Igor’s tip of the week

season two

from 20/08/2021 to 26/08/2022

hex-rays

Welcome to Igor’s Tip Season 2! With this edition, we are looking to build on the success of the previous one and give a broader scope to the users of what IDA is capable of. As usual, Igor begins with some basic and advanced usage of IDA features, and then he touches on working with types. In the next section, Igor reveals a few not-so-widely-known functionalities that can improve your work. Much attention has been put on the Decompiler and how to get the most out of it. In the last two sections, Igor talks briefly about automating repetitive tasks and how to customize IDA’s User Interface to suit your workflow better.

Finally, we hope you enjoy this Season 2 and follow Igor’s Tip every Friday!

Usage: basic and advanced usage of IDA features

Types: working with types

Hidden: hidden gems, not widely known but useful functionality

Decompiler: related to the Hex-Rays decompiler

Automation: automating repetitive tasks

Customization: customizing IDA UI to better suit your workflow

a blog series on
HEX-RAYS BLOG

CHECK ALL ARTICLES : **WWW.HEX-RAYS.COM/BLOG/**

#53: Manual switch idioms

20 Aug 2021

https://hex-rays.com/blog/igors-tip-of-the-week-53-manual-switch-idioms/

IDA supports most of the switch patterns produced by major compilers out-of-box and usually you don't need to worry about them. However, occasionally you may encounter a code which has been produced by an unusual or a very recent compiler version, or some peculiarity of the code prevented IDA from recognizing the pattern, so it may become necessary to help IDA and tell it about the switch so a proper function graph can be presented and decompiler can produce nice pseudocode.

Switch pattern components

The common switch pattern is assumed to have the following components:

1. indirect jump
   This is an instruction which actually performs the jump to the destination block handling the switch case; usually involves some register holding the address value;
2. jump table
   A table of values, containing either direct addresses of the destination blocks, or some other values allowing to calculate those addresses (e.g. offsets from some base address). It has to be of a specific fixed size (number of elements) and the values may be scaled with a shift value. Some switches may use two tables, first containing indexes into the second one with addresses.
3. input register
   register containing the initial value which is being used to determine the destination block. Most commonly, it is used to index the jump table.

Switch formula

The standard switches are assumed to use the following calculation for the destination address:

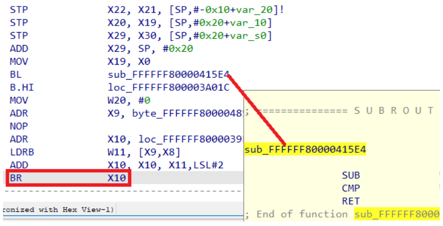
```
        target = base +/- (table_element << shift)
```

`base` and `shift` can be set to zero if not used.

Example

Here's a snippet from an ARM64 firmware.

The indirect jump is highlighted with the red rectangle. Here's the same code in text format:



```
STP             X22, X21, [SP,#-0x10+var_20]!
STP             X20, X19, [SP,#0x20+var_10]
STP             X29, X30, [SP,#0x20+var_s0]
ADD             X29, SP, #0x20
MOV             X19, X0
BL              sub_FFFFFF80000415E4
B.HI            loc_FFFFFF800003A01C
MOV             W20, #0
ADR             X9, byte_FFFFFF800004859
NOP
ADR             X10, loc_FFFFFF8000039
LDRB            W11, [X9,X8]
ADD             X10, X10, X11,LSL#2
BR              X10
```

```
; =============== S U B R O U T

sub_FFFFFF80000415E4

                SUB
                CMP
                RET
; End of function sub_FFFFFF8000
```

```
__text:FFFFFF8000039F88 STP X22, X21, [SP,#-0x10+var_20]!
__text:FFFFFF8000039F8C STP X20, X19, [SP,#0x20+var_10]
__text:FFFFFF8000039F90 STP X29, X30, [SP,#0x20+var_s0]
__text:FFFFFF8000039F94 ADD X29, SP, #0x20
__text:FFFFFF8000039F98 MOV X19, X0
__text:FFFFFF8000039F9C BL sub_FFFFFF80000415E4
__text:FFFFFF8000039FA0 B.HI loc_FFFFFF800003A01C
__text:FFFFFF8000039FA4 MOV W20, #0
__text:FFFFFF8000039FA8 ADR X9, byte_FFFFFF8000048593
__text:FFFFFF8000039FAC NOP
__text:FFFFFF8000039FB0 ADR X10, loc_FFFFFF8000039FC0
__text:FFFFFF8000039FB4 LDRB W11, [X9,X8]
__text:FFFFFF8000039FB8 ADD X10, X10, X11,LSL#2
__text:FFFFFF8000039FBC BR X10
```

We can see that the register used in the indirect branch (X10) is a result of some calculation so it is probably a switch pattern. However, because the code was compiled with size optimization (the range check is moved into a separate function used from several places), IDA was not able to match the pattern in the automatic fashion. Let's see if we can find out components of the standard switch described above.

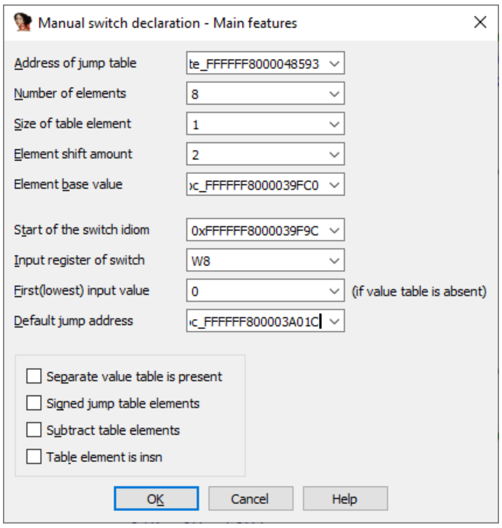
The formula matches the instruction `ADD X10, X10, X11,LSL#2`(in C syntax: `X10 = X10+(X11<<2)`). We can see that the table element (`X11`) is shifted by 2 before being added to the base (`X10`). The value of `X11` comes from the previous load of `W11` using `LDRB` (load byte) from the table at `X9` and index `X8`. Thus:

1. Indirect jump: yes, the `BR X10` instruction at `FFFFFF8000039FBC`.
2. jump table: yes, at `byte_FFFFFF8000048593`. Additionally, we have a base at `loc_FFFFFF8000039FC0` and shift value of **2**. It contains **eight** elements (this can be checked visually or deduced from the range check which uses **7** as the maximum allowed value).
3. input register: yes, `X8` is used to index the table (we can also use **W8** which is the 32-bit part of `X8` and is used by the range check function.

#53: Manual switch idioms

20 Aug 2021
https://hex-rays.com/blog/igors-tip-of-the-week-53-manual-switch-idioms/

Now that we have everything, we can specify the pattern by putting the cursor on the indirect branch and invoking Edit > Other > Specify switch idiom…

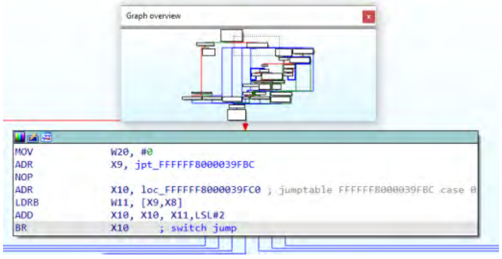


Manual switch declaration - Main features

Address of jump table	te_FFFFFF8000048593
Number of elements	8
Size of table element	1
Element shift amount	2
Element base value	›c_FFFFFF8000039FC0
Start of the switch idiom	0xFFFFFF8000039F9C
Input register of switch	W8
First(lowest) input value	0 (if value table is absent)
Default jump address	›c_FFFFFF800003A01C

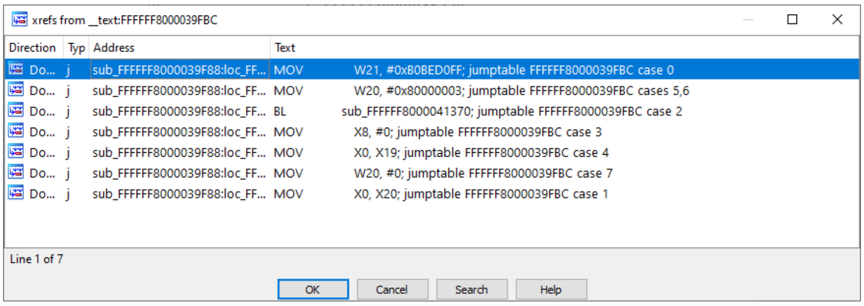
☐ Separate value table is present
☐ Signed jump table elements
☐ Subtract table elements
☐ Table element is insn

The values can be specified in C syntax (0x…) or as labels thanks to the `expression evaluation`[1] feature. Once the dialog is confirmed, we can observe the switch nicely labeled and function graph updated to include newly reachable nodes.



```
MOV             W20, #0
ADR             X9, jpt_FFFFFF8000039FBC
NOP
ADR             X10, loc_FFFFFF8000039FC0 ; jumptable FFFFFF8000039FBC case 0
LDRB            W11, [X9,X8]
ADD             X10, X10, X11,LSL#2
BR              X10     ; switch jump
```

We can also use “List cross-references from…” (`Ctrl–J`) to see the list of targets from the indirect jump.



xrefs from __text:FFFFFF8000039FBC

Direction	Typ	Address	Text
Do…	j	sub_FFFFFF8000039F88:loc_FF…	MOV W21, #0xB0BED0FF; jumptable FFFFFF8000039FBC case 0
Do…	j	sub_FFFFFF8000039F88:loc_FF…	MOV W20, #0x80000003; jumptable FFFFFF8000039FBC cases 5,6
Do…	j	sub_FFFFFF8000039F88:loc_FF…	BL sub_FFFFFF8000041370; jumptable FFFFFF8000039FBC case 2
Do…	j	sub_FFFFFF8000039F88:loc_FF…	MOV X8, #0; jumptable FFFFFF8000039FBC case 3
Do…	j	sub_FFFFFF8000039F88:loc_FF…	MOV X0, X19; jumptable FFFFFF8000039FBC case 4
Do…	j	sub_FFFFFF8000039F88:loc_FF…	MOV W20, #0; jumptable FFFFFF8000039FBC case 7
Do…	j	sub_FFFFFF8000039F88:loc_FF…	MOV X0, X20; jumptable FFFFFF8000039FBC case 1

Line 1 of 7

Additional options

Our example was pretty straightforward but in some cases you can make use of the additional options in the dialog.

1. separate value table is present: when a two-level table is used, i.e.:

   `table_element = jump_table[value_table[input_register]];` instead of the default `table_element = jump_table[input_register];`

2. signed jump table elements: when table elements are loaded using a sign-extension instruction, for example `LDRSB` or `LDRSW` on ARM or `movsx` on x86.
3. Subtract table elements: if the values are subtracted from the base instead of being added (minus sign is used in the formula).
4. Table element is insn: the “jump table” contains instructions instead of data values. This is used in some architectures which can perform relative jumps using a delta value from the instruction pointer. For example, the legacy ARM jumps using direct `PC` manipulation:

```
                CMP             R3, #7 ; SWITCH ; switch 8 cases
                ADDLS           PC, PC, R3,LSL#2 ; switch jump
; ---------------------------------------------------------------------------

loc_6684                                ; CODE XREF: __pthread_manager+1BC↑j
                B               def_6680 ; jumptable 00006680 default case, c
; ---------------------------------------------------------------------------

loc_6688                                ; CODE XREF: __pthread_manager+1BC↑j
                B               loc_66A8 ; jumptable 00006680 case 0
; ---------------------------------------------------------------------------

loc_668C                                ; CODE XREF: __pthread_manager+1BC↑j
                B               loc_6854 ; jumptable 00006680 case 1
; ---------------------------------------------------------------------------

loc_6690                                ; CODE XREF: __pthread_manager+1BC↑j
                B               loc_68CC ; jumptable 00006680 case 2
; ---------------------------------------------------------------------------

loc_6694                                ; CODE XREF: __pthread_manager+1BC↑j
                B               loc_695C ; jumptable 00006680 case 3
; ---------------------------------------------------------------------------

loc_6698                                ; CODE XREF: __pthread_manager+1BC↑j
                B               loc_6990 ; jumptable 00006680 case 4
; ---------------------------------------------------------------------------

loc_669C                                ; CODE XREF: __pthread_manager+1BC↑j
                B               loc_69FC ; jumptable 00006680 case 5
; ---------------------------------------------------------------------------

loc_66A0                                ; CODE XREF: __pthread_manager+1BC↑j
                B               def_6680 ; jumptable 00006680 default case, c
; ---------------------------------------------------------------------------

loc_66A4                                ; CODE XREF: __pthread_manager+1BC↑j
                B               loc_699C ; jumptable 00006680 case 7
; ---------------------------------------------------------------------------
```

Usually in such situation the table “elements” are fixed-size branches to the actual destinations.

Optional values

Some values can be omitted by default but you may also fill them for a more complete mapping to the original code:

1. Input register of switch: can be omitted if you only need cross-references for the proper function flow graph but it has to be specified if you want decompiler to properly parse and represent the switch.
2. First(lowest) input value: the value of the input register corresponding to the entry 0 of the jump table. In the example above, we can see that the range check calculates `W8 = W1 - 1`, so we could specify lowest value of 1 (this would also update the comments at the destination addresses to be 1 to 8 instead of 0 to 7).
3. default jump address: the address executed when the input range check fails (in our example – destination of the `B.HI` instruction). Can make the listing and/or decompilation a little more neat but is not strictly required otherwise.

For even more detailed info about supported switch patterns, see the `switch_info_t` structure[2] and the uiswitch plugin source code in the SDK. If you encounter a switch which cannot be handled by the standard formula, you can also look into writing a `custom jump table handler`[3].

[1] https://hex-rays.com/blog/igors-tip-of-the-week-21-calculator-and-expression-evaluation-feature-in-ida/
[2] https://hex-rays.com/products/ida/support/sdkdoc/structswitch__info__t.html
[3] https://hex-rays.com/blog/jump-tables/

#54: Shifted pointers

27 Aug 2021

https://hex-rays.com/blog/igors-tip-of-the-week-54-shifted-pointers/

Previously[1] we briefly mentioned shifted pointers but without details. What are they?

Shifted pointers is another custom extension to the C syntax. They are used by IDA and decompiler to represent a pointer to an object with some offset or adjustment (positive or negative). Let's see how they work and several situations where they can be useful.

Shifted pointer description and syntax

A shifted pointer is a regular pointer with additional information about the name of the parent structure and the offset from its beginning. For example, consider this structure:

```
struct mystruct
{
 char buf[16];
 int dummy;
 int value;            // <- myptr points here
 double fval;
};
```

And this pointer declaration:

```
int *__shifted(mystruct,20) myptr;
```

It means that `myptr` is a pointer to `int` and if we decrement it by **20** bytes, we end up with `mystruct*`.

In fact, the offset value is not limited to the containing structure and can even be negative. Also, the "parent" type does not have to be a structure but can be any type except void. This can be useful in some situations.

Whenever a shifted pointer is used with an adjustment, it will be displayed using the ADJ helper, a pseudo-operator which returns the pointer to the parent type (in our case mystruct). For example, if the pointer is dereferenced after adding 4 bytes, it can be represented like this:

```
ADJ(myptr)->fval
```

Optimized loop on array of structures

When compiling code which is processing an array of structures, a compiler may optimize the loop so that the "current item" pointer points into a middle of the structure instead of the beginning. This is especially common when only a small subset of fields are being accessed. Consider this example:

```
struct mydata
{
  int a, b, c;
  void *pad[2];
  int d, e, f;
  char path[260];
};

int sum_c_d(struct mydata *arr, int count)
{
    int sum=0;
    for (int i=0; i< count; i++)
    {
        sum+=arr[i].d+arr[i].c*43;
    }
    return sum;
}
```

When compiled with Microsoft Visual C++ x86, it can produce the following code:

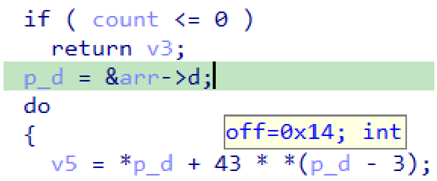
```
?sum_c_d@@YAHPAUmydata@@H@Z proc near

arg_0 = dword ptr  4
arg_4 = dword ptr  8

        mov     edx, [esp+arg_4]
        push    esi
        xor     esi, esi
        test    edx, edx
        jle     short loc_25
        mov     eax, [esp+4+arg_0]
        add     eax, 14h

loc_12:                                 ; CODE XREF: sum_c_d(mydata *,int)+23↓j
        imul    ecx, [eax-0Ch], 2Bh ; ‘+’
        add     ecx, [eax]
        lea     eax, [eax+124h]
        add     esi, ecx
        sub     edx, 1
        jnz     short loc_12

loc_25:                                 ; CODE XREF: sum_c_d(mydata *,int)+9↑j
        mov     eax, esi
        pop     esi
        retn
```

And initial decompilation looks quite strange even after adding and specifying the correct types:

```
int __cdecl sum_c_d(struct mydata *arr, int count)
{
  int v2; // edx
  int v3; // esi
  int *p_d; // eax
  int v5; // ecx

  v2 = count;
  v3 = 0;
  if ( count <= 0 )
    return v3;
  p_d = &arr->d;
  do
  {
    v5 = *p_d + 43 * *(p_d - 3);
    p_d += 73;
    v3 += v5;
    --v2;
  }
  while ( v2 );
  return v3;
}
```

Apparently, the compiler decided to use the pointer to the `d` field and accesses `c` relative to it. How can we make this look nicer?

We can find out the offset at which d is situated in the structure via manual calculation, by inspecting disassembly, or by hovering the mouse over it in pseudocode.



```
  if ( count <= 0 )
    return v3;
  p_d = &arr->d;
  do
  {                   off=0x14; int
    v5 = *p_d + 43 * *(p_d - 3);
```

Thus, we can change the type of `p_d` to `int * __shifted(mydata, 0x14)` to get improved pseudocode:

```
int __cdecl sum_c_d(struct mydata *arr, int count)
{
  int v2; // edx
  int v3; // esi
  int *__shifted(mydata,0x14) p_d; // eax
  int v5; // ecx

  v2 = count;
  v3 = 0;
  if ( count <= 0 )
    return v3;
  p_d = &arr->d;
  do
  {
    v5 = ADJ(p_d)->d + 43 * ADJ(p_d)->c;
    p_d += 73;
    v3 += v5;
    --v2;
  }
  while ( v2 );
  return v3;
}
```

Prepended metadata

This technique is used in situations where a raw block of memory needs to have some management info attached to it, i.e. heap allocators, managed strings and so on.

