address, as shown in Figure 16.



```
.text:00007FF9647C1000
.text:00007FF9647C1000 ; ===========================================================================
.text:00007FF9647C1000
.text:00007FF9647C1000 ; Segment type: Pure code
.text:00007FF9647C1000 ; Segment permissions: Read/Execute
.text:00007FF9647C1000 _text           segment para public 'CODE' use64
.text:00007FF9647C1000                 assume cs:_text
.text:00007FF9647C1000                 ;org 7FF9647C1000h
.text:00007FF9647C1000                 assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
.text:00007FF9647C1000
.text:00007FF9647C1000 ; =============== S U B R O U T I N E =======================================
.text:00007FF9647C1000
.text:00007FF9647C1000
.text:00007FF9647C1000 sub_7FF9647C1000 proc near              ; CODE XREF: sub_7FF9647EC830:loc_7FF9647EC8AF↓p
.text:00007FF9647C1000
.text:00007FF9647C1000 pe              = PROCESSENTRY32W ptr -258h
.text:00007FF9647C1000 var_18          = qword ptr -18h
.text:00007FF9647C1000 arg_0           = qword ptr  8
.text:00007FF9647C1000
.text:00007FF9647C1000                 mov     [rsp+arg_0], rbx
.text:00007FF9647C1005                 push    rdi
.text:00007FF9647C1006                 sub     rsp, 270h
```

Figure 16: An address could have multiple lines.

In this case, the first definition of the jumpto() API cannot handle it. On the other hand, the second one can do it. The second argument of it is an instance of the place_t object, which can be acquired by calling the get_custom_viewer_place() API. The API also returns x and y of the current cursor position. And place_t also has a member named 'lineno', which can adjust a line number in the same address. You can tweak them and pass it to the second definition of the jumpto() API. Then you can move wherever you want.

You can also use the custom_viewer_jump() API to move vertically and/or horizontally. You can find an example of how to use it at [7].

4. TIPS

4.1 Reloading your libraries frequently while developing

If you develop an IDAPython script to be imported by other scripts, and if you modify it, it will not be re-imported by a script once it is executed unless you reopen the idb. That is not efficient. In that case, use the require() API on ida_idaapi.py. Then it will get re-imported from a script every time it is executed.

4.2 How to find the cause of a crash on IDAPython

When developing plug-ins running on IDAPython, I frequently find that *IDA* itself crashes.

To identify the cause of the crash is very difficult because *IDA* will be terminated before any debug print messages on the Python side are output. In such cases, you can use the PyExt tool [8].

Figure 17 shows that PyExt found a Python thread on IDA, and the cause of the crash was the viewer_get_gli() API in ida_graph.py with the crash dump. You can also check its arguments when being crashed by clicking '[Frame]'.

