Reversing Stories: Updating the Undocumented ESTROBJ and STROBJ Structures for Windows 10 x64

Overview

From time to time the VS-Labs, VerSprite's Cybersecurity Research and Development division, encounters scenarios whereby undocumented Windows functions or structures need to be reverse engineered in order to perform vulnerability analysis.

One example of this that was recently encountered was when the team decided to reverse a recently patched N-Day vulnerability and determined that the undocumented function ESTROBJ::ptlBaseLineAdjustSet(), which was introduced with Windows 8.1, needed to be reversed in order to fully understand the vulnerability. What made this task more difficult than normal was the fact that two undocumented structures, ESTROBJ and STROBJ had been updated in Windows 10. As a result, HexRays Decompiler was failing to output valid pseudocode anymore.

So how could one solve this scenario? In the following section, we will review the skills needed to reverse undocumented structures. By the end of the tutorial, readers will have created updated definitions of the ESTROBJ and STROBJ structures for Windows 10 x64 1903 and will have produced valid pseudocode for ESTROBJ::ptlBaseLineAdjustSet().

Obtaining Background Information

When first reversing a function, it is always good to see if there is any information available about it online. In some cases, there may be forums or websites that have already discussed the internals of a function and how it works, which may allow one to obtain the information they need without performing any additional work.

Unfortunately, if one looked online at the time this post was published, they would be unable to find any documentation on the ESTROBJ::ptlBaseLineAdjustSet() function, as was only introduced in Windows 8.1 and therefore had not been part of any source code leaks or public analysis. However, there is some hope, as if one loads the public symbols for ESTROBJ::ptlBaseLineAdjustSet() into IDA Pro, they will obtain the following function definition:

```
ESTROBJ::ptlBaseLineAdjustSet Function Prototype
void __fastcall ESTROBJ::ptlBaseLineAdjustSet(ESTROBJ *__hidden this, struct _POINTL
*)
```

This tells us that the function has two arguments. Closer inspection reveals that the second argument is a POINTL structure, whose definition can be found on MSDN and whose structure can be seen below:

```
POINTL Structure Definition
typedef struct _POINTL {
  LONG x;
  LONG y;
} POINTL, *PPOINTL;
```

From this definition, it is possible to determine that the ESTROBJ::ptlBaseLineAdjustSet() function is likely working with points on the screen as POINTL is designed to be a structure that describes the coordinates of a single point on the screen.

The other parameter, the hidden this parameter is an ESTROBJ structure. This is an internal, undocumented structure designed to handle the "*global aspects of the text positioning and text size computation*" according to the leaked NT 4.0 source code.

The most recent definition for an ESTROBJ structure is from ReactOS. Unfortunately, ReactOS is designed to be an open-source copy of Windows XP and therefore does not reflect the changes that have been added to recent versions of Windows, such as Windows 10.

Since the Windows kernel can change quite dramatically between releases, it is likely that some elements of the ESTROBJ structure have been updated since it was last documented by ReactOS. With these points in mind, here is ReactOS's definition of the ESTROBJ and STROBJ structures:

```
ReactOS's ESTROBJ and STROBJ Definitions
typedef struct _STROBJ
{
  ULONG    cGlyphs;
  FLONG    flAccel;
  ULONG    ulCharInc;
```

```
  RECTL    rclBkGround;
  GLYPHPOS *pgp;
  LPWSTR   pwszOrg;
} STROBJ;

typedef struct _ESTROBJ
{
  STROBJ    strobj;                  // 000
  ULONG     cgposCopied;             // 024
  ULONG     cgposPositionsEnumerated; // 028
  RFONTOBJ *prfo;                    // 02c
  FLONG     flTO;                    // 030
  GLYPHPOS *pgpos;                   // 034
  POINTFX  ptfxRef;                  // 038
  POINTFX  ptfxUpdate;               // 040
  POINTFX  ptfxEscapement;           // 048
  RECTFX    rcfx;                    // 050
  FIX       fxExtent;                // 060
  FIX       fxExtra;                 // 064
  FIX       fxBreakExtra;            // 068
  DWORD     dwCodePage;              // 06c
  ULONG     cExtraRects;             // 070
  RECTL     arclExtra[3];            // 074
  RECTL     rclBkGroundSave;         // 0a8
  PWCHAR    pwcPartition;            // 0b4
  PLONG     plPartition;             // 0b8
  PLONG     plNext;                  // 0bc
  GLYPHPOS *pgpNext;                 // 0c0
  LONG      lCurrentFont;            // 0c4
  POINTL    ptlBaseLineAdjust;       // 0c8
  ULONG     cTTSysGlyphs;            // 0d0
  ULONG     cSysGlyphs;              // 0d4
  ULONG     cDefGlyphs;              // 0d8
  ULONG     cNumFaceNameLinks;       // 0dc
  PULONG    pacFaceNameGlyphs;       // 0e0
  ULONG     acFaceNameGlyphs[8];     // 0e4
// size                                 104
} ESTROBJ, PESTROBJ;
```

Now that ReactOS has provided the definition of the ESTROBJ structure and the associated STROBJ structure that it encapsulates, it should be possible to decompile ESTROBJ::ptlBaseLineAdjustSet() using HexRays Decompiler to see if a clean decompilation is possible or not.

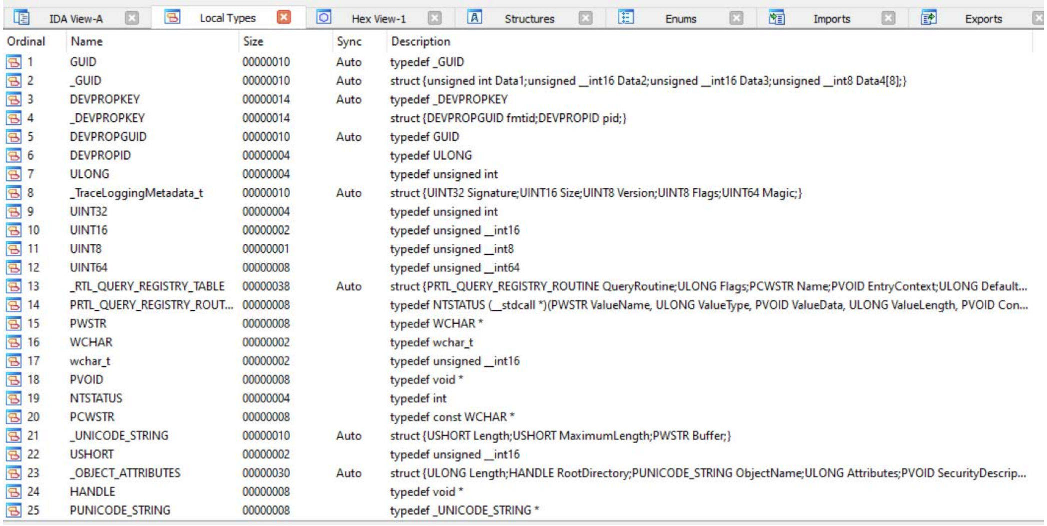
Decompiling ESTROBJ::ptlBaseLineAdjustSet() – Adding in Type Info

Before using HexRays Decompiler to decompile ESTROBJ::ptlBaseLineAdjustSet(), users need to provide the decompiler with as much type information as possible. This includes the types of each of the function's parameters, the types of each of the variables that the function uses, the return type of the function itself, and the return types and parameter types of any additional functions that are called.