As a specific example, let's consider the classic MFC 4.x CString class. It uses a structure placed before the actual character array:

```
struct CStringData
{
    long  nRefs;    // reference count
    int   nDataLength;    // length of data (including terminator)
    int   nAllocLength;   // length of allocation
    // TCHAR data[nAllocLength]

    TCHAR* data()         // TCHAR* to managed data
    {
        return (TCHAR*)(this+1);
    }
};
```

The `CString` class itself has just one data member:

```
class CString
{
public:
// Constructors
[...skipped]
private:
    LPTSTR   m_pchData;        // pointer to ref counted string data

    // implementation helpers
    CStringData* GetData() const;
[...skipped]
};
inline
CStringData*
CString::GetData(
    ) const
```

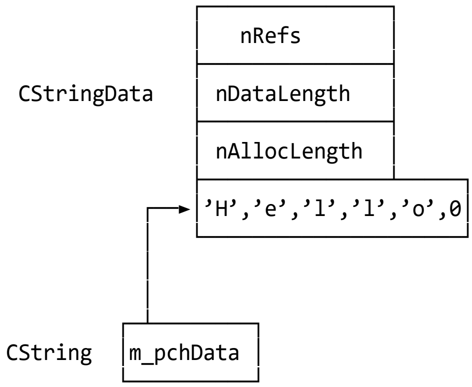
#54: Shifted pointers

27 Aug 2021

https://hex-rays.com/blog/igors-tip-of-the-week-54-shifted-pointers/

```
{
    ASSERT(m_pchData != NULL);
    return ((CStringData*)m_pchData)-1;
}
```

Here’s how it looks in memory:



Here’s how the CString’s destructor looks like in initial decompilation:

```
void __thiscall CString::~CString(CString *this)
{
  if ( *(_DWORD *)this - (_DWORD)off_4635E0 != 12 && InterlockedDecrement((volatile LONG *)(*(_DWORD *)this
- 12)) <= 0 )
    operator delete((void *)(*(_DWORD *)this - 12));
}
```

Even after creating a CString structure with a single member `char *m_pszData` it’s still somewhat confusing:

```
void __thiscall CString::~CString(CString *this)
{
  if ( this->m_pszData - (char *)off_4635E0 != 12 && InterlockedDecrement((volatile LONG *)this->m_pszData
- 3) <= 0 )
    operator delete(this->m_pszData - 12);
}
```

Finally, if we create the `CStringData` struct as described above and change the type of the CString member to: `char *__shifted(CStringData,0xC) m_pszData`:

void __thiscall CString::~CString(CString *this)

```
{
  if ( ADJ(this->m_pszData)->data - (char *)off_4635E0 != 12 && InterlockedDecrement(&ADJ(this->m_psz-
Data)->nRefs) <= 0 )
    operator delete(ADJ(this->m_pszData));
}
```

Now the code is more understandable: if the decremented reference count becomes zero, the CStringDatainstance is deleted.

More info: IDA Help: Shifted pointers[2]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-52-special-attributes/t

[2] https://hex-rays.com/products/ida/support/idadoc/1695.shtml

#55: Using debug symbols

03 Sep 2021

https://hex-rays.com/blog/igors-tip-of-the-week-55-using-debug-symbols/

IDA supports many file formats, among them the main ones used on the three major operating systems:

- PE (Portable Executable) on Windows;
- ELF (Executable and Linkable Format) on Linux;
- Mach-O (Mach object) on macOS.

Symbols and debugging information

Symbols associate locations inside the file (e.g. addresses of functions or variables) with textual names (usually the names used in the original source code). The part of the file storing this association is commonly called *symbol table*. Symbols can be stored in the file itself or separately.

Traditionally, the PE files do not contain any symbols besides those that are required for imports or exports for inter-module linking. ELF and Mach-O commonly do keep names for global functions, however most of this information can be removed, or stripped, without affecting execution of the file. Because such information is very valuable for possible debugging later, it can be stored in a separate debug information file.

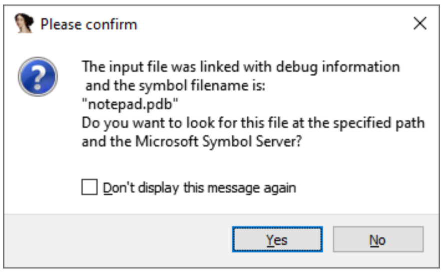
For PE files, a common debug format is **PDB** (Program Database), although other formats were used in the past, for example **TDS** (Turbo Debugger Symbols) was used by Borland compilers, and DBG in legacy versions of Visual Studio. Both ELF and Mach-O use DWARF[1]. All of the above can contain not only plain symbols but also **types** (structures, enums, typedefs), function **prototypes**, information on **local variables** as well as mapping of binary code to **source files** (filenames and line numbers).

Although originally intended to improve debugging experience, all this information obviously makes the reverse engineering process much easier, so IDA supports these formats out of box, using standard plugins shipped with IDA:

- pdb for PDB;
- tds for TDS;
- dbg for DBG;
- dwarf for DWARF.

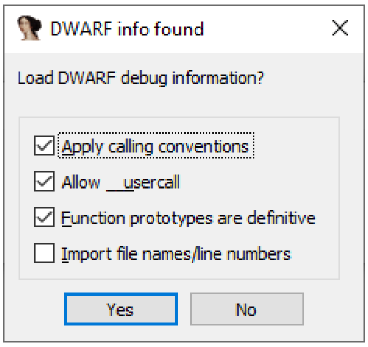
Automatic debug info loading

Standard file loaders detect when the file has been built with debug information and invoke the corresponding debug info loader. If debug info is found in the input file, next to it, or in another well-known location, the user is prompted whether to load it.


Please confirm

The input file was linked with debug information
and the symbol filename is:
"notepad.pdb"
Do you want to look for this file at the specified path
and the Microsoft Symbol Server?

□ Don't display this message again


DWARF info found

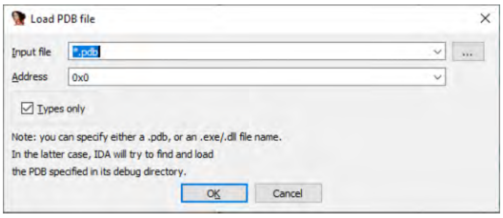
Load DWARF debug information?

☑ Apply calling conventions
☑ Allow __usercall
☑ Function prototypes are definitive
□ Import file names/line numbers

Manual debug info loading

If the separate debug info file is not present in standard location or discovered later, after you've already loaded the file, it can be loaded manually. Currently only PDB and DWARF can be loaded using this option.

- For PDB, use File > Load file > PDB File...
- For DWARF, Edit > Plugins > Load DWARF File


Load PDB file

Input file: *.pdb

Address: 0x0

☑ Types only

Note: you can specify either a .pdb, or an .exe/.dll file name.
In the latter case, IDA will try to find and load
the PDB specified in its debug directory.

#55: Using debug symbols

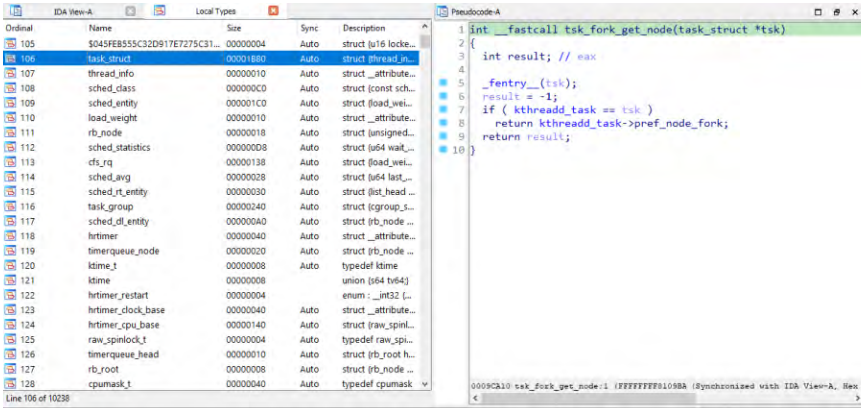
03 Sep 2021
https://hex-rays.com/blog/igors-tip-of-the-week-55-using-debug-symbols/

For the PDB loader, you can specify a DLL or EXE file instead of the PDB; in that case IDA will try to find and load a matching PDB for it, including downloading it from symbol servers if necessary. By using the “Types only” option, you can import types from an arbitrary PDB and not necessarily PDB for the current file. For example, PDB for the Windows kernel (ntoskrnl.exe) contains various structures used in kernel-mode code (drivers etc.) so this feature can be useful when reverse-engineering files without available debug info.

Example: Linux kernel debug info

Linux kernels are usually stripped during build, however many distros provide separate debug info repositories[2], or you can recompile the kernel with debug info[3]. How to load it into IDA?

For self-built kernel it’s pretty simple – the **vmlinux**file is a normal ELF which can be simply loaded into IDA. However, the pre-built kernels are usually distributed as **vmlinuz** which is a PE file (so that it can be booted directly by the UEFI firmware), with the actual kernel code stored as compressed payload inside it. The unpacked kernel can be extracted manually[4] or using the vmlinux-to-elf project[5], loaded into IDA, and the external debuginfo file can then be loaded via Edit > Plugins > Load DWARF File, producing a nice database with all kernel types and proper function prototypes.



Ordinal	Name	Size	Sync	Description
105	$045FEB555C32D917E7275C31...	00000004	Auto	struct {u16 locke...
106	task_struct	00001B80	Auto	struct {thread_in...
107	thread_info	00000010	Auto	struct _attribute...
108	sched_class	000000C0	Auto	struct {const sch...
109	sched_entity	000001C0	Auto	struct {load_wei...
110	load_weight	00000010	Auto	struct _attribute...
111	rb_node	00000018	Auto	struct {unsigned...
112	sched_statistics	000000D8	Auto	struct {u64 wait_...
113	cfs_rq	00000138	Auto	struct {load_wei...
114	sched_avg	00000028	Auto	struct {u64 last_...
115	sched_rt_entity	00000030	Auto	struct {list_head ...
116	task_group	00000240	Auto	struct {cgroup_s...
117	sched_dl_entity	000000A0	Auto	struct {rb_node ...
118	hrtimer	00000040	Auto	struct _attribute...
119	timerqueue_node	00000020	Auto	struct {rb_node ...
120	ktime_t	00000008	Auto	typedef ktime
121	ktime	00000008		union {s64 tv64;}
122	hrtimer_restart	00000004		enum : _int32 {...
123	hrtimer_clock_base	00000040	Auto	struct _attribute...
124	hrtimer_cpu_base	00000140	Auto	struct {raw_spinl...
125	raw_spinlock_t	00000004	Auto	typedef raw_spi...
126	timerqueue_head	00000010	Auto	struct {rb_root h...
127	rb_root	00000008	Auto	struct {rb_node ...
128	cpumask_t	00000040	Auto	typedef cpumask

Line 106 of 10238

```
int __fastcall tsk_fork_get_node(task_struct *tsk)
{
  int result; // eax

  _fentry__(tsk);
  result = -1;
  if ( kthreadd_task == tsk )
    return kthreadd_task->pref_node_fork;
  return result;
}
```

```
0009CA10 tsk_fork_get_node:1 (FFFFFFFF8109BA (Synchronized with IDA View-A, Hex
```

[1] https://en.wikipedia.org/wiki/DWARF
[2] http://ddebs.ubuntu.com/pool/main/l/linux/
[3] https://wiki.ubuntu.com/Kernel/Systemtap#How_do_I_build_a_debuginfo_kernel_if_one_isn.27t_available.3F
[4] https://stackoverflow.com/questions/12002315/extract-vmlinux-from-vmlinuz-or-bzimage
[5] https://github.com/marin-m/vmlinux-to-elf

#56: String literals in pseudocode

10 Sep 2021

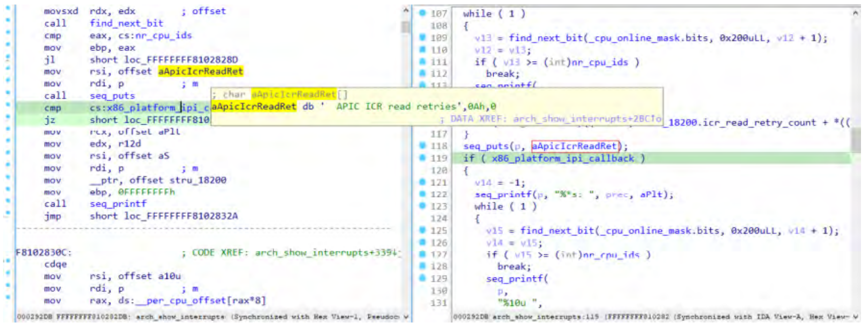
https://hex-rays.com/blog/igors-tip-of-the-week-56-string-literals-in-pseudocode/

Strings in binaries are very useful for the reverse engineer: they often contain messages shown to the user, or sometimes even internal debugging information (function or variable names) and so having them displayed in the decompiled code is very helpful.

However, sometimes you may see named variables in pseudocode even though the disassembly shows the string nicely. Why does this happen and how to fix it?

Memory access permissions

When deciding whether to display a string literal inline, the main criteria are attributes of the memory area it resides in. If the memory is writable, it means that the string is not really constant but may change, so displaying a variable name is more correct. For example, here's the default pseudocode of a function from a decompressed Linux kernel:

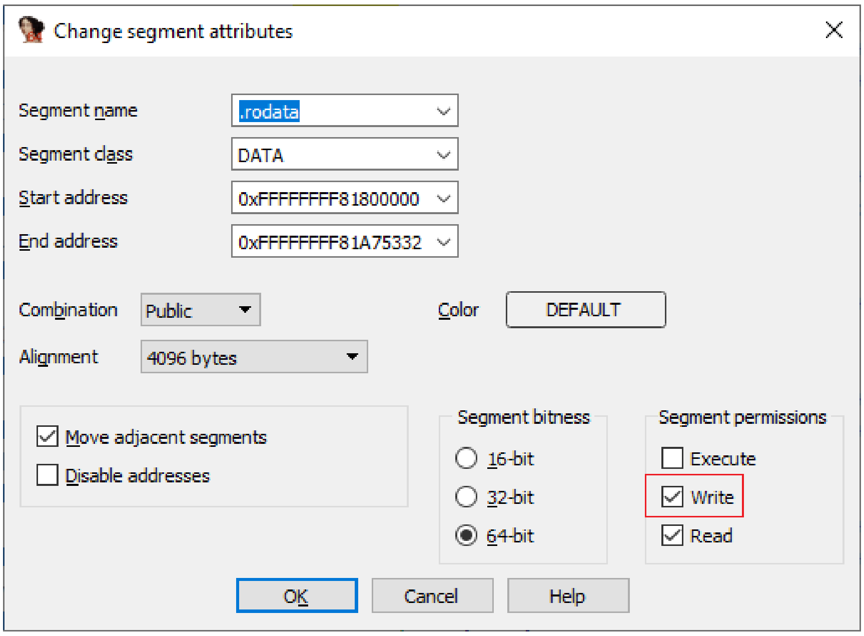


```
        movsxd  rdx, edx        ; offset
        call    find_next_bit
        cmp     eax, cs:nr_cpu_ids
        mov     ebp, eax
        jl      short loc_FFFFFFFF8102828D
        mov     rsi, offset aApicIcrReadRet
        mov     rdi, p          ; m
        call    seq_puts
        cmp     cs:x86_platform_ipi_c
        jz      short loc_FFFFFFFF810
        mov     rcx, offset aPlt
        mov     edx, r12d
        mov     rsi, offset aS
        mov     rdi, p          ; m
        mov     __ptr, offset stru_18200
        mov     ebp, 0FFFFFFFFh
        call    seq_printf
        jmp     short loc_FFFFFFFF8102832A
; ---------------------------------------------------------------------------

F8102830C:                      ; CODE XREF: arch_show_interrupts+339↓
        cdqe
        mov     rsi, offset a10u
        mov     rdi, p          ; m
        mov     rax, ds:__per_cpu_offset[rax*8]
000292DB FFFFFFFF810282DB: arch_show_interrupts (Synchronized with Hex View-1, Pseudoc
```

```
; char aApicIcrReadRet[]
aApicIcrReadRet db '  APIC ICR read retries',0Ah,0
                    ; DATA XREF: arch_show_interrupts+2BC↑o
```

```
while ( 1 )
{
  v13 = find_next_bit(_cpu_online_mask.bits, 0x200uLL, v12 + 1);
  v12 = v13;
  if ( v13 >= (int)nr_cpu_ids )
    break;
  seq_printf(
_18200.icr_read_retry_count + *((
}
seq_puts(p, aApicIcrReadRet);
if ( x86_platform_ipi_callback )
{
  v14 = -1;
  seq_printf(p, "%*s: ", prec, aPlt);
  while ( 1 )
  {
    v15 = find_next_bit(_cpu_online_mask.bits, 0x200uLL, v14 + 1);
    v14 = v15;
    if ( v15 >= (int)nr_cpu_ids )
      break;
    seq_printf(
      p,
      "%10u ",
000292DB arch_show_interrupts:115 (FFFFFFFF810282 (Synchronized with IDA View-A, Hex View-
```

We can see a string literal is displayed as a variable name (`aApicIcrReadRet`) even though it is a nice-looking string in the disassembly. The mystery can be cleared up if we jump to its definition (e.g. by double-clicking) and inspect the segment properties (Edit > Segment > Edit Segment..., or `Alt – S`). We can see that the segment is marked as writable:



Change segment attributes

Segment name: .rodata

Segment class: DATA

Start address: 0xFFFFFFFF81800000

End address: 0xFFFFFFFF81A75332

Combination: Public

Color: DEFAULT

Alignment: 4096 bytes

☑ Move adjacent segments

☐ Disable addresses

Segment bitness

○ 16-bit

○ 32-bit

● 64-bit

Segment permissions

☐ Execute

☑ Write

☑ Read

Why does `.rodata` ("read-only data") have write permissions? We can't say for sure, but the section does include this flag in the ELF headers:

(`readelf` output)

```
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000940
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         ffffff81000000  00001000
       0000000000628281  0000000000000000  AX       0     0     4096
  [ 2] .notes            NOTE             ffffff81628284  00629284
       0000000000000204  0000000000000000  AX       0     0     4
  [ 3] __ex_table        PROGBITS         ffffff81628490  00629488
```

#56: String literals in pseudocode

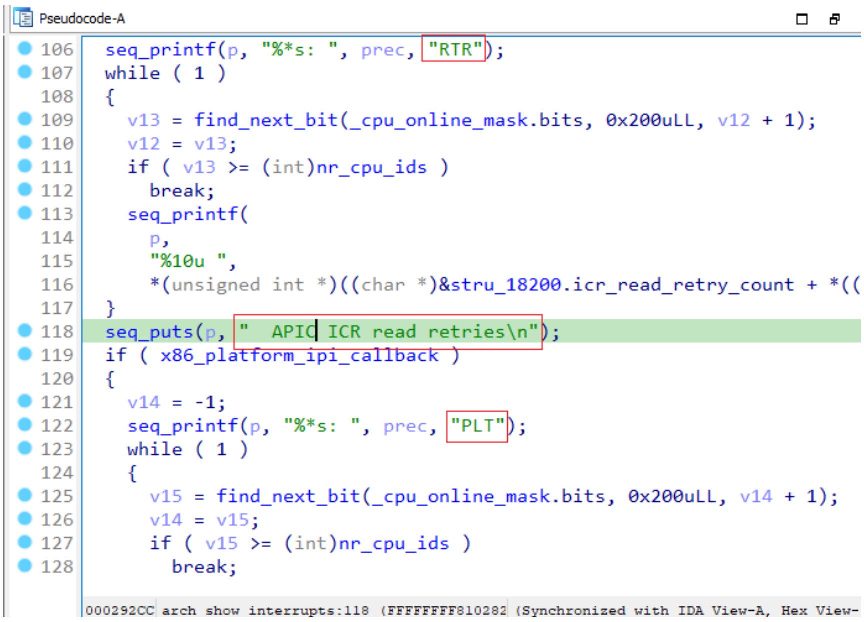
10 Sep 2021

https://hex-rays.com/blog/igors-tip-of-the-week-56-string-literals-in-pseudocode/

```
       0000000000002cdc  0000000000000000   A       0     0     4
  [ 4] .rodata           PROGBITS         ffffffff81800000  0062d000
       0000000000275332  0000000000000000  WA       0     0     4096

<...skipped...>

Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

One possibility is that it is made actually read-only later in the boot process.

So one solution for our problem is to make sure that the segment has only Read (and possibly Execute) permissions but not Write. If you do that, the string literals from that segment will be displayed inline:

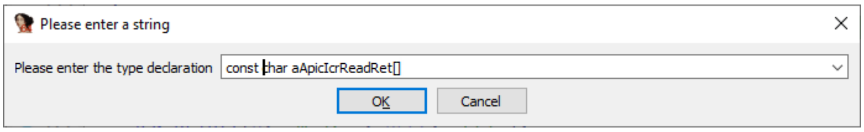


```
seq_printf(p, "%*s: ", prec, "RTR");
while ( 1 )
{
  v13 = find_next_bit(_cpu_online_mask.bits, 0x200uLL, v12 + 1);
  v12 = v13;
  if ( v13 >= (int)nr_cpu_ids )
    break;
  seq_printf(
    p,
    "%10u ",
    *(unsigned int *)((char *)&stru_18200.icr_read_retry_count + *((
}
seq_puts(p, "  APIC ICR read retries\n");
if ( x86_platform_ipi_callback )
{
  v14 = -1;
  seq_printf(p, "%*s: ", prec, "PLT");
  while ( 1 )
  {
    v15 = find_next_bit(_cpu_online_mask.bits, 0x200uLL, v14 + 1);
    v14 = v15;
    if ( v15 >= (int)nr_cpu_ids )
      break;
```

000292CC arch_show_interrupts:118 (FFFFFFFF810282 (Synchronized with IDA View-A, Hex View-

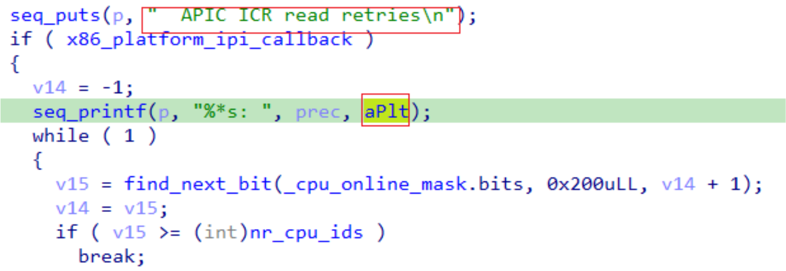
Override access permissions

While changing segment attributes works, it may not be suitable for all cases. For example, some compilers can put string constants in the same section as other writable data, so if you change the segment permissions to read-only, the decompiler could produce wrong output for functions using the writable variables. You may also have an opposite situation: a string constant is not actually constant but simply has a default value, so it needs to be marked as variable. In such cases, you can override the attributes of each string variable using `const` or `volatile` type attributes. For example, instead of changing the whole segment's permission, you could edit the type of the `aApicIcrReadRet` variable by pressing `Y` (change type) and changing its type to `const char aApicIcrReadRet[]`.



Please enter the type declaration: const char aApicIcrReadRet[]

With this option, only the edited strings literals will be shown inline and others remain as variables.