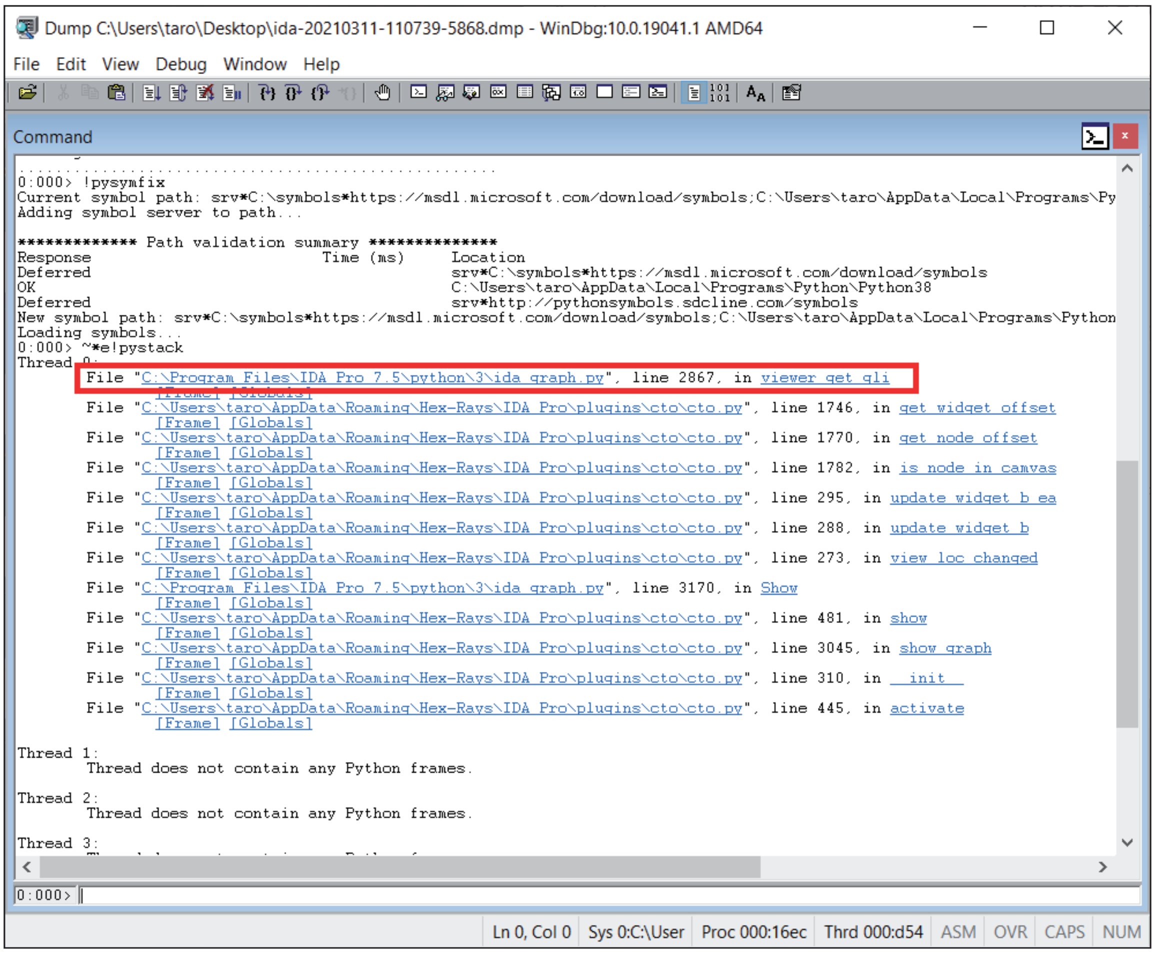
5. FUTURE WORK

The following are my objectives for future work on CTO:

- More tool corroboration
- Collecting instructions to be observed
- Implementing a more efficient way to collect paths
- Enhancing the speed
- Improving stability.

6. CONCLUSION

IDA's scripting and plug-in features are very powerful and they help us enhance our speed of analysis. I hope CTO helps your analysis and I also hope you will be inspired to develop new scripts or plug-ins in the feature.


Dump C:\Users\taro\Desktop\ida-20210311-110739-5868.dmp - WinDbg:10.0.19041.1 AMD64
File Edit View Debug Window Help
Command

```
.................................................
0:000> !pysymfix
Current symbol path: srv*C:\symbols*https://msdl.microsoft.com/download/symbols;C:\Users\taro\AppData\Local\Programs\Py
Adding symbol server to path...

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*C:\symbols*https://msdl.microsoft.com/download/symbols
OK                                             C:\Users\taro\AppData\Local\Programs\Python\Python38
Deferred                                       srv*http://pythonsymbols.sdcline.com/symbols
New symbol path: srv*C:\symbols*https://msdl.microsoft.com/download/symbols;C:\Users\taro\AppData\Local\Programs\Python
Loading symbols...
0:000> ~*e!pystack
Thread 0:
       File "C:\Program Files\IDA Pro 7.5\python\3\ida_graph.py", line 2867, in viewer_get_gli
            [Frame] [Globals]
       File "C:\Users\taro\AppData\Roaming\Hex-Rays\IDA Pro\plugins\cto\cto.py", line 1746, in get_widget_offset
            [Frame] [Globals]
       File "C:\Users\taro\AppData\Roaming\Hex-Rays\IDA Pro\plugins\cto\cto.py", line 1770, in get_node_offset
            [Frame] [Globals]
       File "C:\Users\taro\AppData\Roaming\Hex-Rays\IDA Pro\plugins\cto\cto.py", line 1782, in is_node_in_canvas
            [Frame] [Globals]
       File "C:\Users\taro\AppData\Roaming\Hex-Rays\IDA Pro\plugins\cto\cto.py", line 295, in update_widget_b_ea
            [Frame] [Globals]
       File "C:\Users\taro\AppData\Roaming\Hex-Rays\IDA Pro\plugins\cto\cto.py", line 288, in update_widget_b
            [Frame] [Globals]
       File "C:\Users\taro\AppData\Roaming\Hex-Rays\IDA Pro\plugins\cto\cto.py", line 273, in view_loc_changed
            [Frame] [Globals]
       File "C:\Program Files\IDA Pro 7.5\python\3\ida_graph.py", line 3170, in Show
            [Frame] [Globals]
       File "C:\Users\taro\AppData\Roaming\Hex-Rays\IDA Pro\plugins\cto\cto.py", line 481, in show
            [Frame] [Globals]
       File "C:\Users\taro\AppData\Roaming\Hex-Rays\IDA Pro\plugins\cto\cto.py", line 3045, in show_graph
            [Frame] [Globals]
       File "C:\Users\taro\AppData\Roaming\Hex-Rays\IDA Pro\plugins\cto\cto.py", line 310, in __init__
            [Frame] [Globals]
       File "C:\Users\taro\AppData\Roaming\Hex-Rays\IDA Pro\plugins\cto\cto.py", line 445, in activate
            [Frame] [Globals]

Thread 1:
       Thread does not contain any Python frames.

Thread 2:
       Thread does not contain any Python frames.

Thread 3:
0:000>
```

Figure 17: PyExt with IDA's crash dump.

REFERENCES

[1] https://www.hex-rays.com/products/ida/support/freefiles/qwingraph_src.zip.

[2] https://github.com/you0708/ida/tree/master/idapython_tools/findcrypt.

[3] https://github.com/fireeye/flare-ida/tree/master/python/flare/ironstrings.

[4] https://github.com/idapython/src/blob/master/examples/uihooks/log_misc_events.py.

[5] https://github.com/idapython/src/blob/master/examples/uihooks/prevent_jump.py.

[6] https://github.com/idapython/src/blob/master/examples/widgets/graphs/custom_graph_with_actions.py.

[7] https://github.com/idapython/src/blob/master/examples/widgets/listings/jump_next_comment.py.

[8] https://github.com/SeanCline/PyExt/releases.