This is an important part of the process because without this information, the decompiler is not able to understand what data types it is operating on at each point within the disassembly. By providing the type information, the decompiler can more accurately determine how the code should be handing certain parameters. This not only makes the pseudocode more accurate but can also allow it to apply appropriate optimizations to make it shorter and more concise, which can result in pseudocode output that is much easier to read and understand.

Thankfully, IDA Pro has already defined the function prototype from the public PDB files so there is no need to fill in parameter type information. As there is no publicly available information on the return type for ESTROBJ::ptlBaseLineAdjustSet(), it will be set to void, as this is what was defined in the PDB files.

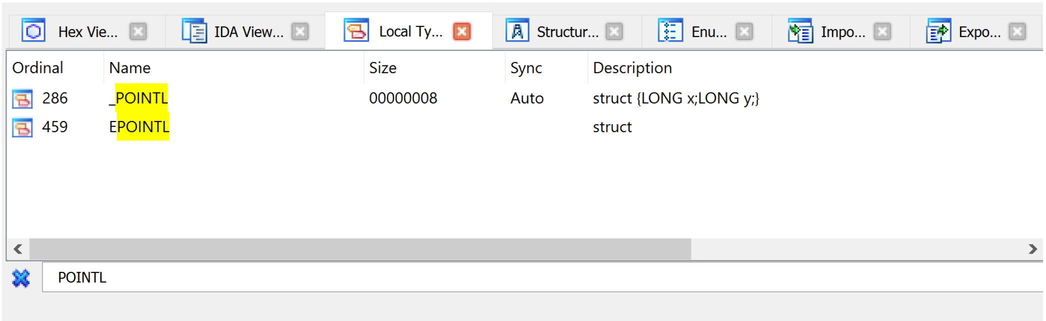
Once this is done, the next step is to verify that IDA Pro has the information needed to process the types for each of the parameters, which are of type ESTROBJ and POINTL respectively. One can verify which types are in IDA Pro's type database by pressing **SHIFT+F1** or selecting View->Open subviews->Local types. This will open the local types window, which should look similar to Figure 1.



Ordinal	Name	Size	Sync	Description
1	GUID	00000010	Auto	typedef _GUID
2	_GUID	00000010	Auto	struct {unsigned int Data1;unsigned __int16 Data2;unsigned __int16 Data3;unsigned __int8 Data4[8];}
3	DEVPROPKEY	00000014	Auto	typedef _DEVPROPKEY
4	_DEVPROPKEY	00000014		struct {DEVPROPGUID fmtid;DEVPROPID pid;}
5	DEVPROPGUID	00000010	Auto	typedef GUID
6	DEVPROPID	00000004		typedef ULONG
7	ULONG	00000004		typedef unsigned int
8	_TraceLoggingMetadata_t	00000010	Auto	struct {UINT32 Signature;UINT16 Size;UINT8 Version;UINT8 Flags;UINT64 Magic;}
9	UINT32	00000004		typedef unsigned int
10	UINT16	00000002		typedef unsigned __int16
11	UINT8	00000001		typedef unsigned __int8
12	UINT64	00000008		typedef unsigned __int64
13	_RTL_QUERY_REGISTRY_TABLE	00000038	Auto	struct {PRTL_QUERY_REGISTRY_ROUTINE QueryRoutine;ULONG Flags;PCWSTR Name;PVOID EntryContext;ULONG Default...
14	PRTL_QUERY_REGISTRY_ROUT...	00000008		typedef NTSTATUS (__stdcall *)(PWSTR ValueName, ULONG ValueType, PVOID ValueData, ULONG ValueLength, PVOID Con...
15	PWSTR	00000008		typedef WCHAR *
16	WCHAR	00000002		typedef wchar_t
17	wchar_t	00000002		typedef unsigned __int16
18	PVOID	00000008		typedef void *
19	NTSTATUS	00000004		typedef int
20	PCWSTR	00000008		typedef const WCHAR *
21	_UNICODE_STRING	00000010	Auto	struct {USHORT Length;USHORT MaximumLength;PWSTR Buffer;}
22	USHORT	00000002		typedef unsigned __int16
23	_OBJECT_ATTRIBUTES	00000030	Auto	struct {ULONG Length;HANDLE RootDirectory;PUNICODE_STRING ObjectName;ULONG Attributes;PVOID SecurityDescrip...
24	HANDLE	00000008		typedef void *
25	PUNICODE_STRING	00000008		typedef _UNICODE_STRING *

Figure 1 – Viewing the local type window in IDA Pro

Pressing **CTRL+F** will allow one to search for a specific type in IDA Pro's type database. Type POINTL to see if the POINTL structure has already been defined in IDA Pro's local types. IDA Pro should indicate that it already has a definition for the POINTL structure, as can be seen in Figure 2. Notice that this definition is the same one that was shown on MSDN, so no alterations are needed.



Ordinal	Name	Size	Sync	Description
286	_POINTL	00000008	Auto	struct {LONG x;LONG y;}
459	EPOINTL			struct

Figure 2 – Searching for a specific type in IDA Pro's type database

The next structure that needs to be defined is ESTROBJ. However, before one can define this structure, one needs to define the STROBJ structure. This is needed as the first field within the ESTROBJ structure is a STROBJ structure named strobj. As the STROBJ structure is not yet defined in IDA Pro, it must be defined in IDA as a local type before the ESTROBJ structure can be defined.

Looking again, one can see that STROBJ in turn depends on the RECTL and GLYPHPOS structures. RECTL is defined by default in IDA Pro 7.4, however GLYPHPOS is not, so one needs to add it to IDA Pro's local type database. Adding a local type in IDA Pro can be achieved by clicking **SHIFT+F1** to open the local types window and then pressing the **INSERT** key. This will display a new screen where one can define their own structure using C code.