```
seq_puts(p, "  APIC ICR read retries\n");
if ( x86_platform_ipi_callback )
{
  v14 = -1;
  seq_printf(p, "%*s: ", prec, aPlt);
  while ( 1 )
  {
    v15 = find_next_bit(_cpu_online_mask.bits, 0x200uLL, v14 + 1);
    v14 = v15;
    if ( v15 >= (int)nr_cpu_ids )
      break;
```

#56: String literals in pseudocode

10 Sep 2021

https://hex-rays.com/blog/igors-tip-of-the-week-56-string-literals-in-pseudocode/

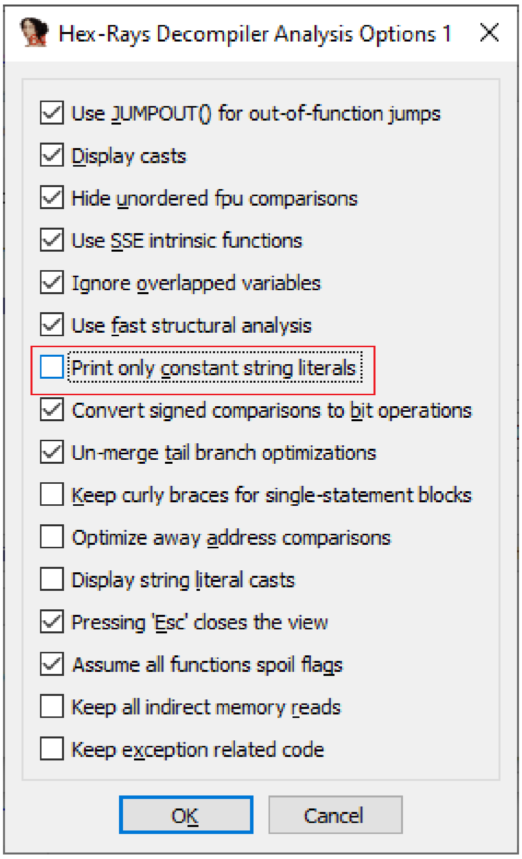
Show all string literals

Yet another possibility is to rely on IDA's analysis of disassembly and show all strings marked as string literals on the disassembly level. This can be done in the decompiler options (Edit > Plugins > Hex-Rays Decompiler, Options, Analysis Options 1) by turning off “Print only constant string literals” option.

To change this option for all future databases, see the `HO_CONST_STRINGS` option in `hexrays.cfg`.

For more info see the decompiler manual:

- Tips and tricks: Constant memory[1]
- Configuration[2]



Hex-Rays Decompiler Analysis Options 1

☑ Use JUMPOUT() for out-of-function jumps
☑ Display casts
☑ Hide unordered fpu comparisons
☑ Use SSE intrinsic functions
☑ Ignore overlapped variables
☑ Use fast structural analysis
□ Print only constant string literals
☑ Convert signed comparisons to bit operations
☑ Un-merge tail branch optimizations
□ Keep curly braces for single-statement blocks
□ Optimize away address comparisons
□ Display string literal casts
☑ Pressing 'Esc' closes the view
☑ Assume all functions spoil flags
□ Keep all indirect memory reads
□ Keep exception related code

[1] https://hex-rays.com/products/decompiler/manual/tricks.shtml#02
[2] https://hex-rays.com/products/decompiler/manual/config.shtml

#57: Shifted pointers 2

17 Sep 2021

https://hex-rays.com/blog/igors-tip-of-the-week-57-shifted-pointers-2/

This week we'll cover another situation where shifted pointers[1] can be useful.

Intrusive linked lists

This approach is used in many linked list implementations. Let's consider the one used in the Linux kernel. `list.h` defines the linked list structure:

```
struct list_head {
  struct list_head *next, *prev;
  };
```

As an example of its use, consider the struct `module` from `module.h`:

```
struct module {
    enum module_state state;

    /* Member of list of modules */
    struct list_head list;

    /* Unique handle for this module */
    char name[MODULE_NAME_LEN];

[..skipped..]
} ____cacheline_aligned __randomize_layout;
```

Where struct `list_head list;` is used to link the instances of `struct module` together. Because the `next` and `prev` pointers do not point to the start of `struct module`, some pointer math is required to access its fields. For this, various macros in list.h are used:

```
/**
 * list_entry - get the struct for this entry
 * @ptr:    the &struct list_head pointer.
 * @type:   the type of the struct this is embedded in.
 * @member: the name of the list_head within the struct.
 */
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
/**
 * list_first_entry - get the first element from a list
 * @ptr:    the list head to take the element from.
 * @type:   the type of the struct this is embedded in.
 * @member: the name of the list_head within the struct.
 *
 * Note, that list is expected to be not empty.
 */
#define list_first_entry(ptr, type, member) \
    list_entry((ptr)->next, type, member)
/**
 * list_last_entry - get the last element from a list
 * @ptr:    the list head to take the element from.
 * @type:   the type of the struct this is embedded in.
 * @member: the name of the list_head within the struct.
 *
 * Note, that list is expected to be not empty.
 */
#define list_last_entry(ptr, type, member) \
    list_entry((ptr)->prev, type, member)
```

Let's look at some functions from `module.c`. For example, `m_show()`:

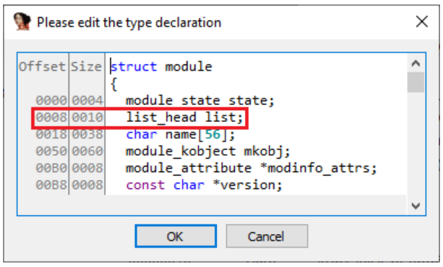
#57: Shifted pointers 2

17 Sep 2021

https://hex-rays.com/blog/igors-tip-of-the-week-57-shifted-pointers-2/

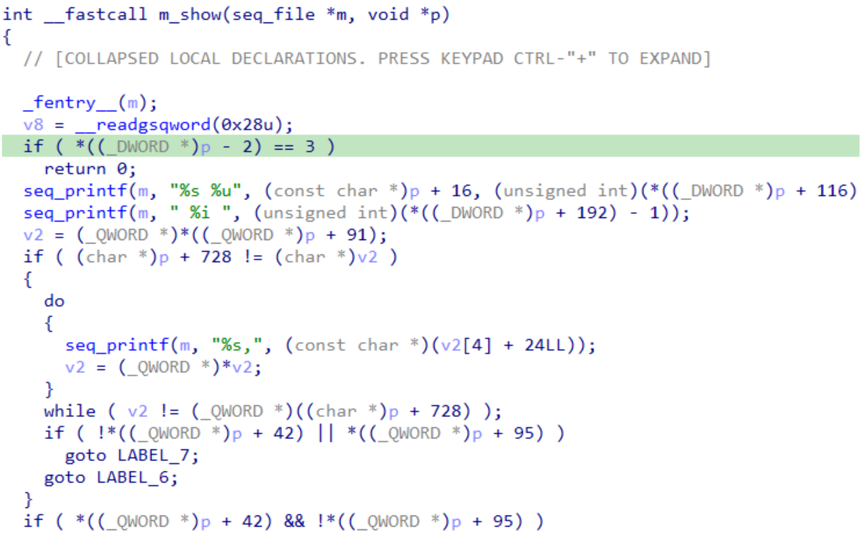
```
static int m_show(struct seq_file *m, void *p)
{
    struct module *mod = list_entry(p, struct module, list);
    char buf[MODULE_FLAGS_BUF_SIZE];
    void *value;

    /* We always ignore unformed modules. */
    if (mod->state == MODULE_STATE_UNFORMED)
            return º;

    seq_printf(m, “%s %u”,
               mod->name, mod->init_layout.size + mod->core_layout.size);
    print_unload_info(m, mod);

    /* Informative for users. */
    seq_printf(m, “ %s”,
               mod->state == MODULE_STATE_GOING ? “Unloading” :
               mod->state == MODULE_STATE_COMING ? “Loading” :
               “Live”);
    /* Used by oprofile and other similar tools. */
    value = m->private ? NULL : mod->core_layout.base;
    seq_printf(m, “ ºx%px”, value);

    /* Taints info */
    if (mod->taints)
            seq_printf(m, “ %s”, module_flags(mod, buf));

    seq_puts(m, “\n”);
    return º;
}
```

Although the function accepts a `void * p`, from the code we can see that it actually points to the list entry for the module at offset 8.



```
Offset|Size|struct module
          |    |{
  0000|0004|  module_state state;
  0008|0010|  list_head list;
  0018|0038|  char name[56];
  0050|0060|  module_kobject mkobj;
  00B0|0008|  module_attribute *modinfo_attrs;
  00B8|0008|  const char *version;
```

The initial decompilation looks like follows:



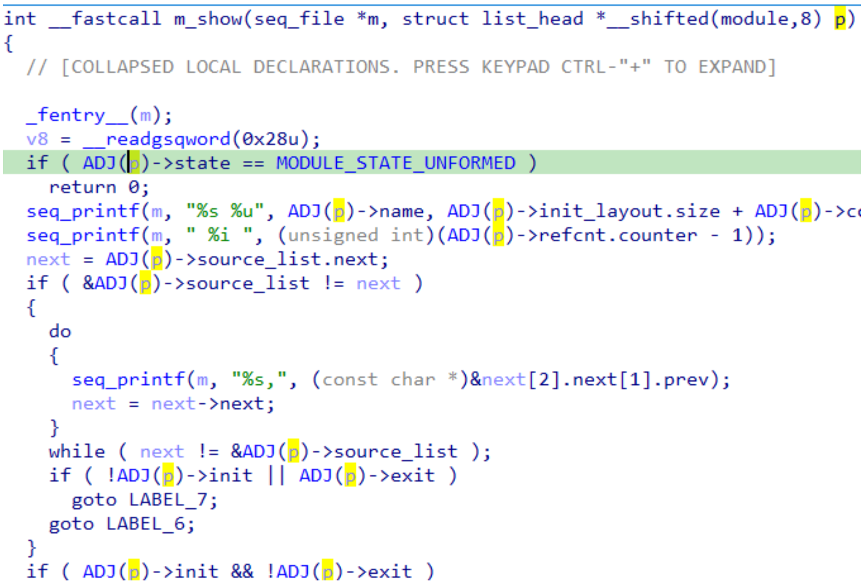
```
int __fastcall m_show(seq_file *m, void *p)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  _fentry__(m);
  v8 = __readgsqword(0x28u);
  if ( *((_DWORD *)p - 2) == 3 )
    return 0;
  seq_printf(m, "%s %u", (const char *)p + 16, (unsigned int)(*((_DWORD *)p + 116)
  seq_printf(m, " %i ", (unsigned int)(*((_DWORD *)p + 192) - 1));
  v2 = (_QWORD *)*((_QWORD *)p + 91);
  if ( (char *)p + 728 != (char *)v2 )
  {
    do
    {
      seq_printf(m, "%s,", (const char *)(v2[4] + 24LL));
      v2 = (_QWORD *)*v2;
    }
    while ( v2 != (_QWORD *)((char *)p + 728) );
    if ( !*((_QWORD *)p + 42) || *((_QWORD *)p + 95) )
      goto LABEL_7;
    goto LABEL_6;
  }
  if ( *((_QWORD *)p + 42) && !*((_QWORD *)p + 95) )
```

#57: Shifted pointers 2

17 Sep 2021

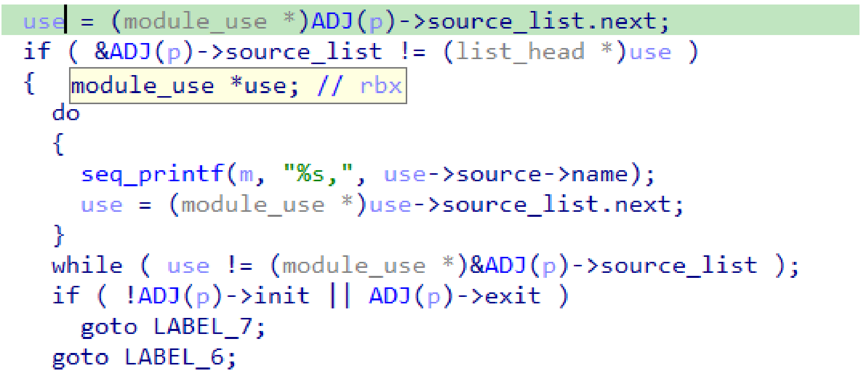
https://hex-rays.com/blog/igors-tip-of-the-week-57-shifted-pointers-2/

Not very readable, is it? But since we know that p actually points to `list` inside `struct module`, we can use a shifted pointer instead:



```
int __fastcall m_show(seq_file *m, struct list_head *__shifted(module,8) p)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  _fentry__(m);
  v8 = __readgsqword(0x28u);
  if ( ADJ(p)->state == MODULE_STATE_UNFORMED )
    return 0;
  seq_printf(m, "%s %u", ADJ(p)->name, ADJ(p)->init_layout.size + ADJ(p)->c
  seq_printf(m, " %i ", (unsigned int)(ADJ(p)->refcnt.counter - 1));
  next = ADJ(p)->source_list.next;
  if ( &ADJ(p)->source_list != next )
  {
    do
    {
      seq_printf(m, "%s,", (const char *)&next[2].next[1].prev);
      next = next->next;
    }
    while ( next != &ADJ(p)->source_list );
    if ( !ADJ(p)->init || ADJ(p)->exit )
      goto LABEL_7;
    goto LABEL_6;
  }
  if ( ADJ(p)->init && !ADJ(p)->exit )
```

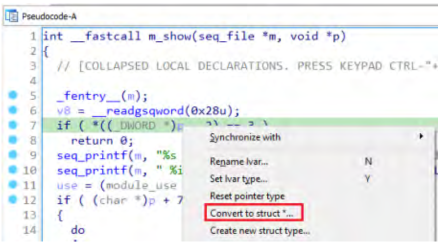
This is already much better. The ugly expression with the `next` variable is caused by the fact that `source_list` actually stores instances of `struct module_use` so by changing the variable's type we can improve the output again:



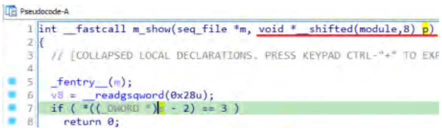
```
  use = (module_use *)ADJ(p)->source_list.next;
  if ( &ADJ(p)->source_list != (list_head *)use )
  {  module_use *use; // rbx
    do
    {
      seq_printf(m, "%s,", use->source->name);
      use = (module_use *)use->source_list.next;
    }
    while ( use != (module_use *)&ADJ(p)->source_list );
    if ( !ADJ(p)->init || ADJ(p)->exit )
      goto LABEL_7;
    goto LABEL_6;
```

Creating shifted pointers for structures

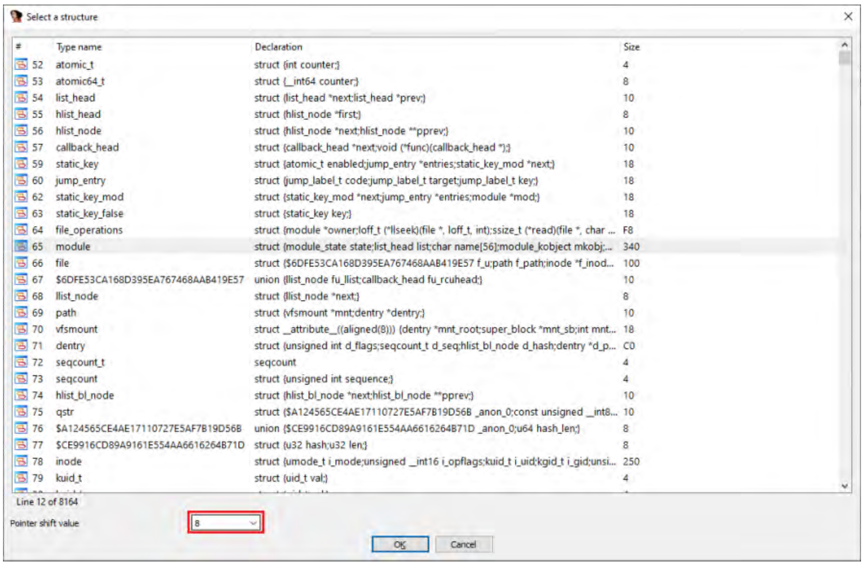
Although shifted pointers are not limited to structure members, it is the most common use case, and thus we implemented a UI feature to make their creation easier.



```
int __fastcall m_show(seq_file *m, void *p)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+

  _fentry__(m);
  v8 = __readgsqword(0x28u);
  if ( *((_DWORD *)p
    return 0;
  seq_printf(m, "%s
  seq_printf(m, " %i
  use = (module_use
  if ( (char *)p + 7
  {
    do
```



```
int __fastcall m_show(seq_file *m, void *__shifted(module,8) p)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXP

  _fentry__(m);
  v8 = __readgsqword(0x28u);
  if ( *((_DWORD *)p - 2) == 3 )
    return 0;
```



#	Type name	Declaration	Size
52	atomic_t	struct {int counter;}	4
53	atomic64_t	struct {_int64 counter;}	8
54	list_head	struct {list_head *next;list_head *prev;}	10
55	hlist_head	struct {hlist_node *first;}	8
56	hlist_node	struct {hlist_node *next;hlist_node **pprev;}	10
57	callback_head	struct {callback_head *next;void (*func)(callback_head *);}	10
59	static_key	struct {atomic_t enabled;jump_entry *entries;static_key_mod *next;}	18
60	jump_entry	struct {jump_label_t code;jump_label_t target;jump_label_t key;}	18
62	static_key_mod	struct {static_key_mod *next;jump_entry *entries;module *mod;}	18
63	static_key_false	struct {static_key key;}	18
64	file_operations	struct {module *owner;loff_t (*llseek)(file *, loff_t, int);ssize_t (*read)(file *, char ...	F8
65	module	struct {module_state state;list_head list;char name[56];module_kobject mkobj;...	340
66	file	struct {$6DFE53CA168D395EA767468AAB419E57 f_u;path f_path;inode *f_inod...	100
67	$6DFE53CA168D395EA767468AAB419E57	union {llist_node fu_llist;callback_head fu_rcuhead;}	10
68	llist_node	struct {llist_node *next;}	8
69	path	struct {vfsmount *mnt;dentry *dentry;}	10
70	vfsmount	struct __attribute__((aligned(8))) {dentry *mnt_root;super_block *mnt_sb;int mnt...	18
71	dentry	struct {unsigned int d_flags;seqcount_t d_seq;hlist_bl_node d_hash;dentry *d_p...	C0
72	seqcount_t	seqcount	4
73	seqcount	struct {unsigned int sequence;}	4
74	hlist_bl_node	struct {hlist_bl_node *next;hlist_bl_node **pprev;}	10
75	qstr	struct {$A124565CE4AE17110727E5AF7B19D56B _anon_0;const unsigned __int8...	10
76	$A124565CE4AE17110727E5AF7B19D56B	union {$CE9916CD89A9161E554AA6616264B71D _anon_0;u64 hash_len;}	8
77	$CE9916CD89A9161E554AA6616264B71D	struct {u32 hash;u32 len;}	8
78	inode	struct {umode_t i_mode;unsigned __int16 i_opflags;kuid_t i_uid;kgid_t i_gid;unsi...	250
79	kuid_t	struct {uid_t val;}	4

Line 12 of 8164

Pointer shift value: 8

#57: Shifted pointers 2

17 Sep 2021

https://hex-rays.com/blog/igors-tip-of-the-week-57-shifted-pointers-2/

In the decompiler, untyped variables and void pointers have a context menu item “Convert to struct *…”. When invoked, the dialog shows a list of structures (and unions) available in the local type library so you can easily create a pointer to it without typing manually. But in addition to simple struct pointers, you can create a shifted pointer by entering a non-zero delta value in the “Pointer shift value” field.

Because the original pointer had type `void *`, the shifted pointer retained it, so you may need to change the final type to get proper decompilation (in our example, `struct list_head *__shifted(module,8) p`).

If you want to practice this, here’s the 7.6 IDB with the function described: vmlinux_trimmed.elf.i64[2]. To save space, it’s been trimmed to only include the function in question and its direct dependencies. To get the full kernel with symbols, see the post on DWARF info loading[3].

`list_entry` https://elixir.bootlin.com/linux/latest/C/ident/list_entry
`list_first_entry` https://elixir.bootlin.com/linux/latest/C/ident/list_first_entry
`list_last_entry` https://elixir.bootlin.com/linux/latest/C/ident/list_last_entry
`member` https://elixir.bootlin.com/linux/latest/C/ident/member
`container_of` https://elixir.bootlin.com/linux/latest/C/ident/container_of

[1] https://hex-rays.com/blog/igors-tip-of-the-week-54-shifted-pointers/
[2] https://hex-rays.com/wp-content/uploads/2021/09/vmlinux_trimmed.elf_.i64.zip
[3] https://hex-rays.com/blog/igors-tip-of-the-week-57-shifted-pointers-2/

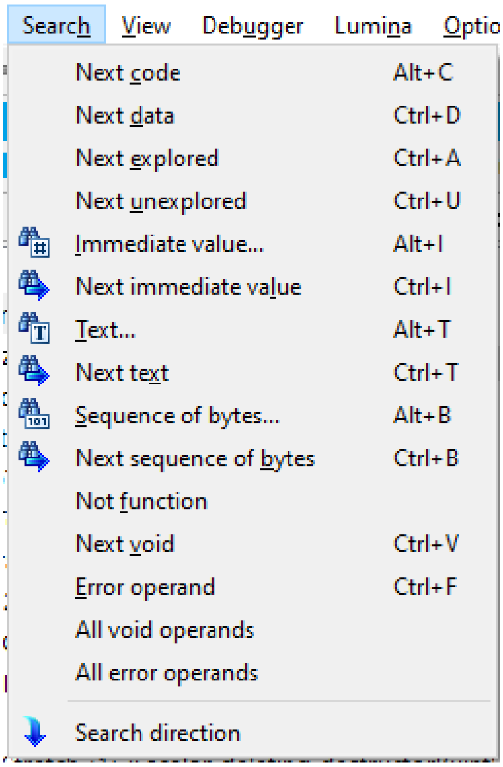
#58: Keyboard modifiers

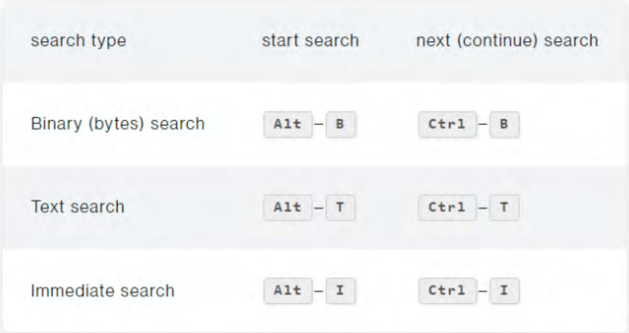
24 Sep 2021
https://hex-rays.com/blog/igors-tip-of-the-week-58-keyboard-modifiers/

Today we'll cover how keyboard modifiers (`Ctrl`, `Alt`, `Shift`) can be used with some IDA actions to modify their behavior or provide additional functionality.

Modifiers in shortcuts

Obviously, some shortcuts already include modifiers as part of their key sequence, but some commonalities may be not immediately obvious. For example, the Search menu commands tend to use `Alt`-letter to start search and corresponding `Ctrl`-letter to continue the search:





search type	start search	next (continue) search
Binary (bytes) search	Alt – B	Ctrl – B
Text search	Alt – T	Ctrl – T
Immediate search	Alt – I	Ctrl – I

A somewhat similar situation exists with data formatting shortcuts: same as `D` defines byte/word/dword items and `Alt–D` is used for extra item types and configuration, `A` creates a default string literal type while `Alt–A` handles additional ones and configuration.

Modifiers and the mouse

In some situations modifiers also change how mouse operations are interpreted:

- In the text IDA View, holding `Ctrl` while double-clicking a label or address opens the target in a new tab.
- Holding `Ctrl` or `Shift` while using the mouse wheel scrolls text tpage (like `PgDn`/`PgUp`). This also resizes hint popups[1] faster.
- In the graph view[2], `Ctrl` + wheel zooms the graph, while `Alt` + wheel scrolls horizontally (you can also use two-finger panning on trackpads).
- `Ctrl` + wheel also zooms in the navigation band[3].

Miscellaneous

- if one of the recent file entries in the File menu is selected while `Shift` key is held down, the file is opened in a new IDA instance.
- Windows only: if `Shift` key is held down while clicking the close window (X) corner button, IDA closes without confirmation with default exit options (save the database). If `Ctrl` is also held down, IDA exits without saving.

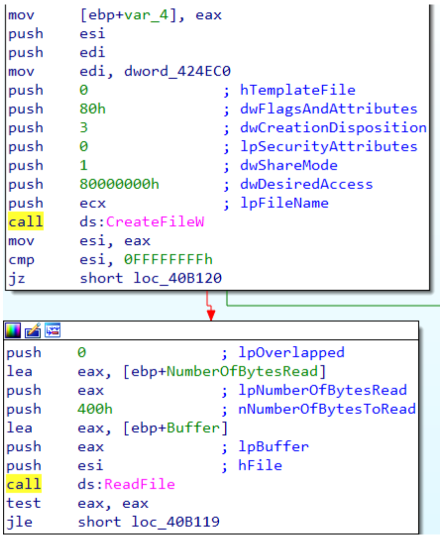
[1] https://hex-rays.com/blog/igors-tip-of-the-week-47-hints-in-ida/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-23-graph-view/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-49-navigation-band/

#59: Automatic function arguments comments

08 Oct 2021

https://hex-rays.com/blog/igors-tip-of-the-week-59-automatic-function-arguments-comments/

You may have observed that IDA knows about standard APIs or library functions and adds automatic function comments for the arguments passed to them.

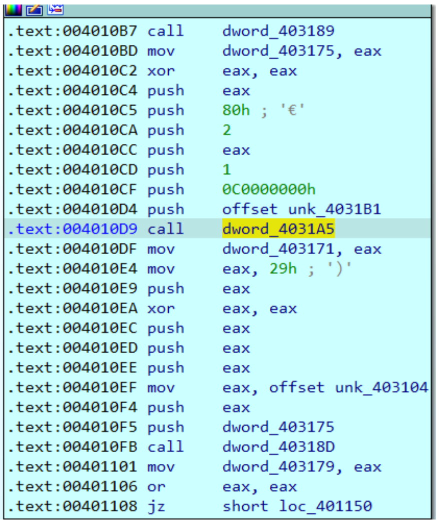


```
mov     [ebp+var_4], eax
push    esi
push    edi
mov     edi, dword_424EC0
push    0               ; hTemplateFile
push    80h             ; dwFlagsAndAttributes
push    3               ; dwCreationDisposition
push    0               ; lpSecurityAttributes
push    1               ; dwShareMode
push    80000000h       ; dwDesiredAccess
push    ecx             ; lpFileName
call    ds:CreateFileW
mov     esi, eax
cmp     esi, 0FFFFFFFFh
jz      short loc_40B120
```

```
push    0               ; lpOverlapped
lea     eax, [ebp+NumberOfBytesRead]
push    eax             ; lpNumberOfBytesRead
push    400h            ; nNumberOfBytesToRead
lea     eax, [ebp+Buffer]
push    eax             ; lpBuffer
push    esi             ; hFile
call    ds:ReadFile
test    eax, eax
jle     short loc_40B119
```

For example, here's a fragment of disassembly with commented arguments to Win32 APIs `CreateFileW` and `ReadFile`:

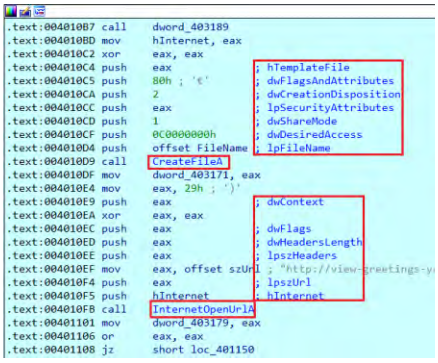
This works well when functions are imported in a standard way and are known at load time. However, there may be cases when the actual function is only known after analysis (e.g. imported dynamically using `GetProcAddress` or using a name hash). In that case, there may be only a call to some dummy name and no commented arguments:



```
.text:004010B7 call    dword_403189
.text:004010BD mov     dword_403175, eax
.text:004010C2 xor     eax, eax
.text:004010C4 push    eax
.text:004010C5 push    80h ; '€'
.text:004010CA push    2
.text:004010CC push    eax
.text:004010CD push    1
.text:004010CF push    0C0000000h
.text:004010D4 push    offset unk_4031B1
.text:004010D9 call    dword_4031A5
.text:004010DF mov     dword_403171, eax
.text:004010E4 mov     eax, 29h ; ')'
.text:004010E9 push    eax
.text:004010EA xor     eax, eax
.text:004010EC push    eax
.text:004010ED push    eax
.text:004010EE push    eax
.text:004010EF mov     eax, offset unk_403104
.text:004010F4 push    eax
.text:004010F5 push    dword_403175
.text:004010FB call    dword_40318D
.text:00401101 mov     dword_403179, eax
.text:00401106 or      eax, eax
.text:00401108 jz      short loc_401150
```

You can of course add a comment that `dword_4031A5` is `CreateFileA`, and comment arguments manually, but this can be quite tedious. Is there a way to do it automatically?

In fact, it is sufficient to simply rename the pointer variable to the corresponding API name for IDA to pick up the prototype and comment the arguments:



```
.text:004010B7 call    dword_403189
.text:004010BD mov     hInternet, eax
.text:004010C2 xor     eax, eax
.text:004010C4 push    eax             ; hTemplateFile
.text:004010C5 push    80h ; '€'       ; dwFlagsAndAttributes
.text:004010CA push    2               ; dwCreationDisposition
.text:004010CC push    eax             ; lpSecurityAttributes
.text:004010CD push    1               ; dwShareMode
.text:004010CF push    0C0000000h      ; dwDesiredAccess
.text:004010D4 push    offset FileName ; lpFileName
.text:004010D9 call    CreateFileA
.text:004010DF mov     dword_403171, eax
.text:004010E4 mov     eax, 29h ; ')'
.text:004010E9 push    eax             ; dwContext
.text:004010EA xor     eax, eax
.text:004010EC push    eax             ; dwFlags
.text:004010ED push    eax             ; dwHeadersLength
.text:004010EE push    eax             ; lpszHeaders
.text:004010EF mov     eax, offset szUrl ; "http://view-greetings-y
.text:004010F4 push    eax             ; lpszUrl
.text:004010F5 push    hInternet       ; hInternet
.text:004010FB call    InternetOpenUrlA
.text:00401101 mov     dword_403179, eax
.text:00401106 or      eax, eax
.text:00401108 jz      short loc_401150
```

A few notes about this feature:

1. The function prototype must be present in one of the loaded type libraries;
2. The comments are added only for code inside a function, so you may need to create one around the call (e.g. in case of decrypted or decompressed code);
3. if the function is called in many places, it may take a few seconds for IDA to analyze and comment all call sites.

#60: Type libraries

15 Oct 2021

https://hex-rays.com/blog/igors-tip-of-the-week-60-type-libraries/

Type libraries are collections of high-level type information for selected platforms and compilers which can be used by IDA and the decompiler.

A type library may contain:

1. function prototypes, e.g.:

```
void *__cdecl memcpy(void *, const void *Src, size_t Size);
BOOL __stdcall EnumWindows(WNDENUMPROC lpEnumFunc, LPARAM lParam);
```

2. typedefs, e.g.:

```
typedef unsigned long DWORD;
BOOL (__stdcall *WNDENUMPROC)(HWND, LPARAM);
```

3. standard structure and enum definitions, e.g.:

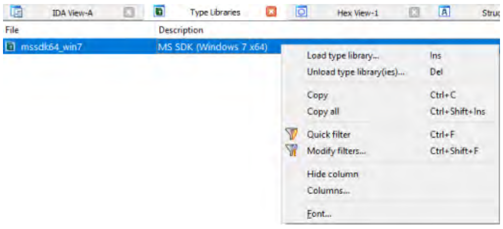
```
struct tagPOINT
{
 LONG x;
 LONG y;
};
enum tagSCRIPTGCTYPE
{
  SCRIPTGCTYPE_NORMAL = 0x0,
  SCRIPTGCTYPE_EXHAUSTIVE = 0x1,
};
```

4. Synthetic enums created from groups of preprocessor definitions (macros):

```
enum MACRO_WM
{
  WM_NULL = 0x0,
  WM_CREATE = 0x1,
  WM_DESTROY = 0x2,
  WM_MOVE = 0x3,
  WM_SIZEWAIT = 0x4,
  WM_SIZE = 0x5,
  WM_ACTIVATE = 0x6,
  WM_SETFOCUS = 0x7,
  WM_KILLFOCUS = 0x8,
  WM_SETVISIBLE = 0x9,
  [...]
 };
```

Manipulating type libraries

The list of currently loaded type libraries is available in the Type Libraries view (View > Open subiews > Type Libraries, or `Shift–F11`).



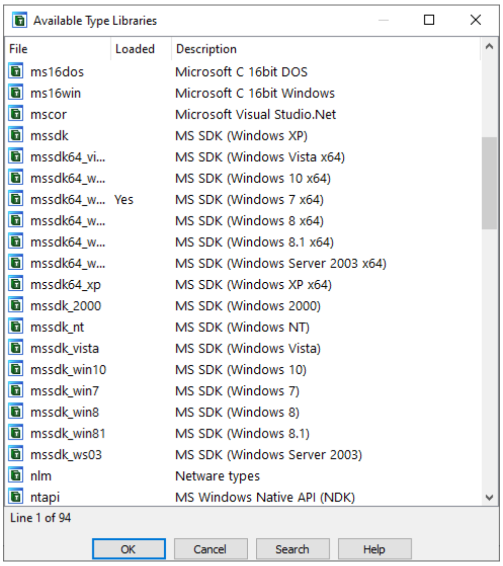
File	Description
mssdk64_win7	MS SDK (Windows 7 x64)

Additional libraries can be loaded using “Load type library...” context menu item or the `Ins` hotkey.

#60: Type libraries

15 Oct 2021

https://hex-rays.com/blog/igors-tip-of-the-week-60-type-libraries/



File	Loaded	Description
ms16dos		Microsoft C 16bit DOS
ms16win		Microsoft C 16bit Windows
mscor		Microsoft Visual Studio.Net
mssdk		MS SDK (Windows XP)
mssdk64_vi...		MS SDK (Windows Vista x64)
mssdk64_w...		MS SDK (Windows 10 x64)
mssdk64_w...	Yes	MS SDK (Windows 7 x64)
mssdk64_w...		MS SDK (Windows 8 x64)
mssdk64_w...		MS SDK (Windows 8.1 x64)
mssdk64_w...		MS SDK (Windows Server 2003 x64)
mssdk64_xp		MS SDK (Windows XP x64)
mssdk_2000		MS SDK (Windows 2000)
mssdk_nt		MS SDK (Windows NT)
mssdk_vista		MS SDK (Windows Vista)
mssdk_win10		MS SDK (Windows 10)
mssdk_win7		MS SDK (Windows 7)
mssdk_win8		MS SDK (Windows 8)
mssdk_win81		MS SDK (Windows 8.1)
mssdk_ws03		MS SDK (Windows Server 2003)
nlm		Netware types
ntapi		MS Windows Native API (NDK)

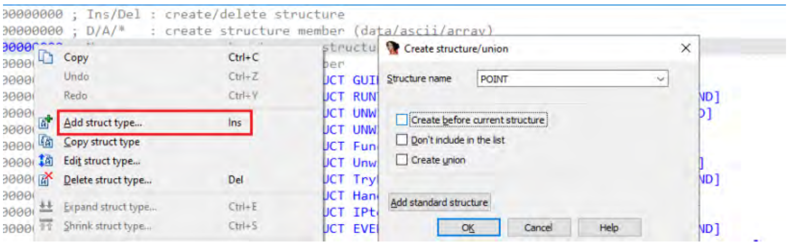
Line 1 of 94

Once loaded, definitions from the type library can be used in IDA and the decompiler: you can use them in function prototypes and global variable types (`Y` hotkey), as well as when adding new definitions in Local Types[1].

Importing types into IDB

While the decompiler can use types from loaded type libraries without extra work, to use them in the disassembly some additional action may be necessary. For example, to use a standard structure or enum, it has to be added to the list in the corresponding view first:

1. Open the Structures (`Shift– F9`) or Enums (`Shift– F10`) window;
2. Select “Add struct type..” or “Add enum” from the context menu, or use the hotkey (`Ins`);
3. If you know the struct/enum name, enter it in the name field and click OK;

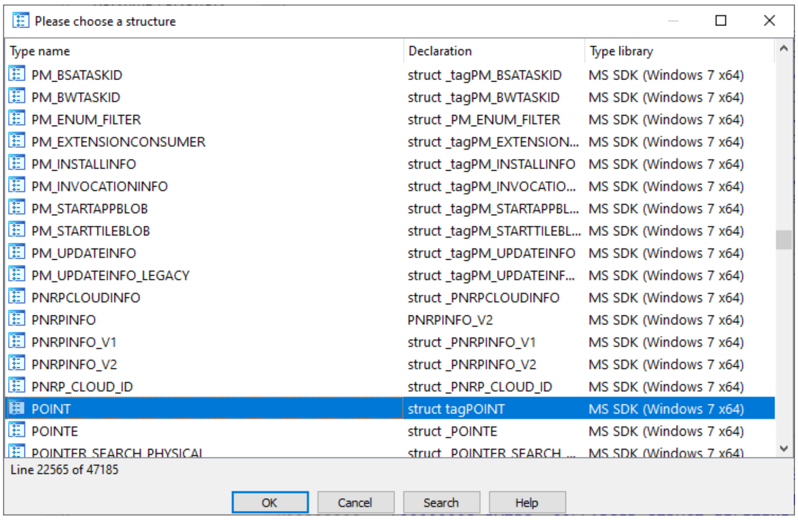