A second look at the GLYPHPOS structure shows that it requires the GLYPHDEF structure to be defined. GLYPHDEF in turn will require the GLYPHBITS and PATHOBJ structures to be defined. Thankfully, all these structures are documented on MSDN. The following snippet shows their definitions:

```
GLYPHBITS, PATHOBJ, GLYPHDEF, and GLYPHPOS Structure Definitions
typedef struct _GLYPHBITS {
  POINTL ptlOrigin;
  SIZEL  sizlBitmap;
  BYTE   aj[1];
} GLYPHBITS;

typedef struct _PATHOBJ {
  FLONG fl;
  ULONG cCurves;
} PATHOBJ;

typedef union _GLYPHDEF {
  GLYPHBITS *pgb;
  PATHOBJ   *ppo;
} GLYPHDEF;

typedef struct _GLYPHPOS {
```

```
    HANDLE    hg;
    GLYPHDEF *pgdf;
    POINTL    ptl;
} GLYPHPOS, *PGLYPHPOS;
```

To speed up the process of defining all these structures, one can use a nifty feature of IDA Pro that allows multiple structures to be defined in the types window at once. This can be achieved by copying and pasting the list above directly into IDA Pro's type window and then hitting the **OK** button. Figure 3 below shows what the type window should look like prior to hitting the **OK** button.

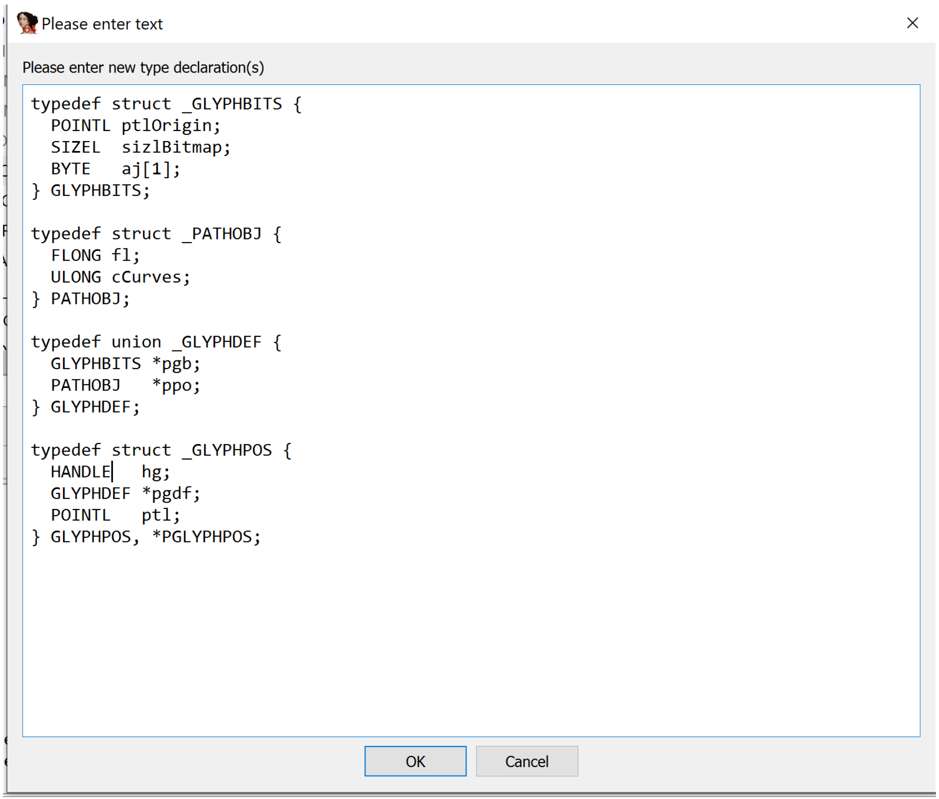


Figure 3 – Declaring multiple structure definitions in IDA Pro's type editor

Next, add in the ReactOS definition of the STROBJ field. Figure 4 shows what the type window should look like prior to hitting the **OK** button.

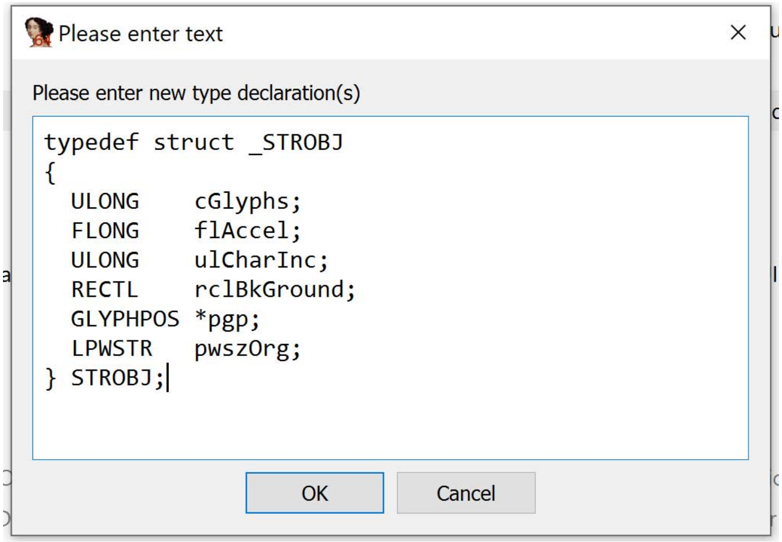


Figure 4 – Defining the STROBJ structure in IDA Pro's type editor

If everything went well, no error messages should appear. If any error messages appear, it is possible that RECTL has not been defined in IDA Pro's type database (newer versions of IDA Pro have more updated type definition databases). In this case, one should define RECTL using the RECTL definition from MSDN. After successfully updating the IDA Pro type database with the definition for RECTL, attempt to add the STROBJ definition again. This should result in STROBJ being added without any error messages being shown.

Once STROBJ has been successfully added to the local types, it is time to repeat the same process for ReactOS's definition of ESTROBJ. To do this, one must provide definitions for the POINTFX, RECTFX and FIX structures. The definitions of these structures can all be found on the GDI Data Types page on MSDN. Copy and paste the following lines into a new IDA Pro types window to define the POINTFX, RECTFX, and FIX structures in the correct order. Once this is done, press the **OK** button to save the types into IDA Pro's database.

```
RECTFX and POINTFX Structure Definitions
typedef int FIX;
typedef struct _RECTFX {
  FIX xLeft;
  FIX yTop;
  FIX xRight;
  FIX yBottom;
} RECTFX;
typedef struct _POINTFX {
  FIX x;
  FIX y;
} POINTFX;
```

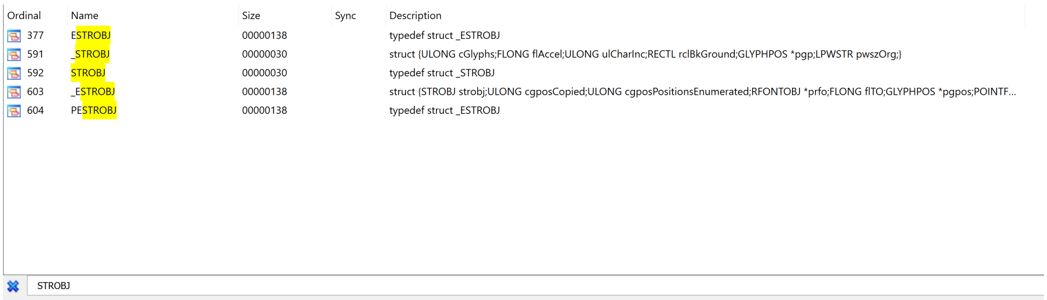
Once this is done, it should be possible to create the local structure definition for ESTROBJ by using the following definition and repeating the same type definition process as before.

```
ESTROBJ Structure Definition from ReactOS
typedef struct _ESTROBJ
{
  STROBJ    strobj;                    // 000
  ULONG     cgposCopied;               // 024
  ULONG     cgposPositionsEnumerated;  // 028
  RFONTOBJ *prfo;                      // 02c
  FLONG     flTO;                      // 030
  GLYPHPOS *pgpos;                     // 034
  POINTFX  ptfxRef;                    // 038
  POINTFX  ptfxUpdate;                 // 040
  POINTFX  ptfxEscapement;             // 048
  RECTFX    rcfx;                      // 050
  FIX       fxExtent;                  // 060
  FIX       fxExtra;                   // 064
  FIX       fxBreakExtra;              // 068
  DWORD     dwCodePage;                // 06c
  ULONG     cExtraRects;               // 070
  RECTL     arclExtra[3];              // 074
  RECTL     rclBkGroundSave;           // 0a8
  PWCHAR    pwcPartition;              // 0b4
  PLONG     plPartition;               // 0b8
  PLONG     plNext;                    // 0bc
  GLYPHPOS *pgpNext;                   // 0c0
  LONG      lCurrentFont;              // 0c4
  POINTL    ptlBaseLineAdjust;         // 0c8
  ULONG     cTTSysGlyphs;              // 0d0
  ULONG     cSysGlyphs;                // 0d4
  ULONG     cDefGlyphs;                // 0d8
  ULONG     cNumFaceNameLinks;         // 0dc
  PULONG    pacFaceNameGlyphs;         // 0e0
  ULONG     acFaceNameGlyphs[8];       // 0e4
                                       // Total size 104

} ESTROBJ, PESTROBJ;
```

After this definition is added, enter the following two lines into a new local type definition to ensure that there are appropriate links between the backend structures that were just created (_ESTROBJ and _STROBJ) and the common symbol names (ESTROBJ and STROBJ).

```
Creating Links Between Backend Structures and Commonly Used Names
typedef struct _ESTROBJ ESTROBJ;
typedef struct _STROBJ STROBJ;
```

Finally, to confirm all the previous steps completed successfully, open the local types window by pressing **SHIFT+F1** and search for STROBJ. The results should be the same as the ones shown in Figure 5.



Ordinal	Name	Size	Sync	Description
377	ESTROBJ	00000138		typedef struct _ESTROBJ
591	_STROBJ	00000030		struct {ULONG cGlyphs;FLONG flAccel;ULONG ulCharInc;RECTL rclBkGround;GLYPHPOS *pgp;LPWSTR pwszOrg;}
592	STROBJ	00000030		typedef struct _STROBJ
603	_ESTROBJ	00000138		struct {STROBJ strobj;ULONG cgposCopied;ULONG cgposPositionsEnumerated;RFONTOBJ *prfo;FLONG flTO;GLYPHPOS *pgpos;POINTF...
604	PESTROBJ	00000138		typedef struct _ESTROBJ

STROBJ

Figure 5 – Type window after STROBJ and ESTROBJ are properly defined

Decompiling ESTROBJ::ptlBaseLineAdjust() – Initial Decompilation

Now that IDA has all of the prerequisite information, it is possible to utilize HexRays Decompiler to examine what the decompiled code for ESTROBJ:ptlBaseLineAdjustSet() looks like. To do this, assuming one has purchased and installed the HexRays Decompiler plugin, navigate to ESTROBJ:ptlBaseLineAdjustSet() And press **F5**. The following pseudocode should be shown:

```
Initial ESTROBJ::ptlBaseLineAdjustSet Pesudocode From HexRays Decompiler
void __fastcall ESTROBJ::ptlBaseLineAdjustSet(ESTROBJ *this, struct _POINTL *a2)
{
  struct _POINTL v2; // rax
  ULONG v3; // edx
  __int64 v4; // r9
  __int64 v5; // r10

  v2 = *a2;
  v3 = 0;
  *(struct _POINTL *)&this->lCurrentFont = v2;
  if ( (v2.x || this->ptlBaseLineAdjust.x) && this->strobj.cGlyphs )
  {
    v4 = 0i64;
    v5 = 0i64;
    do
    {
      if ( *(_DWORD *)&this->pwcPartition[v5] == HIDWORD(this->pgpNext) )
      {
        *(_DWORD *)(*(_QWORD *)&this->flTO + v4 + 16) += this->lCurrentFont;
        *(_DWORD *)(*(_QWORD *)&this->flTO + v4 + 20) += this->ptlBaseLineAdjust.x;
        ++v3;
      }
      v5 += 2i64;
      v4 += 24i64;
    }
    while ( v3 < this->strobj.cGlyphs );
  }
}
```

A quick visual inspection of this code shows there are still a few places where the decompiler's output could be improved. In particular, v3 is clearly a loop counter of some sort, given that it is initially set to 0 and is being compared against this->strobj.cGlyphs. Let's rename v3 to var_current_glyph_number so the decompiled code is clearer. Additionally, let's rename a2, aka the second argument to the function, to pPOINTL to appropriately reflect the fact that it is a pointer to POINTL structure. Finally, let's rename v2 to var_POINTL so that it is easier to see where the local copy of this parameter is being used within ESTROBJ:ptlBaseLineAdjustSet(). With these changes, the updated, decompiled code looks like the following:

```
ESTROBJ::ptlBaseLineAdjustSet Pesudocode After Initial Renaming of Variables
void __fastcall ESTROBJ::ptlBaseLineAdjustSet(ESTROBJ *this, struct _POINTL *pPOINTL)
{
  struct _POINTL var_POINTL; // rax
  ULONG var_current_glyph_number; // edx
  __int64 v4; // r9
  __int64 v5; // r10

  var_POINTL = *pPOINTL;
  var_current_glyph_number = 0;
  *(struct _POINTL *)&this->lCurrentFont = var_POINTL;
  if ( (var_POINTL.x || this->ptlBaseLineAdjust.x) && this->strobj.cGlyphs )
```

```
  {
    v4 = 0i64;
    v5 = 0i64;
    do
    {
      if ( *(_DWORD *)&this->pwcPartition[v5] == HIDWORD(this->pgpNext) )
      {
        *(_DWORD *)(*(_QWORD *)&this->flTO + v4 + 16) += this->lCurrentFont;
        *(_DWORD *)(*(_QWORD *)&this->flTO + v4 + 20) += this->ptlBaseLineAdjust.x;
        ++var_current_glyph_number;
      }
      v5 += 2i64;
      v4 += 24i64;
    }
    while ( var_current_glyph_number < this->strobj.cGlyphs );
  }
}
```

Decompiling ESTROBJ::ptlBaseLineAdjust() – Updating STROBJ

Whilst the updates have managed to make the code easier to read, it is clear upon closer inspection that there are still some noticeable issues within the pseudocode. In particular, the flTO field seems to have been relocated or removed from the ESTROBJ structure since it is being used as an array within the pseudocode, despite the ReactOS definition stating that flTO is of type FLONG, which MSDN notes is the type for “a set of 32-bit flags”.