```
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/*   : create structure member (data/ascii/array)
```

Structure name: POINT

☐ Create before current structure
☐ Don't include in the list
☐ Create union

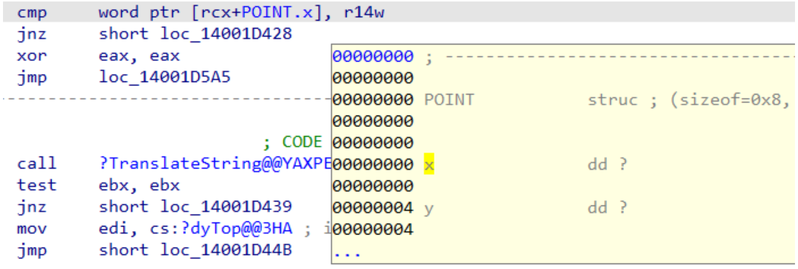
4. If you don’t know or remember the exact name, click “Add standard structure” (“Add standard enum”) and select the struct or enum from the list of all corresponding types in the loaded type libraries. As with all choosers[2], you can use incremental search or filtering (`Ctrl–F`).



Type name	Declaration	Type library
PM_BSATASKID	struct _tagPM_BSATASKID	MS SDK (Windows 7 x64)
PM_BWTASKID	struct _tagPM_BWTASKID	MS SDK (Windows 7 x64)
PM_ENUM_FILTER	struct _PM_ENUM_FILTER	MS SDK (Windows 7 x64)
PM_EXTENSIONCONSUMER	struct _tagPM_EXTENSION...	MS SDK (Windows 7 x64)
PM_INSTALLINFO	struct _tagPM_INSTALLINFO	MS SDK (Windows 7 x64)
PM_INVOCATIONINFO	struct _tagPM_INVOCATIO...	MS SDK (Windows 7 x64)
PM_STARTAPPBLOB	struct _tagPM_STARTAPPBL...	MS SDK (Windows 7 x64)
PM_STARTTILEBLOB	struct _tagPM_STARTTILEBL...	MS SDK (Windows 7 x64)
PM_UPDATEINFO	struct _tagPM_UPDATEINFO	MS SDK (Windows 7 x64)
PM_UPDATEINFO_LEGACY	struct _tagPM_UPDATEINF...	MS SDK (Windows 7 x64)
PNRPCLOUDINFO	struct _PNRPCLOUDINFO	MS SDK (Windows 7 x64)
PNRPINFO	PNRPINFO_V2	MS SDK (Windows 7 x64)
PNRPINFO_V1	struct _PNRPINFO_V1	MS SDK (Windows 7 x64)
PNRPINFO_V2	struct _PNRPINFO_V2	MS SDK (Windows 7 x64)
PNRP_CLOUD_ID	struct _PNRP_CLOUD_ID	MS SDK (Windows 7 x64)
POINT	struct tagPOINT	MS SDK (Windows 7 x64)
POINTE	struct _POINTE	MS SDK (Windows 7 x64)
POINTER_SEARCH_PHYSICAL	struct _POINTER_SEARCH_...	MS SDK (Windows 7 x64)

Line 22565 of 47185

After importing, the structure or enum can be used in the disassembly view.



```
cmp     word ptr [rcx+POINT.x], r14w
jnz     short loc_14001D428
xor     eax, eax                 00000000 ; ---------------------------------
jmp     loc_14001D5A5            00000000
-------------------------------- 00000000 POINT           struc ; (sizeof=0x8,
                                 00000000
                          ; CODE 00000000
call    ?TranslateString@@YAXPE00000000 x               dd ?
test    ebx, ebx                 00000000
jnz     short loc_14001D439      00000004 y               dd ?
mov     edi, cs:?dyTop@@3HA ; i00000004
jmp     short loc_14001D44B      ...
```

Function prototypes

When a type library is loaded, functions with name matching the prototypes present in the library will have their prototypes applied in the database. Alternatively, you can rename functions after loading the library, like we described last week[3].

[1] https://hex-rays.com/blog/igors-tip-of-the-week-36-working-with-list-views-in-ida/

[2] https://hex-rays.com/blog/igors-tip-of-the-week-^{36}-working-with-list-views-in-ida/

[3] https://hex-rays.com/blog/igors-tip-of-the-week-59-automatic-function-arguments-comments/

#61: Status bars

22 Oct 2021

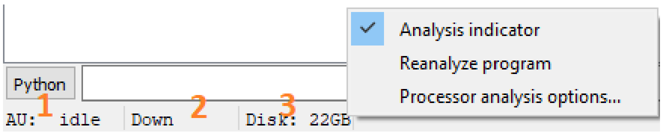
https://hex-rays.com/blog/igors-tip-of-the-week-61-status-bars/

Many of IDA's windows have status bars and they contain useful information and functionality which may not be always obvious.

Main window status bar

The status bar at the bottom of IDA's main window contains:

1. Autoanalysis progress indicator. See IDA Help: Analysis options[1] for possible values you may see there.
2. Search direction indicator for "Next search" commands[2] (`Ctrl`+Letter).
3. Free disk space.



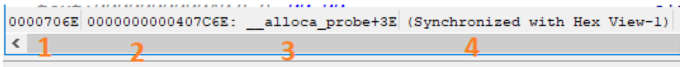
```
AU:  idle   Down    Disk: 22GB
```

It also has a context menu offering quick access to analysis and processor-specific options (if supported by current processor module).

Disassembly view status bar

Each disassembly (IDA View) windows has a separate status bar too. In the text mode it contains:

1. offset in the input file (for addresses which can be mapped directly to the input file);
2. address on the current cursor position
3. symbolic location (if available). for locations inside functions, a function name and offset from its start is printed;
4. synchronization status.



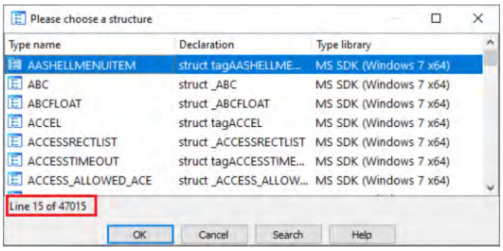
```
0000706E 0000000000407C6E: __alloca_probe+3E (Synchronized with Hex View-1)
```

Same status bar style is also used for Hex View and Pseudocode windows.

In the Graph mode[3], additional graph-related information is displayed (zoom level, mouse position etc.).

Chooser (list view) status bar

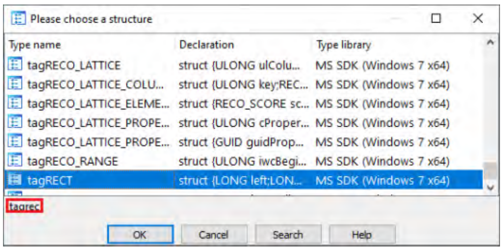
List views'[4] status bars by default display the current index and total number of items in the list



Type name	Declaration	Type library
AASHELLMENUITEM	struct tagAASHELLME...	MS SDK (Windows 7 x64)
ABC	struct _ABC	MS SDK (Windows 7 x64)
ABCFLOAT	struct _ABCFLOAT	MS SDK (Windows 7 x64)
ACCEL	struct tagACCEL	MS SDK (Windows 7 x64)
ACCESSRECTLIST	struct _ACCESSRECTLIST	MS SDK (Windows 7 x64)
ACCESSTIMEOUT	struct tagACCESSTIME...	MS SDK (Windows 7 x64)
ACCESS_ALLOWED_ACE	struct _ACCESS_ALLOW...	MS SDK (Windows 7 x64)

Line 15 of 47015

However, when using incremental search (type the first letters of the item to jump to the matching item), the typed letters replace it.



Type name	Declaration	Type library
tagRECO_LATTICE	struct {ULONG ulColu...	MS SDK (Windows 7 x64)
tagRECO_LATTICE_COLU...	struct {ULONG key;REC...	MS SDK (Windows 7 x64)
tagRECO_LATTICE_ELEME...	struct {RECO_SCORE sc...	MS SDK (Windows 7 x64)
tagRECO_LATTICE_PROPE...	struct {ULONG cProper...	MS SDK (Windows 7 x64)
tagRECO_LATTICE_PROPE...	struct {GUID guidProp...	MS SDK (Windows 7 x64)
tagRECO_RANGE	struct {ULONG iwcBegi...	MS SDK (Windows 7 x64)
tagRECT	struct {LONG left;LON...	MS SDK (Windows 7 x64)

tagrec

[1] https://hex-rays.com/products/ida/support/idadoc/620.shtml
[2] https://hex-rays.com/blog/igors-tip-of-the-week-58-keyboard-modifiers/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-23-graph-view/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-28-functions-list/

#62: Creating custom type libraries

29 Oct 2021

https://hex-rays.com/blog/igors-tip-of-the-week-62-creating-custom-type-libraries/

Previously we've talked about using type libraries[1] shipped with IDA, but what can be done when dealing with uncommon or custom APIs or SDKs not covered by them?

In such situation it is possible to use the **tilib** utility available for IDA Pro users from our download center[2].

Creating type libraries

tilib is a powerful command-line utility and the full list of options may look somewhat scary.

```
Type Information Library Utility v1.227 Copyright (c) 2000-2021 Hex-Rays
usage: tilib [-sw] til-file
  -c     create til-file             -t...  set til-file title
  -h...  parse .h file               -P     C++ mode (not ready yet)
  -D...  define a symbol             -I...  list of include dirs
  -M...  create macro defs file      -x     external display types
  -i     internal display types       -z     debug .h file parsing (use it!)
  -B...  dump bad macro defs          -q     internal check: unpack types
  -C...  compiler info(-C? help)      -G...  mangling format (n=org.name)
  -m...  parse macro defs file       -S     strip macro table
  -dt... delete type definition      -rtX:Y rename type X as Y
  -ds... delete symbol definition    -rsX:Y rename symbol X as Y
  -b...  use base til                 -o...  directory with til files
  -l[1csxf] show til-file contents; 1-with decorated names, c-as c code
           s-dump struct layout, x-exercise udt serialization, f-dump funcarg locations
  -v     verbose                      -e     ignore errors
  -R     allow redeclarations         -n     ignore til macro table
  -u+    uncompress til-file         -u-    compress til-tile
  -U     set ‘universal til’ bit      -em    suppress macro creation errors
  -#     enable ordinal types         -#-    disable ordinal types
  -p...  load types from PDB (Win32)  -TL    lower existing type
  -TAL   assume low level types       -TH    keep high types
  -g[nb]X:Y move macro X (regex) to group Y; n-name, b-body
   @...  response file with switches
example: tilib -c -Cc1 -hstdio.h stdio.til
```

However, as mentioned at the botttom, the basic usage can be quite simple:

```
tilib -c -Cc1 -hstdio.h stdio.til
```

This creates a type library `stdio.til` by parsing the header file **`stdio.h`** as a Visual C++ compiler.

Advanced options

The sample commandline might work in simple cases (e.g. a single, self-contained header) but with real life SDKs you will likely run into problems quickly. To handle them, additional options may be necessary:

1. Include directories for headers from **#include** directives: **-I<directory>** (can be specified multiple times);
2. preprocessor defines: **-Dname[=value]**;

Instead of using `-D` on command line, you can also create a new header with **#define** statements and include other headers from it.

Response files

To avoid specifying the same options again and again, you can use response files. These files contain one command line option per line and can be passed to tilib using the @ option:

```
tilib @vc32.cfg -c -hinput.h output.til
```

There are sample response files shipped with the tilib package for Visual C++ (32- and 64-bit), GCC and Borland C++.

Examining type libraries

You can dump the contents of a til file using the **-l** switch:

```
tilib -l mylib.til
```

Using created type libraries in IDA

To make the custom type library available in IDA, copy it in the `til/<processor>` subdirectory of IDA. For example, libraries for x86/x64 files should go under `til/pc`. After this, the new library should appear in the list shown when you invoke the “Load type library” command.

Advanced example

One of our users made a very nice write-up on generating a type library for Apache modules. Please find it here: https://github.com/trou/apache-module-ida-til.

See also readme.txt in the tilib package for advanced usage such as creating enums from groups of preprocessor macro definitions.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-60-type-libraries/
[2] https://hex-rays.com/download-center/

#63: IDA installer command-line options

05 Nov 2021
https://hex-rays.com/blog/igors-tip-of-the-week-63-ida-installer-command-line-options/

Most users probably run IDA installers in standard, interactive mode. However, they also can be run in unattended mode (e.g. for automatic, non-interactive installation).

Available options

To get the list of available options, run the installer with the --help argument. For example, here's the list on Linux:

```
igor@/home/igor$ ./idapronl[...].run --help
IDA Pro and Hex-Rays Decompilers (x86, x64, ARM, ARM64, PPC, PPC64, MIPS) 7.6 7.6
Usage:

 --help                                      Display the list of valid options

 --version                                   Display product information

 --unattendedmodeui <unattendedmodeui>     Unattended Mode UI
                                             Default: none
                                             Allowed: none minimal minimalWithDialogs

 --optionfile <optionfile>                Installation option file
                                             Default:

 --debuglevel <debuglevel>                   Debug information level of verbosity
                                             Default: 2
                                             Allowed: 0 1 2 3 4

 --mode <mode>                               Installation mode
                                             Default: gtk
                                             Allowed: gtk xwindow text unattended

 --debugtrace <debugtrace>                   Debug filename
                                             Default:

 --installer-language <installer-language>   Language selection
                                             Default: en
                                             Allowed: sq ar es_AR az eu pt_BR bg ca hr cs da nl en et fi fr
de el he hu id it ja kk ko lv lt no fa pl pt ro ru sr zh_CN sk sl es sv th zh_TW tr tk uk va vi cy

 --prefix <prefix>                         Installation Directory
                                             Default: /home/igor/idapro-7.6

 --python_version <python_version>           IDAPython Version
                                             Default: 3
                                             Allowed: 2 3

 --installpassword <installpassword>         Installation password
                                             Default:
```

For example, to run the installer in text (console) instead of GUI mode, specify `--mode text`.

```
On Windows, the set of options is slightly different (and is shown in a GUI dialog instead of console):

IDA Pro and Hex-Rays Decompilers (x86, x64, ARM, ARM64, PPC, PPC64, MIPS) 7.6 7.6
Usage:

 --help                                      Display the list of valid options

 --version                                   Display product information

 --unattendedmodeui <unattendedmodeui>       Unattended Mode UI
                                             Default: none
                                             Allowed: none minimal minimalWithDialogs

 --optionfile <optionfile>                 Installation option file
                                             Default:
```

#63: IDA installer command-line options

05 Nov 2021

https://hex-rays.com/blog/igors-tip-of-the-week-63-ida-installer-command-line-options/

```
 --debuglevel <debuglevel>                    Debug information level of verbosity
                                              Default: 2
                                              Allowed: 0 1 2 3 4

 --mode <mode>                                Installation mode
                                              Default: win32
                                              Allowed: win32 unattended

 --debugtrace <debugtrace>                    Debug filename
                                              Default:

 --installer-language <installer-language>    Language selection
                                              Default: en
                                              Allowed: sq ar es_AR az eu pt_BR bg ca hr cs da nl en et fi fr
de el he hu id it ja kk ko lv lt no fa pl pt ro ru sr zh_CN sk sl es sv th zh_TW tr tk uk va vi cy

 --prefix <prefix>                          Installation Directory
                                              Default: C:\Program Files/idapro-7.6

 --python_version <python_version>            IDAPython Version
                                              Default: 3
                                              Allowed: 2 3

 --installpassword <installpassword>          Installation password
                                              Default:

 --install_python <install_python>            Install Python 3
                                              Default:
```

In partcular, `--install_python` option allows to enable installation of the bundled Python 3 (useful for machines without Python preinstalled). On Linux and Mac, the system-wide Python is presumed to be available.

Using the option file

Especially for unattended mode, you may need to specify multiple options (install path, Python version, installation password etc.). Instead of passing them all on the command line, you can use the option file:

1. Create a simple text file with a list of option=value lines. The option names are those from the usage screen without the leading --.
2. Pass the filename to the installer using the --optionfile <filename> switch.

For example, here's a file for unattended install for Python 3:

```
installpassword=mypassword
prefix=/home/igor/ida-7.6
mode=unattended
python_version=3
```

Running Mac installer from command line

Because the Mac installer is not a single binary but an app bundle, you need to pass arguments to the executable inside the bundle, for example:

```
ida[...].app/Contents/MacOS/installbuilder.sh --mode text
```

Uninstaller options

The uninstaller can also be run with commandline opions:

```
~/idapro-7.6/uninstall --mode unattended
```

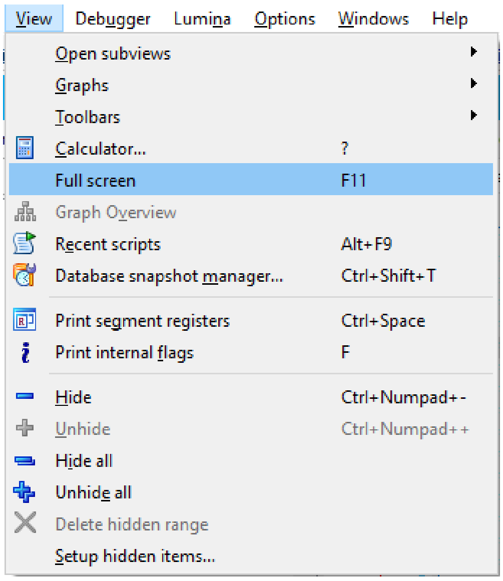
#64: Full-screen mode

12 Nov 2021

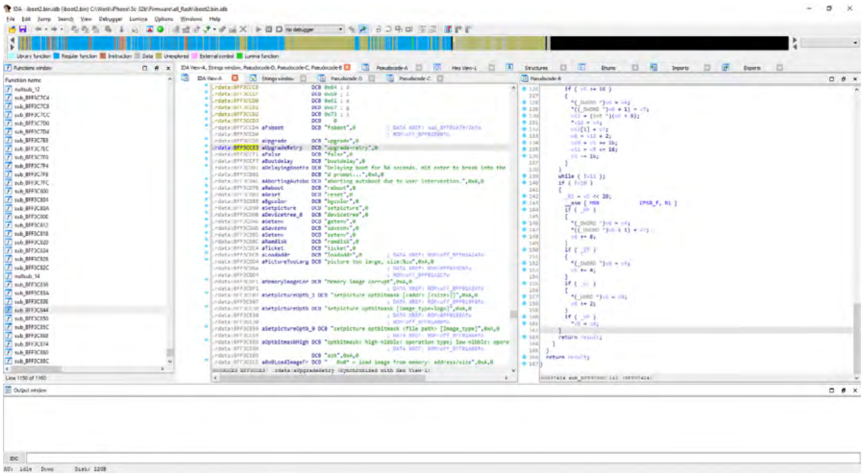
https://hex-rays.com/blog/igors-tip-of-the-week-64-full-screen-mode/

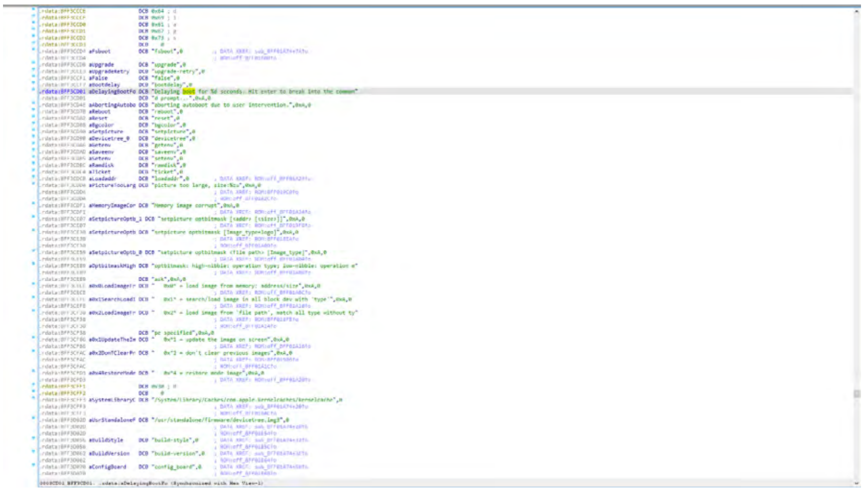
While not commonly used, full-screen mode can be useful on complex IDA layouts when working with a single monitor or on a laptop, for example when you need to read a long listing line but are tired of scrolling around.

The feature is somewhat hidden, but the action is present in the View menu.



By pressing `F11`, the current view is temporarily expanded to full screen, and all other views and UI elements (toolbars, menus) are hidden to remove all distractions.





To go back to the standard layout, just press `F11` again.

#65: Stack frame view

19 Nov 2021

https://hex-rays.com/blog/igors-tip-of-the-week-65-stack-frame-view/

The *stack frame* is part of the stack which is managed by the current function and contains the data used by it.

Background

The stack frame usually contains data such as:

- local and temporary variables;
- incoming arguments (for calling conventions which use stack for passing arguments);
- saved volatile registers;
- other bookkeeping information (e.g. the return address on x86).

Because the stack may change unpredictably during execution, the stack frame and its parts do not have a fixed address. Thus, IDA uses a pseudo structure to represent its layout. This structure is very similar to other structures in the Structures view, with a few differences:

1. The frame structure has no name and is not included in the global Structures list; it can only be reached from the corresponding function;
2. Instead of offsets from the structure start, offsets from the frame pointer are shown (both positive and negative);
3. It may contain special members to represent the saved return address and/or saved register area.

Stack frame view

To open the stack frame view:

- Edit > Functions > Stack variables... or press `Ctrl–K` while positioned in a function in disassembly (IDA View);
- Double-click or press `Enter` on a stack variable in the disassembly or pseudocode.

In this view, you can perform most of the same operations as in the Structures view:

1. Define new or change existing stack variables (`D`);
2. Rename variables (`N`);
3. Create arrays[1] (`*`) or structure instances (`Alt– Q`).

Example

Consider this vulnerable program:

```
#include <stdio.h>
int main () {
    char username[8];
    int allow = 0;
    printf external link(“Enter your username, please: “);
    gets(username); // user inputs “malicious”
    if (grantAccess(username)) {
        allow = 1;
    }
    if (allow != 0) { // has been overwritten by the overflow of the username.
        privilegedAction();
    }
    return 0;
}
```

Source: CERN Computer Security[2]

When compiled by an old GCC version, it might produce the following assembly:

```
.text:0000000000400580 main proc near                          ; DATA XREF: _start+1D↑o
.text:0000000000400580
.text:0000000000400580 var_10= byte ptr -10h
.text:0000000000400580 var_4= dword ptr -4
.text:0000000000400580
.text:0000000000400580 ; __unwind {
.text:0000000000400580     push    rbp
.text:0000000000400581     mov     rbp, rsp
.text:0000000000400584     sub     rsp, 10h
.text:0000000000400588     mov     [rbp+var_4], 0
.text:000000000040058F     mov     edi, offset format          ; “Enter your username, please: “
```

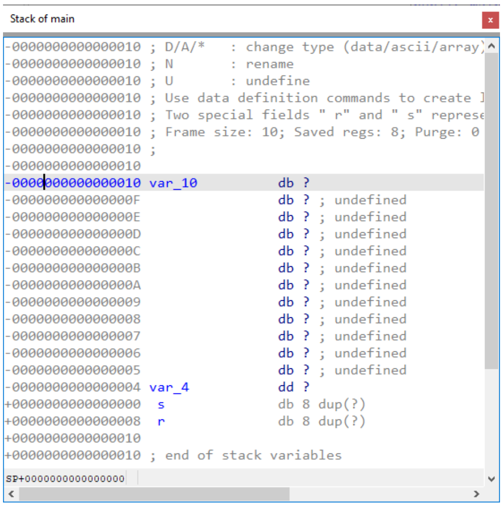
#65: Stack frame view

19 Nov 2021

https://hex-rays.com/blog/igors-tip-of-the-week-65-stack-frame-view/

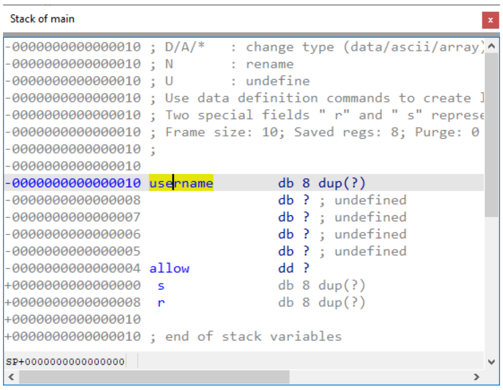
```
.text:0000000000400594     mov     eax, 0
.text:0000000000400599     call    _printf
.text:000000000040059E     lea     rax, [rbp+var_10]
.text:00000000004005A2     mov     rdi, rax
.text:00000000004005A5     call    _gets
.text:00000000004005AA     lea     rax, [rbp+var_10]
.text:00000000004005AE     mov     rdi, rax
.text:00000000004005B1     call    grantAccess
.text:00000000004005B6     test    eax, eax
.text:00000000004005B8     jz      short loc_4005C1
.text:00000000004005BA     mov     [rbp+var_4], 1
.text:00000000004005C1
.text:00000000004005C1 loc_4005C1:                            ; CODE XREF: main+38↑j
.text:00000000004005C1     cmp     [rbp+var_4], 0
.text:00000000004005C5     jz      short loc_4005D1
.text:00000000004005C7     mov     eax, 0
.text:00000000004005CC     call    privilegedAction
.text:00000000004005D1
.text:00000000004005D1 loc_4005D1:                            ; CODE XREF: main+45↑j
.text:00000000004005D1     mov     eax, 0
.text:00000000004005D6     leave
.text:00000000004005D7     retn
.text:00000000004005D7 ; } // starts at 400580
.text:00000000004005D7 main endp
```

On opening the stack frame we can see the following picture:



```
-0000000000000010 ; D/A/*   : change type (data/ascii/array)
-0000000000000010 ; N       : rename
-0000000000000010 ; U       : undefine
-0000000000000010 ; Use data definition commands to create
-0000000000000010 ; Two special fields " r" and " s" represe
-0000000000000010 ; Frame size: 10; Saved regs: 8; Purge: 0
-0000000000000010 ;
-0000000000000010
-0000000000000010 var_10          db ?
-000000000000000F                 db ? ; undefined
-000000000000000E                 db ? ; undefined
-000000000000000D                 db ? ; undefined
-000000000000000C                 db ? ; undefined
-000000000000000B                 db ? ; undefined
-000000000000000A                 db ? ; undefined
-0000000000000009                 db ? ; undefined
-0000000000000008                 db ? ; undefined
-0000000000000007                 db ? ; undefined
-0000000000000006                 db ? ; undefined
-0000000000000005                 db ? ; undefined
-0000000000000004 var_4           dd ?
+0000000000000000  s              db 8 dup(?)
+0000000000000008  r              db 8 dup(?)
+0000000000000010
+0000000000000010 ; end of stack variables
SP+0000000000000000
```

By comparing the source code and disassembly, we can infer that `var_10` is `username` and `var_4` is `allow`. Because the code only takes the address of start of the buffer, IDA could not detect its full size and created a single byte variable. To improve it, press * on `var_10` and convert it into an array of 8 bytes. We can also rename the variables to their proper names.



```
-0000000000000010 ; D/A/*   : change type (data/ascii/array)
-0000000000000010 ; N       : rename
-0000000000000010 ; U       : undefine
-0000000000000010 ; Use data definition commands to create
-0000000000000010 ; Two special fields " r" and " s" represe
-0000000000000010 ; Frame size: 10; Saved regs: 8; Purge: 0
-0000000000000010 ;
-0000000000000010
-0000000000000010 username        db 8 dup(?)
-0000000000000008                 db ? ; undefined
-0000000000000007                 db ? ; undefined
-0000000000000006                 db ? ; undefined
-0000000000000005                 db ? ; undefined
-0000000000000004 allow           dd ?
+0000000000000000  s              db 8 dup(?)
+0000000000000008  r              db 8 dup(?)
+0000000000000010
+0000000000000010 ; end of stack variables
SP+0000000000000000
```

#65: Stack frame view

19 Nov 2021

https://hex-rays.com/blog/igors-tip-of-the-week-65-stack-frame-view/

Because IDA shows the stack frame layout in the natural memory order (addresses increase towards the bottom), we can immediately see the problem demonstrated by the vulnerable code: the `gets` function has no bounds checking, so entering a long string can overflow the `username` buffer and overwrite the `allow` variable. Since the code is only checking for a non-zero value, this will bypass the check and result in the execution of the `privilegedAction` function.

Frame offsets and stack variables

As mentioned above, in the stack frame view structure offsets are shown relative to the frame pointer. In some cases, like in the example above, it is an actual processor register (`RBP`). For example, the variable `allow` is placed at offset `-4` from the frame pointer and this value is used by IDA in the disassembly listing for the symbolic name instead of raw numerical offset:

```
.text:0000000000400580 allow= dword ptr -4
[...]
.text:0000000000400588 mov [rbp+allow], 0
[...]
```

By pressing `#` or `K` on the instruction, you can ask IDA to show you the instruction's original form:

```
.text:0000000000400588 mov dword ptr [rbp-4], 0
```

Press `K` again to get back to the stack variable representation.

In other situations the frame pointer can be just an arbitrary location used for convenience (usually a fixed offset from the stack pointer value at function entry). This is common in binaries compiled with frame pointer omission, a common optimization technique. In such situation, IDA may use an extra delta to compensate for the stack pointer changes in different parts of function. For example, consider this function:

```
.text:10001030 sub_10001030 proc near                  ; DATA XREF: sub_100010B0:loc_100010E7↓o
.text:10001030
.text:10001030 LCData= byte ptr -0Ch
.text:10001030 var_4= dword ptr -4
.text:10001030
.text:10001030    sub     esp, 0Ch
.text:10001033    mov     eax, dword_100B2960
.text:10001038    push    esi
.text:10001039    mov     [esp+10h+var_4], eax
.text:1000103D    xor     esi, esi
.text:1000103F    call    ds:GetThreadLocale
.text:10001045    push    7                           ; cchData
.text:10001047    lea     ecx, [esp+14h+LCData]
.text:1000104B    push    ecx                         ; lpLCData
.text:1000104C    push    1004h                       ; LCType
.text:10001051    push    eax                         ; Locale
.text:10001052    call    ds:GetLocaleInfoA
.text:10001058    test    eax, eax
.text:1000105A    jz      short loc_1000107D
.text:1000105C    mov     al, [esp+10h+LCData]
.text:10001060    test    al, al
.text:10001062    lea     ecx, [esp+10h+LCData]
.text:10001066    jz      short loc_1000107D
```

Here, the explicit frame pointer (ebp) is not used, and IDA arranges the stack frame so that the return address is placed offset 0:

```
-00000010 ; Frame size: 10; Saved regs: 0; Purge: 0
-00000010 ;
-00000010
-00000010     db ? ; undefined
-0000000F     db ? ; undefined
-0000000E     db ? ; undefined
-0000000D     db ? ; undefined
-0000000C LCData db ?
-0000000B     db ? ; undefined
-0000000A     db ? ; undefined
```

```
-00000009     db ? ; undefined
-00000008     db ? ; undefined
-00000007     db ? ; undefined
-00000006     db ? ; undefined
-00000005     db ? ; undefined
-00000004 var_4 dd ?
+00000000  r  db 4 dup(?)
+00000004
+00000004 ; end of stack variables
```

To compensate for the changes of the stack pointer (`sub esp, 0Ch` and the push instructions), values `10h` or `14h` have to be added in the stack variable operands. Thanks to this, we can easily see that instructions at `10001047` and `1000105C` refer to the same variable, even though in raw form they use different offsets (`[esp+8] and [esp+4]`).

Extra information: IDA Help: Stack Variables Window[3]

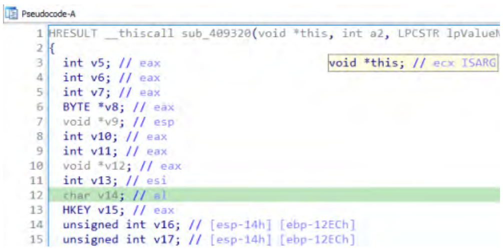
[1] https://hex-rays.com/blog/igor-tip-of-the-week-10-working-with-arrays/
[2] https://security.web.cern.ch/rec ommendations/en/codetools/c.shtml
[3] https://www.hex-rays.com/products/ida/support/idadoc/488.shtml

#66: Decompiler annotations

26 Nov 2021

https://hex-rays.com/blog/igors-tip-of-the-week-66-decompiler-annotations/

When working with pseudocode in the decompiler, you may have noticed that variable declarations and hints have comments with somewhat cryptic contents. What do they mean?



```
HRESULT __thiscall sub_409320(void *this, int a2, LPCSTR lpValueN
{
  int v5; // eax                                void *this; // ecx ISARG
  int v6; // eax
  int v7; // eax
  BYTE *v8; // eax
  void *v9; // esp
  int v10; // eax
  int v11; // eax
  void *v12; // eax
  int v13; // esi
  char v14; // al
  HKEY v15; // eax
  unsigned int v16; // [esp-14h] [ebp-12ECh]
  unsigned int v17; // [esp-14h] [ebp-12ECh]
```

While meaning of some may be obvious, others less so, and a few appear only in rare situations.

Variable location

The fist part of the comment is the variable location. For stack variables, this includes its location relative to the stack and frame pointers. For register variables – the register(s) used for storing its value.

In some cases, you may also see the scattered argloc[1] syntax. For example:

```
struct12 v78; // 0:r2.8,8:^0.4
```

This denotes a 12-byte structure stored partially in registers (8 first bytes, beginning at `r2`), and on stack (4 last bytes, starting from stack offset 8).

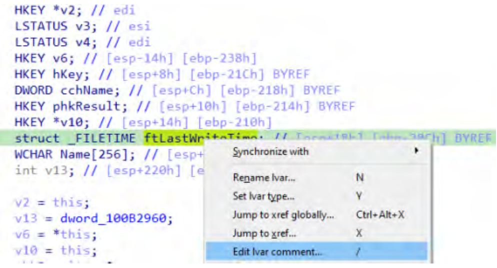
Variable attributes

After the location, there may be additional attributes printed as uppercase keywords. Here are the most common possibilities:

- `BYREF`: address of this variable is taken somewhere in the current function (e.g. for passing to a function call);
- `OVERLAPPED`: shown when the decompiler did not manage to separate all the variables so some of them ended up being stored in intersecting locations. Usually functions with such variables are also marked with the comment: `// local variable allocation has failed, the output may be wrong!`
- `MAPDST`: another variable has been mapped[2] to this one;
- `FORCED`: this is an explicitly forced variable[3].
- `ISARG`: shown for function arguments (in mouse hint popups);

User comment

Local variables may also have additional, user-defined comments which can be added using the `/` shortcut or the context menu:



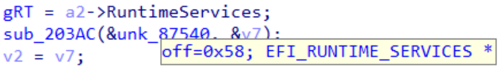
```
HKEY *v2; // edi
LSTATUS v3; // esi
LSTATUS v4; // edi
HKEY v6; // [esp-14h] [ebp-238h]
HKEY hKey; // [esp+8h] [ebp-21Ch] BYREF
DWORD cchName; // [esp+Ch] [ebp-218h] BYREF
HKEY phkResult; // [esp+10h] [ebp-214h] BYREF
HKEY *v10; // [esp+14h] [ebp-210h]
struct _FILETIME ftLastWriteTime; // [esp+18h] [ebp-20Ch] BYREF
WCHAR Name[256]; // [esp+
int v13; // [esp+220h] [e

v2 = this;
v13 = dword_100B2960;
v6 = *this;
v10 = this;
```

If present, it will be printed at the end of the variable comment, after the annotations.

Type annotations

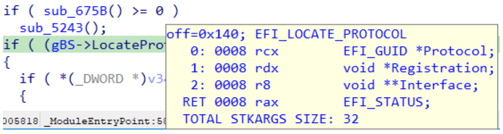
In addition to local variables, decompiler can also show annotations in the hints for:

- Structure and union fields. Offset and type is shown.