In order to confirm that the flTO field was indeed moved; one can use either static analysis or dynamic analysis. In this case, both methods were deployed to ensure that as much information could be gathered as possible.

To do this, the VS-Labs Research Team wrote a small piece of code to execute the ESTROBJ::ptlBaseLineAdjustSet() function. A kernel debugger was then attached to the computer to allow for the examination of the this pointer passed to ESTROBJ::ptlBaseLineAdjustSet(), which is an ESTROBJ object. The following output shows the content of the this pointer:

```
Contents of this Pointer Within ESTROBJ::ptlBaseLineAdjustSet
win32kfull!ESTROBJ::ptlBaseLineAdjustSet+0x3:
ffffd4ea`ee0a0ac3 4c8bc1          mov     r8,rcx
kd> dd rcx
ffff9581`35ee7810  00000001 00000073 00000000 00000000
ffff9581`35ee7820  00000000 0000000e 00000010 00001f80
ffff9581`35ee7830  0300ef50 ffffd48d 35ee7a20 ffff9581
ffff9581`35ee7840  00000000 ffff9581 35ee77d0 ffff9581
ffff9581`35ee7850  0300ef50 ffffd48d eebb40d0 ffffd4ea
ffff9581`35ee7860  757ebce8 fffff802 757e864e fffff802
ffff9581`35ee7870  00000000 00000000 00000000 00000000
ffff9581`35ee7880  00000000 00000000 00000000 00000000
```

For reference, here is ReactOS’s definition of the first few fields of ESTROBJ:

```
First Few Lines of ReactOS’s Definition of ESTROBJ
typedef struct _ESTROBJ
{
  STROBJ    strobj;                   // 000
  ULONG     cgposCopied;              // 024
  ULONG     cgposPositionsEnumerated; // 028
  RFONTOBJ *prfo;                     // 02c
  FLONG     flTO;                     // 030
  GLYPHPOS *pgpos;                    // 034
```

Additionally, since the first field within ESTROBJ is a STROBJ, let’s quickly revisit the definition of STROBJ:

```
ReactOS’s Definition of the STROBJ Structure
typedef struct _STROBJ
{
  ULONG    cGlyphs;
  FLONG    flAccel;
  ULONG    ulCharInc;
  RECTL    rclBkGround;
  GLYPHPOS *pgp;
  LPWSTR   pwszOrg;
} STROBJ;
```

Looking at the dump above, one can see that the first three DWORDs (32-bit long blocks of data) can be mapped on to the first three elements of STROBJ, which means that cGlyphs is 1, flAccel is 0x73, and ulCharInc is 0. However, the following elements appear to have been altered.

In particular, there is supposed to be a RECTL structure named rclBkGround followed by two pointers: a GLYPHPOS pointer named pgp and then a LPWSTR pointer named pwszOrg. Looking at the definition of a RECTL structure reveals that it is made up of 4 DWORDs as shown below:

```
MSDN's Definition of the RECTL Structure
struct _RECTL
{
 LONG left;
 LONG top;
 LONG right;
 LONG bottom;
};
```

The disassembly doesn't match this even though the data starting at ffff9581`35ee7820 appears to match the format of the RECTL structure, there appears to be an extra DWORD worth of data at ffff9581`35ee781c. This can be further confirmed by examining the data at ffff9581`35ee7830 and ffff9581`35ee7838. As ReactOS's definition for STROBJ only has two parameters that are next to one another that are both pointers, namely pgp and pwszOrg, one can conclude that ffff9581`35ee7830 is pgp and ffff9581`35ee7838 is pwszOrg.

This leaves one with an issue as there is now one DWORD worth of extra data that is not being accounted for in the current ReactOS structure definition. By running the test script multiple times however, VS-Labs was able to determine that DWORD at ffff9581`35ee782C, aka offset 0x1C of STROBJ, changed between runs.

This suggests that it is not part of the RECTL structure, since the RECTL structure should remain consistent between attempts. From this, one can determine that the RECTL structure starts at ffff9581`35ee781C and that the data at ffff9581`35ee782C is some unknown 32-bit long value. With this information, one can add a new DWORD sized field named unknown to the STROBJ structure directly after the RECTL field. This should result in the following STROBJ structure:

```
VS-Lab's Updated STROBJ Structure for Windows 10 v1903 on x64
struct _STROBJ
{
 ULONG cGlyphs;
 FLONG flAccel;
 ULONG ulCharInc;
 RECTL rclBkGround;
 DWORD unknown;
 GLYPHPOS *pgp;
 LPWSTR pwszOrg;

};
```

Decompiling ESTROBJ::ptlBaseLineAdjust() – Updating ESTROBJ

With the STROBJ structure updated, one can return to the task of updating the position of the flTO field within the ESTROBJ structure. Several methods can be utilized to find the new location where flTO should be, however, VS-Labs Research Team found that the most effective method was to examine the leaked source code of ESTROBJ::bPartitionInit(), a function which used the flTO field and whose operations had not changed drastically in Windows 10.

Examining the source code for ESTROBJ::bPartitionInit() reveals that the flTO field is utilized at the very beginning of the function, where a check is made to see if the flTO field has the TO_SYS_PARTITION flag set, as can be seen in the snippet below.

```
Leaked Source Code for ESTROBJ::bPartitionInit() Showing TO_SYS_PARTITION Check
// Snippet taken from
https://github.com/ZoloZiak/WinNT4/blob/f5c14e6b42c8f45c20fe88d14c61f9d6e0386b8e/private/ntos/w32/ntgdi/

BOOL ESTROBJ::bPartitionInit(COUNT c, UINT uiNumLinks, BOOL bEudcInit)
{
    flAccel &= ~(SO_CHAR_INC_EQUAL_BM_BASE|SO_ZERO_BEARINGS);
    if(!(flTO & TO_SYS_PARTITION))
```

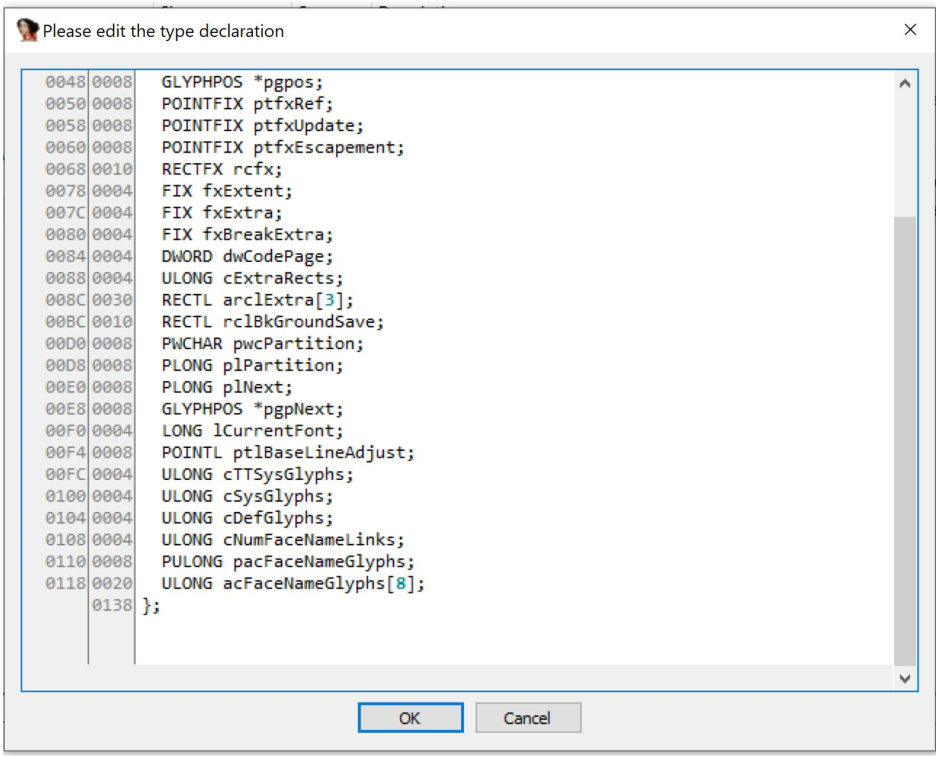
Figure 6 shows the disassembly of the Windows 10 version of this code.

```
; __int64 __fastcall ESTROBJ::bPartitionInit(ESTROBJ *__hidden this, unsigned int, unsigned int, int)
public: int ESTROBJ::bPartitionInit(unsigned long, unsigned int, int) proc near

arg_0= qword ptr  8
arg_8= qword ptr  10h

; FUNCTION CHUNK AT .text:00000001C01BD890 SIZE 00000034 BYTES

mov     [rsp+arg_0], rbx
mov     [rsp+arg_8], rsi
push    rdi
sub     rsp, 20h
and     dword ptr [rcx+4], 0FFFFFFCFh
mov     esi, r9d
test    dword ptr [rcx+0E8h], 1000h
mov     rbx, rcx
mov     edi, r8d
jnz     short loc_1C013689D
```

Figure 6 – Windows 10 bPartitionInit() initial disassembly

By referring to line 42 of textobj.hxx in the leaked NT 4.0 source code, one can translate the 0x1000 in the disassembly shown in Figure 6 to TO_SYS_PARTITION. Therefore, one can confirm that there is a test at the start of the Windows 10 version of ESTROBJ::bPartitionInit() that checks if offset 0xE8 of the ESTROBJ structure contains the flag TO_SYS_PARITION, and will jump to loc_1C013689D if it does not. This matches the if(!(flTO & TO_SYS_PARTITION)) line in the leaked source code, which confirms that offset 0xE8 of the Windows 10 ESTROBJ structure is flTO.

Before one can update the ESTROBJ structure, however, they need to identify what data already exists at offset 0xE8 of ReactOS's definition of the ESTROBJ structure to determine if any additional fields need to be removed or relocated. To find this out, go back to the Local Types window, search for the _ESTROBJ type, and right-click and select **Edit** on the structure. The window shown in Figure 7 should appear (note that only IDA Pro v7.4 and later has support for offset information for structures, so this will not be displayed if you are running an earlier version of IDA Pro):




```
0048 0008   GLYPHPOS *pgpos;
0050 0008   POINTFIX ptfxRef;
0058 0008   POINTFIX ptfxUpdate;
0060 0008   POINTFIX ptfxEscapement;
0068 0010   RECTFX rcfx;
0078 0004   FIX fxExtent;
007C 0004   FIX fxExtra;
0080 0004   FIX fxBreakExtra;
0084 0004   DWORD dwCodePage;
0088 0004   ULONG cExtraRects;
008C 0030   RECTL arclExtra[3];
00BC 0010   RECTL rclBkGroundSave;
00D0 0008   PWCHAR pwcPartition;
00D8 0008   PLONG plPartition;
00E0 0008   PLONG plNext;
00E8 0008   GLYPHPOS *pgpNext;
00F0 0004   LONG lCurrentFont;
00F4 0008   POINTL ptlBaseLineAdjust;
00FC 0004   ULONG cTTSysGlyphs;
0100 0004   ULONG cSysGlyphs;
0104 0004   ULONG cDefGlyphs;
0108 0004   ULONG cNumFaceNameLinks;
0110 0008   PULONG pacFaceNameGlyphs;
0118 0020   ULONG acFaceNameGlyphs[8];
     0138 };
```

Figure 7 – Viewing outdated ESTROBJ structure

From the output shown in Figure 7, one can observe that offset 0xE8 in the ReactOS ESTROBJ structure definition is currently pgpNext. Let's update this structure to relocate the flTO element from offset 0x40 of ESTROBJ to offset 0xE8. The new structure is shown in the snippet below:

```
VS-Labs' Updated ESTROBJ Structure for Windows 10 v1903 on x64
struct _ESTROBJ
{
STROBJ strobj;
ULONG cgposCopied;
```

```
ULONG cgposPositionsEnumerated;
RFONTOBJ *prfo;
GLYPHPOS *pgpos;
POINTFX ptfxRef;
POINTFX ptfxUpdate;
POINTFX ptfxEscapement;
RECTFX rcfx;
FIX fxExtent;
FIX fxExtra;
FIX fxBreakExtra;
DWORD dwCodePage;
ULONG cExtraRects;
RECTL arclExtra[3];
RECTL rclBkGroundSave;
PWCHAR pwcPartition;
PLONG plPartition;
PLONG plNext;
GLYPHPOS *pgpNext;
FLONG flTO;
LONG lCurrentFont;
POINTL ptlBaseLineAdjust;
ULONG cTTSysGlyphs;
ULONG cSysGlyphs;
ULONG cDefGlyphs;
ULONG cNumFaceNameLinks;
PULONG pacFaceNameGlyphs;
ULONG acFaceNameGlyphs[8];
};
```