```
gRT = a2->RuntimeServices;
sub_203AC(&unk_87540, &v7);
v2 = v7;          off=0x58; EFI_RUNTIME_SERVICES *
```

- Global variables. Only the type is shown.
- Functions and function calls. The list of arguments as well as their locations is printed:



```
if ( sub_675B() >= 0 )
  sub_5243();
if ( (gBS->LocatePro
{
  if ( *(_DWORD *)v3
  {
005818 _ModuleEntryPoint:5
```

```
off=0x140; EFI_LOCATE_PROTOCOL
   0: 0008 rcx        EFI_GUID *Protocol;
   1: 0008 rdx        void *Registration;
   2: 0008 r8         void **Interface;
  RET 0008 rax        EFI_STATUS;
  TOTAL STKARGS SIZE: 32
```

[1] https://hex-rays.com/blog/igors-tip-of-the-week-51-custom-calling-conventions/
[2] https://hex-rays.com/products/decompiler/manual/cmd_map_lvar.shtml
[3] https://hex-rays.com/products/decompiler/manual/cmd_force_lvar.shtml

#67: Decompiler helpers

03 Dec 2021

https://hex-rays.com/blog/igors-tip-of-the-week-67-decompiler-helpers/

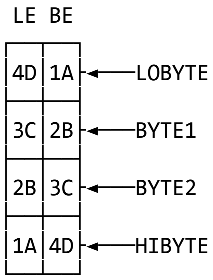
We've already described custom types[1] used in the decompiled code, but you may also encounter some unusual keywords resembling function calls. They are used by the decompiler to represent operations which it was unable to map to nice C code, or just to make the output more compact. They are listed in the `defs.h` header file that is provided with the decompiler (can be found in `plugins/hexrays_sdk/include` in your IDA directory) but here is a high level overview of the commonly seen ones.

Partial access macros

Sometimes the code may access smaller parts of a big variable. To not pollute the code with multitudes of casts, the decompiler uses helper macros for this purpose.

1. `LOWORD(x),LOWORD(x),LODWORD(x)` return the lowest byte/word/dword of the variable x as an unsigned value;
2. `HIWORD(x),HIWORD(x),HIDWORD(x)` return the corresponding high part;
3. `BYTE1(x), BYTE2(x)` etc. return individual bytes in the memory order. The variable is considered to start at byte 0 in memory;
4. same macros but with the S prefix (`SLOBYTE`, `SBYTE1` etc.) return signed values.

Note: this approach may lead to somewhat confusing situations on big endian processors like PPC. Because big-endian data is stored starting from the high byte, the low-order byte of it is stored at the highest memory address and so is accessed using the `HIBYTE` macro. For example, consider a 32-bit variable containing value 0x1A2B3C4D. It will be stored in memory in different order on little-endian(LE) and big-endian(BE) platforms:



LE	BE	
4D	1A	LOBYTE
3C	2B	BYTE1
2B	3C	BYTE2
1A	4D	HIBYTE

Combining values

Sometimes the compiler needs to represent the opposite operation: two values are combined to make a larger one. For this, "pair" macros are used:

1. `__PAIR16__(high, low)` creates a 16-bit value from two 8-bit ones. Unlike partial accesses macros, it does not depend on the memory order but uses simple bit shifts, so the result is the same for little- and big-endian code. For example, `__PAIR16__(0x1A, 0x2B)` returns in `0x1A2B` in either situation;
2. `__PAIR32__`, `__PAIR64__`, `__PAIR128__` perform the corresponding operation for bigger-sized values;
3. `__SPAIR16__` etc. return signed values.

Bit and flag manipulations

Some assembly instructions do not have simple C representation so custom helper functions are used.

1. `__ROLn__(value, count)` and `__RORn__(value, count)` (n=1,2,4,8) represent n-byte left and right bit rotates;
2. `__OFADD__` and `__OFSUB__` return the overflow flag of addition(subtraction) operation on two values;
3. `__CFADD__` and `__CFSUB__` perform the same for carry flag;
4. `__SETP__(x, y)` is used to represent the parity flag generated by expression x-y.

Overflow-checking multiplications

Recent compilers started adding overflow checks in common situations. For example, when calling `operator new[]`, behind the scenes the compiler has to multiply the size of the elements by their count. If this operation overflows, wrong value may be produced, leading to under-allocation or allocation failure. Programmers may also add manual overflow checks. The following helper functions are used to represent such code patterns:

1. `is_mul_ok(count, elsize)` represents overflow check on the result of `count*elsize`. It is presumed to return true if the overflow does not happen.
2. `saturated_mul(count, elsize)` returns either the result of multiplication if it can be calculated safely, or the maximum unsigned integer value of the corresponding size (e.g. `0xFFFFFFFF`). The latter should ensure that the allocation fails in case of overflow. This pattern is commonly used in calls to `operator new[]` in recent versions of Visual C++.

#67: Decompiler helpers

03 Dec 2021

https://hex-rays.com/blog/igors-tip-of-the-week-67-decompiler-helpers/

Value coercion

Sometimes the code treats the same underlying value as different types. For example, the famous inverse square root function from Quake treats a 32-bit floats as an integers and vice versa:

```
float InvSqrt (float x){
    float xhalf = 0.5f*x;
    int i = *(int*)&x;
    i = 0x5f3759df - (i>>1);
    x = *(float*)&i;
    x = x*(1.5f - xhalf*x*x);
    return x;
}
```

Although in the source code this conversion is represented using casts and dereferences, in the optimized code they may be replaced by simple moves between registers, especially when using SSE or AVX instructions which use the same registers to store both floating-point and integer values. Thus the decompiler has to use special macros to represent such code:

1. **`COERCE_FLOAT(v)`**, **`COERCE_DOUBLE(v)`**, **`COERCE_LONG_DOUBLE(v)`** are used to treat the bit pattern of v as the corresponding floating-point type.
2. **`COERCE_UNSIGNED_INT(v)`** and **`COERCE_UNSIGNED_INT64(v)`** are used for the opposite conversions.
3. You may also see **`SLODWORD`** when a floating-point value is treated as a signed integer.

For example, here's how pseudocode for the above function looks like when decompiled:

```
double __cdecl InvSqrt(float a1)
{
  float v2; // [esp+0h] [ebp-8h]

  v2 = a1 * 0.5;
  return (float)((1.5
                - v2 * COERCE_FLOAT(0x5F3759DF - (SLODWORD(a1) >> 1)) * COERCE_FLOAT(0x5F3759DF - (SLODWORD(a1) >> 1)))
               * COERCE_FLOAT(0x5F3759DF - (SLODWORD(a1) >> 1)));
}
```

[1] https://hex-rays.com/blog/igors-tip-of-the-week-45-decompiler-types/

#68: Skippable instructions

10 Dec 2021

https://hex-rays.com/blog/igors-tip-of-the-week-68-skippable-instructions/

In compiled code, you can sometimes find instructions which do not directly represent the code written by the programmer but were added by the compiler for its own purposes or due to the requirements of the environment the program is executing in.

Skippable instruction kinds

Compiled functions usually have prolog instructions at the start which perform various bookkeeping operations, for example:

1. preserve volatile registers used in the function's body;
2. set up new stack frame for the current function;
3. allocate stack space for local stack variables;
4. initialize the stack cookie to detect buffer overflows;
5. set up exception handlers for the current function.

In a similar manner, a function's epilog performs the opposite actions before returning to the caller.

In switch patterns there may also be instructions which only perform additional manipulations to determine the destination of an indirect jump and do not represent the actual logic of the code.

To not spend time analyzing such boilerplate or uninteresting code and only show the "real" body of the function, the decompiler relies on processor modules to mark such instructions.

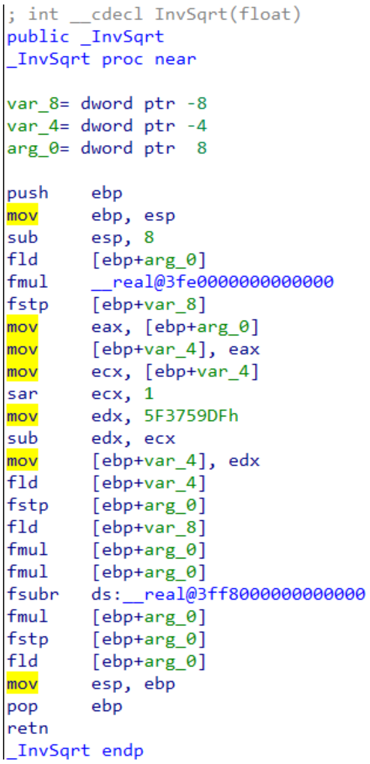
Showing skippable instructions

By default skipped instructions are not distinguished visually in any way. To enable their visualization, create a text file `idauser.cfg` with the following contents:

```
#ifdef __GUI__
PROLOG_COLOR = 0xE0E0E0 // grey
EPILOG_COLOR = 0xE0FFE0 // light green
SWITCH_COLOR = 0xE0E0FF // pink
#endif
```

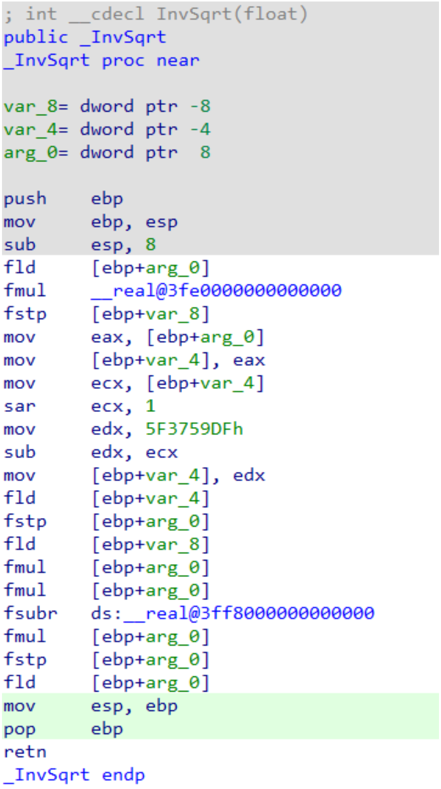
Place the file in the user directory (`%appdata%\Hex-Rays\IDA Pro` on Windows, `$HOME/.idapro` on Unix) and restart IDA or reload the database to the observe the effect in the disassembly listing.

Original disassembly:



```
; int __cdecl InvSqrt(float)
public _InvSqrt
_InvSqrt proc near

var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 8
fld     [ebp+arg_0]
fmul    __real@3fe0000000000000
fstp    [ebp+var_8]
mov     eax, [ebp+arg_0]
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_4]
sar     ecx, 1
mov     edx, 5F3759DFh
sub     edx, ecx
mov     [ebp+var_4], edx
fld     [ebp+var_4]
fstp    [ebp+arg_0]
fld     [ebp+var_8]
fmul    [ebp+arg_0]
fmul    [ebp+arg_0]
fsubr   ds:__real@3ff8000000000000
fmul    [ebp+arg_0]
fstp    [ebp+arg_0]
fld     [ebp+arg_0]
mov     esp, ebp
pop     ebp
retn
_InvSqrt endp
```

After creating the configuration file:



```
; int __cdecl InvSqrt(float)
public _InvSqrt
_InvSqrt proc near

var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 8
fld     [ebp+arg_0]
fmul    __real@3fe0000000000000
fstp    [ebp+var_8]
mov     eax, [ebp+arg_0]
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_4]
sar     ecx, 1
mov     edx, 5F3759DFh
sub     edx, ecx
mov     [ebp+var_4], edx
fld     [ebp+var_4]
fstp    [ebp+arg_0]
fld     [ebp+var_8]
fmul    [ebp+arg_0]
fmul    [ebp+arg_0]
fsubr   ds:__real@3ff8000000000000
fmul    [ebp+arg_0]
fstp    [ebp+arg_0]
fld     [ebp+arg_0]
mov     esp, ebp
pop     ebp
retn
_InvSqrt endp
```

10 Dec 2021
https://hex-rays.com/blog/igors-tip-of-the-week-68-skippable-instructions/

As you can see, the first three and last two instructions are highlighted in the specified colors. These instructions will be skipped during decompilation.

Modifying skippable instructions

There may be situations where you need to adjust IDA's idea of skipped instructions. For example, IDA may fail to mark some register saves as part of prolog (this may manifest as accesses to uninitialized variables in the pseudocode). In that case, you can fix it manually:

1. In the disassembly view, select the instruction(s) which should be marked;
2. invoke Edit > Other > Toggle skippable instructions…;
3. select the category (prolog/epilog/switch) and click OK.

In case of an opposite problem (IDA erroneously marked some instructions which do necessary work), perform the same actions, except there won't be a dialog at step 3 – the instructions will be unmarked directly.

More info: Toggle skippable instructions (Decompiler Manual)[1]

[1] https://hex-rays.com/products/decompiler/manual/interactive.shtml#06

#69: Split expression

17 Dec 2021
https://hex-rays.com/blog/igors-tip-of-the-week-69-split-expression/

While using the decompiler, sometimes you may have seen the item named Split expression in the context menu. What does it do and where it can be useful? Let’s look at two examples where it can be applied.

Structure field initialization

Modern compilers perform many optimizations to speed up code execution. One of them is merging two or more adjacent memory stores or loads into a single wide one. This often happens when writing to nearby structure fields.

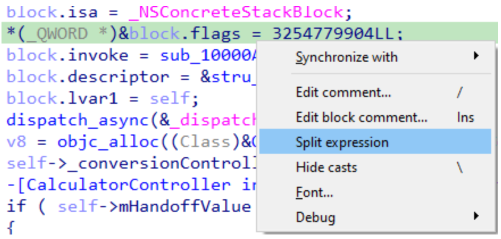
For example, when you decompile a macOS program which uses blocks[1] and use our Objective-C analysis plugin[2] to analyze the supporting code in a function, you may observe pseudocode similar to the following:

```
 block.isa = _NSConcreteStackBlock;
*(_QWORD *)&block.flags = 3254779904LL;
block.invoke = sub_10000A159;
block.descriptor = &stru_10001E0E8;
block.lvar1 = self;
```

The `block` variable uses a structure created by the plugin which looks like this:

```
struct Block_layout_10000A088
{
  void *isa;
  int32_t flags;
  int32_t reserved;
  void (__cdecl *invoke)(Block_layout_10000A088 *block);
  Block_descriptor_1 *descriptor;
  _QWORD lvar1;
};
```

As you can see, the compiler decided to initialize the two 32-bit `flags` and `reserved` fields in one go using a single 64-bit store. Although technically correct, the pseudocode looks somewhat ugly and not easy to understand at a glance. To tell the decompiler that this write should be treated as two separate ones, right-click the assignment and choose “Split expression”:



```
block.isa = _NSConcreteStackBlock;
*(_QWORD *)&block.flags = 3254779904LL;
block.invoke = sub_10000A
block.descriptor = &stru_
block.lvar1 = self;
dispatch_async(&_dispatch
v8 = objc_alloc((Class)&(
self->_conversionControll
-[CalculatorController ir
if ( self->mHandoffValue
{
```

Once the pseudocode is refreshed, two separate assignments are displayed:

```
block.isa = _NSConcreteStackBlock;
block.flags = 0xC2000000;
block.reserved = 0;
block.invoke = sub_10000A159;
block.descriptor = &stru_10001E0E8;
block.lvar1 = self;
```

The newly 32-bit constant could, for example, be converted to hex or a set of flags using a custom enum.

This example is rather benign because the `reserved` field is set to 0 so the constant was already effectively 32-bit; other situations can be more involved when different distinct values are merged into one big constant.

If necessary, expressions can be split further (e.g. when one value is used to initialize 3 or more fields). You can also revert the split by choosing “Unsplit expression” in the context menu.

#69: Split expression

17 Dec 2021

https://hex-rays.com/blog/igors-tip-of-the-week-69-split-expression/

64-bit variables in 32-bit programs

When handling 64-bit values on processors with 32-bit registers, the compiler has to work with data in 32-bit pieces. This can lead to very verbose code if translated as-is, so our decompiler detects common patterns such as 64-bit math, comparisons or data manipulations and automatically creates 64-bit variables consisting of two 32-bit registers or memory locations. While our heuristics work well in most cases, there may be false positives, when two actually separate 32-bit variables get merged into a 64-bit one. In such situation, you can use "Split expression" on the 64-bit operations involving the variable to split the pair and recover proper, separate variables.

See also: Hex-Rays interactive operation: Split/unsplit expression[3]

[1] https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithBlocks/WorkingwithBlocks.html
[2] https://hex-rays.com/products/ida/support/idadoc/1687.shtml
[3] https://hex-rays.com/products/decompiler/manual/cmd_split.shtml

#70: Multiple highlights in IDA 7.7

31 Dec 2021

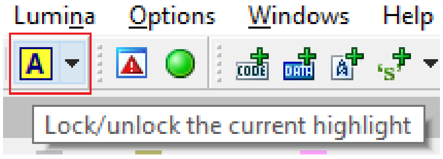
https://hex-rays.com/blog/igors-tip-of-the-week-70-multiple-highlights-in-ida-7-7/

The last week's post got preempted by the IDA 7.7 release so I'll take this opportunity to highlight (ha ha) one of the new features[1].

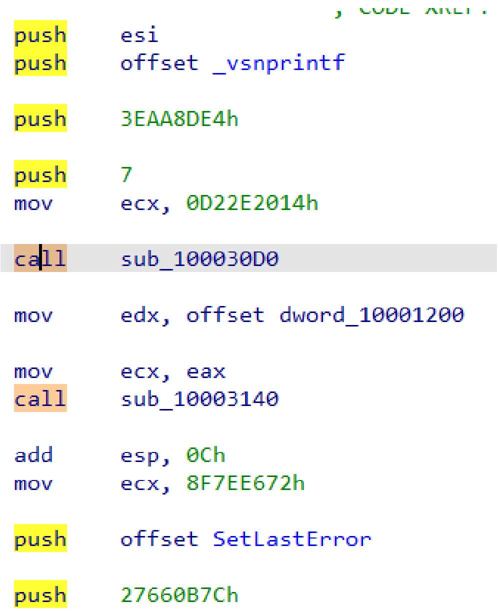
In previous IDA versions we already had highlight[2] with an option to lock it so it remains fixed while browsing the database. In IDA 7.7 it's been improved so that you can have several highlights active at the same time!

Setting highlights

Basic usage remains the same: highlight any string you want (by clicking on a word, dragging mouse, or with Shift-arrows), then click the Lock/unlock current highlight button (initially displaying A on a yellow background).



On the first glance, the effect seems to be the same: the current highlight is locked and stays on as you browse. However, if you click on another word, you'll see that the dynamic highlight now uses another color, and the lock button changes color too.

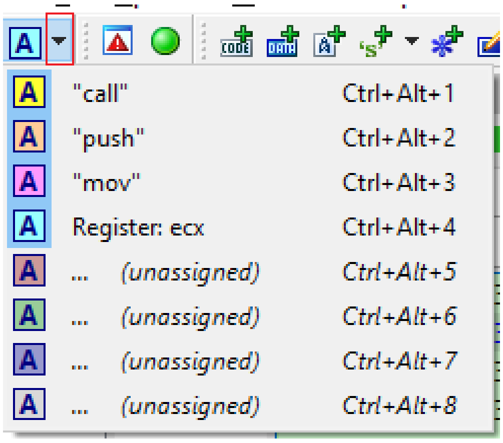


```
push    esi
push    offset _vsnprintf

push    3EAA8DE4h

push    7
mov     ecx, 0D22E2014h

call    sub_100030D0

mov     edx, offset dword_10001200

mov     ecx, eax
call    sub_10003140

add     esp, 0Ch
mov     ecx, 8F7EE672h

push    offset SetLastError

push    27660B7Ch
```

Now, if you click the button again, the second highlight gets locked and the dynamic highlight switches to the next color. You can keep doing this up to the limit (currently 8 color slots).

Removing highlights

Removing a locked highlight is pretty straightforward: click on a currently highlighted item in the listing and click on the toolbar button to unlock it. Alternatively, you can use the dropdown menu next to the button to see the currently assigned highlights and clear a specific one by picking the corresponding entry.


A "call" Ctrl+Alt+1
A "push" Ctrl+Alt+2
A "mov" Ctrl+Alt+3
A Register: ecx Ctrl+Alt+4
A ... (unassigned) Ctrl+Alt+5
A ... (unassigned) Ctrl+Alt+6
A ... (unassigned) Ctrl+Alt+7
A ... (unassigned) Ctrl+Alt+8

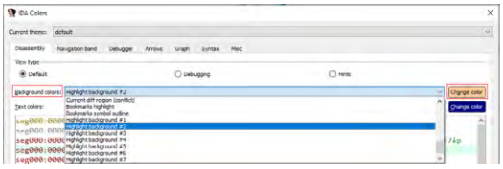
#70: Multiple highlights in IDA 7.7

31 Dec 2021

https://hex-rays.com/blog/igors-tip-of-the-week-70-multiple-highlights-in-ida-7-7/

Changing highlight colors

The highlight colors, like most others, can be changed in the Options > Colors... dialog. Select one of the "Highlight background" entries in the "Background colors" dropdown, then click "Change color" to set the new color.



Current theme: default

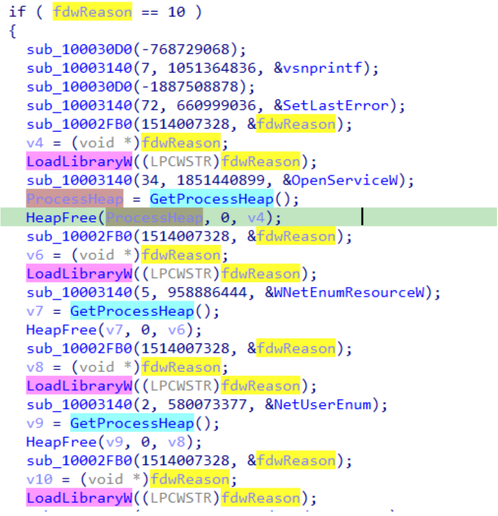
Background colors: Highlight background #2

Shortcuts

As can be seen in the screenshot of the dropdown menu, each highlight color has a corresponding shortcut `Ctrl+Alt+`digit (digit=1,2,..8), which can be used to set or clear the corresponding highlight directly.

Other views

The multiple highlight feature is available not only in the disassembly but also in other text-based views of IDA: Structures, Enums, Pseudocode, and even the Hex View, although some of them may be more or less useful than others.