Decompiling ESTROBJ::ptlBaseLineAdjust() – Double Checking the Decompilation Results

As we have now updated quite a few structure definitions, it would be good idea to check that the modifications have achieved the desired effect. The following snippet shows the decompiler’s view of ESTROBJ::ptlBaseLineAdjustSet now that the ESTROBJ structure definition has been updated:

```
Updated ESTROBJ::ptlBaseLineAdjustSet Pesudocode Using New STROBJ and ESTROBJ
Definitions
void __fastcall ESTROBJ::ptlBaseLineAdjustSet(ESTROBJ *this, struct _POINTL *pPOINTL)
{
  struct _POINTL var_POINTL; // rax
  ULONG var_current_glyph_number; // edx
  __int64 v4; // r9
  __int64 v5; // r10

  var_POINTL = *pPOINTL;
  var_current_glyph_number = 0;
  this->ptlBaseLineAdjust = var_POINTL;
  if ( (var_POINTL.x || this->ptlBaseLineAdjust.y) && this->strobj.cGlyphs )
  {
    v4 = 0i64;
    v5 = 0i64;
    do
    {
      if ( this->plPartition[v5] == this->lCurrentFont )
      {
        this->pgpos[v4].ptl.x += this->ptlBaseLineAdjust.x;
        this->pgpos[v4].ptl.y += this->ptlBaseLineAdjust.y;
        ++var_current_glyph_number;
      }
      ++v5;
      ++v4;
    }
    while ( var_current_glyph_number < this->strobj.cGlyphs );
  }
}
```

By examining this pseudocode, it possible to determine that v4 is a variable that controls the current entry within the pgpos array that is being processed. Similarly, v5 is a variable that controls the current entry within the this-

>plPartition array that is being checked against this->lCurrentFont. Given this information, let's rename this->v4 to var_pgpos_entry and v5 to var_plPartitionEntry. With these edits the code should look much cleaner and should be very easy to understand.

```
ESTROBJ::ptlBaseLineAdjustSet Pesudocode After Renaming v5 and v4
void __fastcall ESTROBJ::ptlBaseLineAdjustSet(ESTROBJ *this, struct _POINTL *pPOINTL)
{
  struct _POINTL var_POINTL; // rax
  ULONG var_current_glyph_number; // edx
  __int64 var_pgpos_entry; // r9
  __int64 var_plPartitionEntry; // r10

  var_POINTL = *pPOINTL;
  var_current_glyph_number = 0;
  this->ptlBaseLineAdjust = var_POINTL;
  if ( (var_POINTL.x || this->ptlBaseLineAdjust.y) && this->strobj.cGlyphs )
  {
    var_pgpos_entry = 0i64;
    var_plPartitionEntry = 0i64;
    do
    {
      if ( this->plPartition[var_plPartitionEntry] == this->lCurrentFont )
      {
        this->pgpos[var_pgpos_entry].ptl.x += this->ptlBaseLineAdjust.x;
        this->pgpos[var_pgpos_entry].ptl.y += this->ptlBaseLineAdjust.y;
        ++var_current_glyph_number;
      }
      ++var_plPartitionEntry;
      ++var_pgpos_entry;
    }
    while ( var_current_glyph_number < this->strobj.cGlyphs );
  }
}
```

Decompiling ESTROBJ::ptlBaseLineAdjust() – Final Checks

Whilst the code looks correct now, to provide complete assurance, it is necessary to double-check that the this->plPartition and this->lCurrentFont fields are located at offset 0xD0 and 0xEC of the ESTROBJ structure respectively.

An initial bit of reassurance can be obtained by looking at the leaked source code for ESTROBJ::bTextToPath(). A close inspection will reveal that one of its lines performs a very similar operation to the one in the pseudocode the decompiler generated for ESTROBJ::ptlBaselineAdjustSet().

In particular, in both functions a loop is utilized which iterates over cGlyphs elements within the plPartition array, and on each iteration lCurrentFont is compared to the current entry being processed within plPartition to see if they match. Therefore, one can conclude that comparing this->plPartition and this->lCurrentFont with one another is normal, as this operation has been performed in the past.

To obtain complete assurance, however, one can once again examine the leaked source code of ESTROBJ::bPartitionInit(). By reviewing the code, one can find that the plPartition field is NULL'd out via RtlZeroMemory(). If one returns to the disassembly of ESTROBJ::bPartitionInit() in IDA Pro, as shown in Figure 8, it should be possible to observe a similar pattern whereby offset 0xD0 of the ESTROBJ object contained in RBX is NULL'd out via a memset() call. This replicates the RtlZeroMemory() call shown in the leaked code and confirms that offset 0xD0 of ESTROBJ is indeed plPartition on Windows 10.

```
mov     rax, [rcx+40h]
mov     r8d, edx
lea     rdx, [r8+r8*2]
shl     r8, 2           ; Size
lea     rcx, [rax+rdx*8] ; void *
xor     edx, edx        ; Val
mov     [rbx+0D0h], rcx ; Set offset 0xD0 of the ESTROBJ in RBX to the
                        ; destination address of the memset() operation.
lea     rax, [rcx+r8]
mov     [rbx+0C8h], rax
call    memset          ; Call memset() which will zero out the plPartition field.
and     qword ptr [rbx+108h], 0
and     dword ptr [rbx+0FCh], 0
and     dword ptr [rbx+100h], 0
and     dword ptr [rbx+0F8h], 0
bts     dword ptr [rbx+0E8h], 0Ch
```

Figure 8 – ESTROBJ::bPartitionInit() zeroing out the memory at offset 0xD0 of the ESTROBJ

Finally, to confirm the offset of this->lCurrentFont within ESTROBJ, let's examine the leaked source code for STROBJ_bEnumLinked(). The following snippet shows the relevant lines of this function:

```
Leaked STROBJ_bEnumLinked Source Code Lines
BOOL STROBJ_bEnumLinked(ESTROBJ *peso, ULONG *pc,PGLYPHPOS *ppgpos)
{
  // Quick exit.
  if ( peso->cgposCopied == 0 )
  {
    for ( peso->plNext = peso->plPartition, peso->pgpNext = peso->pgpos;
      *(peso->plNext) != peso->lCurrentFont;
      (peso->pgpNext)++, (peso->plNext)++ );
      {
        ...
      }
  }
  ...
```

Figure 9 shows the corresponding disassembly for the STROBJ_bEnumLinked function in Windows 10 (comments added for additional clarity).

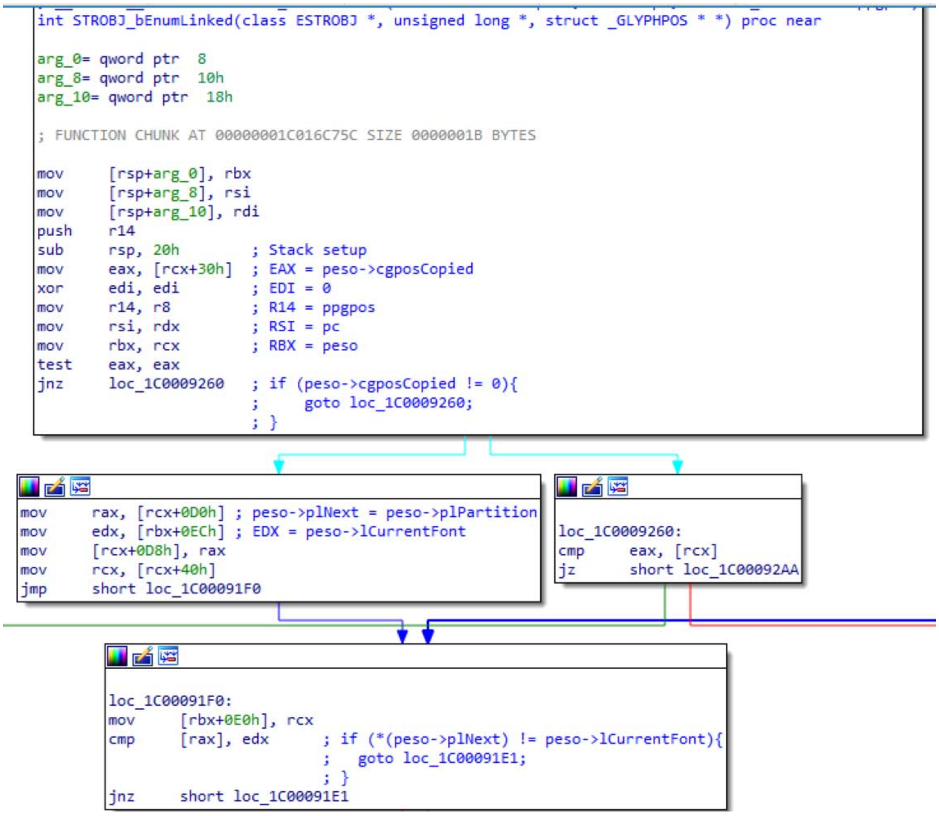


Figure 9 – Disassembly of opening lines of STROBJ_bEnumLinked

A quick glance over the disassembly in Figure 9 reveals that there is a check to see if peso->cgposCopied is 0. This can be confirmed as offset 0x30 of ESTROBJ is moved into EAX, after which EAX is checked to ensure it is not 0. If it is 0, a jump is taken to loc_1C0009260, which has the effect of ensuring that the for loop starting at loc_1C00091F0 is never entered.

An examination of the next block, which is on the left side of the branch in Figure 9, reveals that RAX is set to plPartition as RCX contains the value of the peso parameter, which is of type ESTROBJ, and our earlier analysis confirmed that offset 0xD0 of an ESTROBJ is plPartition. We can also verify that the following line, mov EDX, [RBX+0xEC], sets EDX to offset 0xEC of the ESTROBJ structure. At this point in the analysis, we do not know what this offset is related to, so we will make a note to come back to this and will continue our analysis.

If one skips over the following lines they will see that there is a check at loc_1C00091F0 which will compare RAX, or peso->plPartition to offset 0xEC of the ESTROBJ, aka EDX. If no match is made then execution jumps to loc_1C00091E1 which increments the pointers before execution returns to loc_1C00091F0, or the start of the loop. This can be seen in Figure 10.

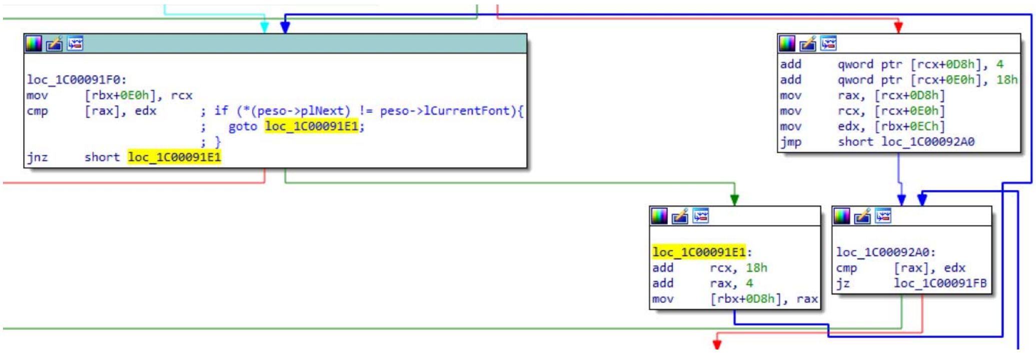


Figure 10 – Disassembly of STROBJ_bEnumLinked's for loop

If one re-examines the leaked source code, one will notice that this behavior looks very similar to the following line, particularly as this is the only check done in its for loop:

```
Initializing the Value That peso->plNext Points to Within the Leaked Source Code
*(peso->plNext) != peso->lCurrentFont;
```

Tracing peso->plNext's usage in the leaked source code shows that it is initialized to peso->plPartition:

```
Initializing peso->plNext in Leaked Source Code
peso->plNext = peso->plPartition
```

This ultimately means that earlier, when RAX was set to RCX+0xD0, it was actually being set to the value of peso->plPartition. This confirms that offset 0xD0 of the ESTROBJ is indeed plPartition. Furthermore, the check between [RAX] and EDX is really a check between *(peso->plNext), aka RAX, and peso->lCurrentFont, aka EDX. Since EDX is set earlier on to the value of RBX+0xEC, and RBX in turn is set earlier on in the code to the value of ECX, aka the peso parameter, it is possible to confirm that offset 0xEC of ESTROBJ is the lCurrentFont field in Windows 10.

With these checks complete, we now have complete assurance that the definition of the ESTROBJ structure is correct and that the pseudocode generated by HexRays Decompiler is as accurate as it can be.

Conclusion

Whilst undocumented structures are often seen as untouchable items that require hours of time and dedication to reverse engineer, this is not always the case. Sometimes all that is needed is a little bit of background information or context to proceed. Be sure to use publicly available information from forums, GitHub, and ReactOS; with more resources, comes more potential leads when it comes time to investigate.

Additionally, don't forget that the best way to verify how a structure has changed is to examine it yourself. Utilize IDA Pro to perform static analysis and identify where the structure is being used, then follow this up with WinDBG to identify what fields exist in the structure and what their types might be.

Finally, remember that all of this takes time. Some structures may be more complex and contain many other structures nested inside them. If possible, start with smaller, simpler structures first and then build up from there. For more complex structures, be sure to make use of IDA Pro's type database and dynamic analysis to ensure that all members of the structure have been updated appropriately and there are no unexpected changes.