```
if ( fdwReason == 10 )
{
  sub_100030D0(-768729068);
  sub_10003140(7, 1051364836, &vsnprintf);
  sub_100030D0(-1887508878);
  sub_10003140(72, 660999036, &SetLastError);
  sub_10002FB0(1514007328, &fdwReason);
  v4 = (void *)fdwReason;
  LoadLibraryW((LPCWSTR)fdwReason);
  sub_10003140(34, 1851440899, &OpenServiceW);
  ProcessHeap = GetProcessHeap();
  HeapFree(ProcessHeap, 0, v4);
  sub_10002FB0(1514007328, &fdwReason);
  v6 = (void *)fdwReason;
  LoadLibraryW((LPCWSTR)fdwReason);
  sub_10003140(5, 958886444, &WNetEnumResourceW);
  v7 = GetProcessHeap();
  HeapFree(v7, 0, v6);
  sub_10002FB0(1514007328, &fdwReason);
  v8 = (void *)fdwReason;
  LoadLibraryW((LPCWSTR)fdwReason);
  sub_10003140(2, 580073377, &NetUserEnum);
  v9 = GetProcessHeap();
  HeapFree(v9, 0, v8);
  sub_10002FB0(1514007328, &fdwReason);
  v10 = (void *)fdwReason;
  LoadLibraryW((LPCWSTR)fdwReason);
```

Hopefully, you'll find this little feature useful in your work!

[1] https://hex-rays.com/products/ida/news/7_7/

[2] https://hex-rays.com/blog/igor-tip-of-the-week-05-highlight/

#71: Decompile as call

07 Jan 2022

https://hex-rays.com/blog/igors-tip-of-the-week-71-decompile-as-call/

Although the Hex-Rays decompiler was originally written to deal with compiler-generated code, it can still do a decent job with manually written assembly. However, such code may use non-standard instructions or use them in non-standard ways, in which case the decompiler may fail to produce equivalent C code and has to fall back to _asm statements.

Analyzing system code

As an example, let's have a look at this function from a PowerPC firmware.

```
ROM:00000C8C sub_C8C:                                    # CODE XREF: ROM:00000B1C↑p
ROM:00000C8C                                             # sub_CF0+44↓p ...
ROM:00000C8C
ROM:00000C8C .set back_chain, -0x18
ROM:00000C8C .set var_C, -0xC
ROM:00000C8C .set sender_lr,  4
ROM:00000C8C
ROM:00000C8C     stwu      r1, back_chain(r1)
ROM:00000C90     mflr      r0
ROM:00000C94     stmw      r29, 0x18+var_C(r1)
ROM:00000C98     stw       r0, 0x18+sender_lr(r1)
ROM:00000C9C     addi      r31, r3, 0
ROM:00000CA0     mflr      r3
ROM:00000CA4     addi      r30, r3, 0
ROM:00000CA8     bl        sub_1264
ROM:00000CAC     lis       r29, 0x40 # '@'
ROM:00000CB0     lhz       r29, -0x2C(r29)
ROM:00000CB4     mtsprg0   r29
ROM:00000CB8     not       r11, r31
ROM:00000CBC     slwi      r11, r11, 16
ROM:00000CC0     or        r31, r11, r31
ROM:00000CC4     mtsprg1   r31
ROM:00000CC8     mtsprg2   r30
ROM:00000CCC     mftb      r3
ROM:00000CD0     addi      r30, r3, 0
ROM:00000CD4     mtsprg3   r30
ROM:00000CD8     bl        sub_1114
ROM:00000CD8 # End of function sub_C8C
```

The code seems to be using Special Purpose Register General (**sprg0**/1/2/3) for its own purposes, probably to store some information for exception processing. Because system instructions are generally not encountered in user-mode code, they are not supported by the decompiler out-of-box and the default output looks like this:

```
void __fastcall __noreturn sub_C8C(int a1)
{
  int v1; // lr

  _R30 = v1;
  sub_1264();
  _R29 = (unsigned __int16)word_3FFFD4;
  __asm { mtsprg0   r29 }
  _R31 = (~a1 << 16) | a1;
  __asm
  {
    mtsprg1   r31
    mtsprg2   r30
    mftb      r3
  }
  _R30 = _R3;
  __asm { mtsprg3   r30 }
  sub_1114();
}
```

Although the instructions themselves are shown as _asm statements, the decompiler could detect the registers used by them and created pseudo variables (_R29, _R30, _R31) to represent the operations performed. However, it is possible to get rid of _asm blocks with a bit of manual work.

#71: Decompile as call

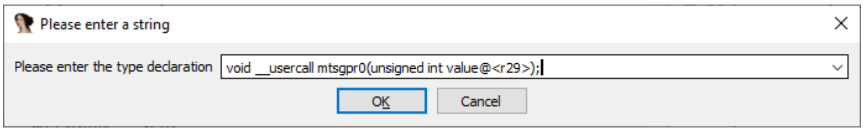
07 Jan 2022

https://hex-rays.com/blog/igors-tip-of-the-week-71-decompile-as-call/

Decompile as call

It is possible to tell the decompiler that specific instructions should be treated as if they were function calls. You can even use a custom calling convention[1] to specify the exact input/output registers of the pseudo function. Let's try it for the unhandled instructions.

1. In the disassembly view, place the cursor on the instruction (e.g. `mtsprg0 r29`);



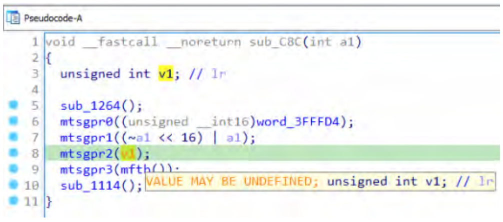
Please enter the type declaration
void __usercall mtsgpr0(unsigned int value@<r29>);

2. Invoke Edit > Other > Decompile as call...
3. Enter the prototype, taking into account input/output registers. In our example we'll use:
   `void __usercall mtsgpr0(unsigned int value<r29>);`
4. Repeat for the remaining instructions, for example:
   ```
   void __usercall mtsgpr1(unsigned int<r31>);
   void __usercall mtsgpr2(unsigned int<r30>);
   void __usercall mtsgpr3(unsigned int<r30>)
   int __usercall mftb<r3>();
   ```
5. Refresh the decompilation if it's not done automatically.

We get something like this:

```
void __fastcall __noreturn sub_C8C(int a1)
{
  unsigned int v1; // lr

  sub_1264();
  mtsgpr0((unsigned __int16)word_3FFFD4);
  mtsgpr1((~a1 << 16) | a1);
  mtsgpr2(v1);
  mtsgpr3(mftb());
  sub_1114();
}
```

No more `_asm` blocks! The only remaining wrinkle is the mysterious variable v1 which is marked in orange ("value may be undefined").



```
void __fastcall __noreturn sub_C8C(int a1)
{
  unsigned int v1; // lr

  sub_1264();
  mtsgpr0((unsigned __int16)word_3FFFD4);
  mtsgpr1((~a1 << 16) | a1);
  mtsgpr2(v1);
  mtsgpr3(mftb());
  sub_1114();
}
```

VALUE MAY BE UNDEFINED; unsigned int v1; // lr

if we look at the assembly, we'll see that the `r30` passed to `mtsprg2` originates from `r3` set by the `mflr r3` instruction. The instruction reads value of the `lr` (link register), which contains the return address to the caller and thus by definition has no determined value. However, we can use a pseudo function such as GCC's `__builtin_return_address`[2] by specifying this prototype for the mflr r3 instruction:
`void * __builtin_return_address ();`

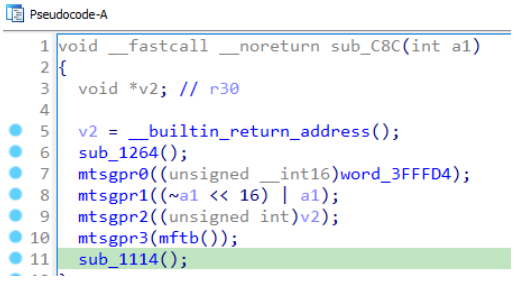
NB: We do not need to use `__usercall` here because `r3` is already the default location for a return value in the PPC ABI.

Finally, the decompilation is looking nice and tidy:

#71: Decompile as call

07 Jan 2022

https://hex-rays.com/blog/igors-tip-of-the-week-71-decompile-as-call/



```
void __fastcall __noreturn sub_C8C(int a1)
{
  void *v2; // r30

  v2 = __builtin_return_address();
  sub_1264();
  mtsgpr0((unsigned __int16)word_3FFFD4);
  mtsgpr1((~a1 << 16) | a1);
  mtsgpr2((unsigned int)v2);
  mtsgpr3(mftb());
  sub_1114();
```

Complex situations

If you want to automate the process of applying prototypes to instructions, you can use a decompiler plugin or script. For example, see the vds8[3] decompiler SDK sample (also shipped with IDA), which handles some of the SVC calls in ARM code. In even more complicated cases, such as when some arguments can't be represented by custom calling convention, or the semantics are better represented by something other than a function call (e.g. the instruction affects multiple registers), you can use a "microcode filter" to generate custom microcode which would then be optimized and converted to C code by the decompiler engine. A great example is the excellent microAVX plugin[4] by Markus Gaasedelen.

See also: Decompile as call[5] in the decompiler manual.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-51-custom-calling-conventions/
[2] https://gcc.gnu.org/onlinedocs/gcc/Return-Address.html
[3] https://github.com/idapython/src/blob/master/examples/hexrays/vds8.py
[4] https://github.com/gaasedelen/microavx/
[5] https://hex-rays.com/products/decompiler/manual/interactive.shtml#08

#72: More string literals

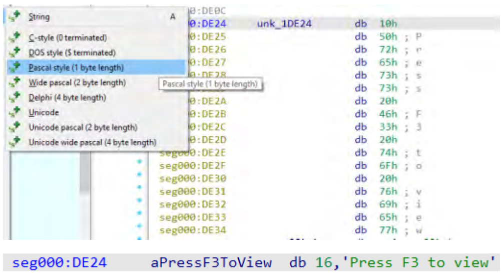
14 Jan 2022

https://hex-rays.com/blog/igors-tip-of-the-week-72-more-string-literals/

We've covered basics of working with string constants (aka string literals) before[1] but IDA support additional features which may be useful in some situations.

Exotic string types

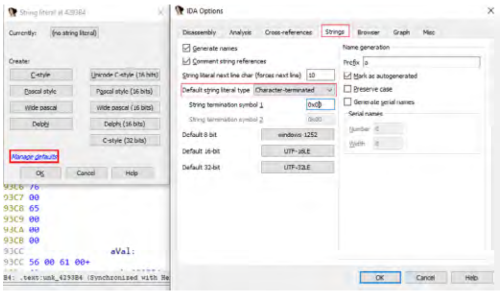
Pascal and derived languages (such as Delphi) sometimes employ string literals which start with the length followed by the characters. Similarly to the wide (Unicode) strings, they can be created using the corresponding buttons in the Options > String literals... dialog or the Edit > Strings submenu.


String A
C-style (0 terminated)
DOS style ($ terminated)
Pascal style (1 byte length)
Wide pascal (2 byte length)
Delphi (4 byte length)
Unicode
Unicode pascal (2 byte length)
Unicode wide pascal (4 byte length)
Pascal style (1 byte length)

```
seg000:DE24     unk_1DE24       db  10h
seg000:DE25                     db  50h ; P
seg000:DE26                     db  72h ; r
seg000:DE27                     db  65h ; e
seg000:DE28                     db  73h ; s
                                db  73h ; s
seg000:DE2A                     db  20h
seg000:DE2B                     db  46h ; F
seg000:DE2C                     db  33h ; 3
seg000:DE2D                     db  20h
seg000:DE2E                     db  74h ; t
seg000:DE2F                     db  6Fh ; o
seg000:DE30                     db  20h
seg000:DE31                     db  76h ; v
seg000:DE32                     db  69h ; i
seg000:DE33                     db  65h ; e
seg000:DE34                     db  77h ; w
```

```
seg000:DE24      aPressF3ToView  db 16,'Press F3 to view'
```

Some OS or embedded firmware can employ a byte other than 0 as string terminator. When analyzing such binary, you can set this up in the Options > General..., Strings tab (also accessible via Options > String literals..., "Manage defaults" link.



Currently: (no string literal)

☑ Generate names

☑ Comment string references

String literal next line char (forces next line) 10

Default string literal type Character-terminated

String termination symbol 1 0x0D

String termination symbol 2 0x00

Default 8 bit windows-1252

Default 16-bit UTF-16LE

Default 32-bit UTF-32LE

Name generation

Prefix a

☑ Mark as autogenerated

☐ Preserve case

☐ Generate serial names

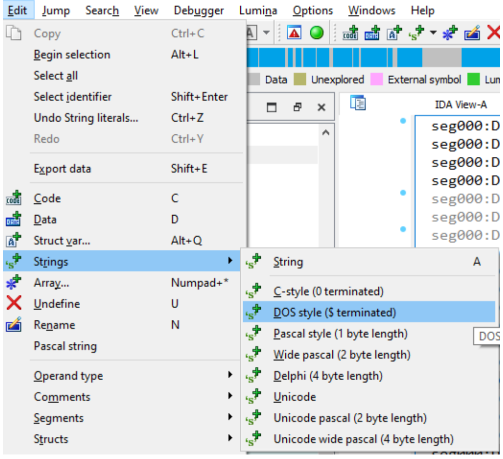
Serial names

Number 0

Width 0

```
93C6 76
93C7 00
93C8 65
93C9 00
93CA 00
93CB 00
93CC              aVal:
93CC 56 00 61 00+
B4: .text:unk_4293B4 (Synchronized with He
```

As a common variation of this type, DOS type strings (terminated with the **$** character) have their own entry in the Edit > Strings menu.



Changing string length

For already-created string literals, you can use the * shortcut to edit them as if they were an array and adjust "Array size" to change the length of the string.

See also:
Unicode strings and custom encodings[2]
How to format multiple strings placed together[3].

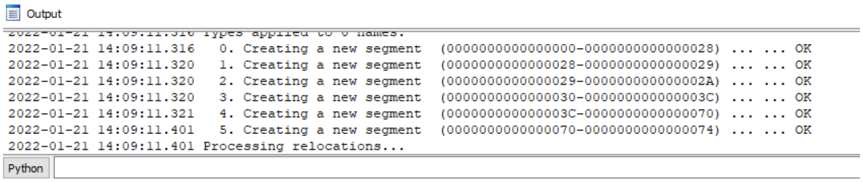
[1] https://hex-rays.com/blog/igor-tip-of-the-week-13-string-literals-and-custom-encodings/
[2] https://hex-rays.com/blog/igor-tip-of-the-week-13-string-literals-and-custom-encodings/
[3] https://hex-rays.com/blog/igor-tip-of-the-week-10-working-with-arrays/

#73: Output window and logging

21 Jan 2022

https://hex-rays.com/blog/igors-tip-of-the-week-73-output-window-and-logging/

Output window is part of IDA's default desktop layout and shows various messages from IDA and possibly third-party components (plugins, processor modules, scripts…). It also contains the Command-line interface (CLI) input box.



```
2022-01-21 14:09:11.316 Types applied to 0 names.
2022-01-21 14:09:11.316   0. Creating a new segment  (0000000000000000-0000000000000028) ... ... OK
2022-01-21 14:09:11.320   1. Creating a new segment  (0000000000000028-0000000000000029) ... ... OK
2022-01-21 14:09:11.320   2. Creating a new segment  (0000000000000029-000000000000002A) ... ... OK
2022-01-21 14:09:11.320   3. Creating a new segment  (0000000000000030-000000000000003C) ... ... OK
2022-01-21 14:09:11.321   4. Creating a new segment  (000000000000003C-0000000000000070) ... ... OK
2022-01-21 14:09:11.401   5. Creating a new segment  (0000000000000070-0000000000000074) ... ... OK
2022-01-21 14:09:11.401 Processing relocations...
```

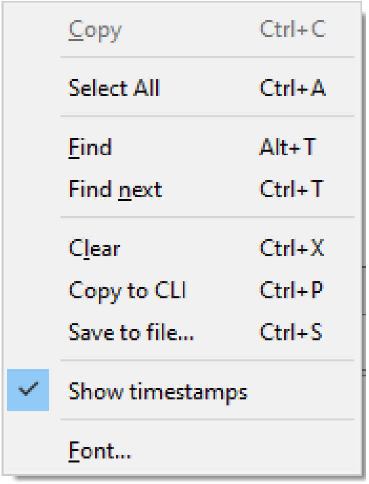
Opening the Output window

Although it is present by default, it is possible to close this window, or use a desktop layout[1] without it. If this happens, one way to restore it is to use Windows > Reset desktop to bring the layout to the initial state. But you can also use:

- Windows > Output window (shortcut `Alt+ 0`), to (re)open it and focus on the text box (for example, to select text for copying);
- Windows > Focus command line (Shortcut `Ctrl+ .`) to switch to the CLI input field, which also re-opens the Output window if it was closed.

Context menu

There are several actions available in the text box of the Output window, which can be consulted by opening the context menu:



For example, similarly to other IDA windows, you can search for text using `Alt+T`/`Ctrl+T` shortcuts, or clear the current text to easier see output of a script you're planning to run.

Timestamps

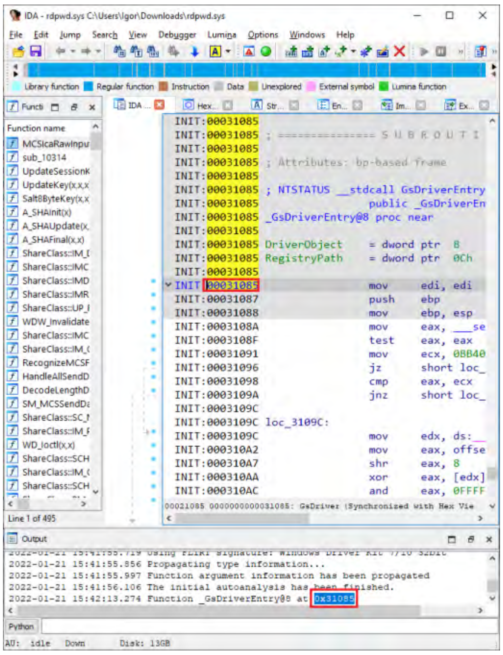
Starting from IDA 7.7[2], you can turn on timestamps for every message printed to the Output window. They are stored independently from the text so can be turned on or off at any point and affect all (past and future) messages in the current IDA session.

Navigation

Double-clicking on an address or identifier in Output window will jump to the corresponding location (if it exists) in the last active disassembly, pseudocode, or Hex view. This can be useful when writing quick scripts: just print addresses or names of interest using `msg()` function and double-click to inspect them in the disassembly listing.

#73: Output window and logging

21 Jan 2022
https://hex-rays.com/blog/igors-tip-of-the-week-73-output-window-and-logging/



```
INIT:00031085
INIT:00031085 ; =============== S U B R O U T I
INIT:00031085
INIT:00031085 ; Attributes: bp-based frame
INIT:00031085
INIT:00031085 ; NTSTATUS __stdcall GsDriverEntry
INIT:00031085                 public _GsDriverEn
INIT:00031085 _GsDriverEntry@8 proc near
INIT:00031085
INIT:00031085 DriverObject    = dword ptr  8
INIT:00031085 RegistryPath    = dword ptr  0Ch
INIT:00031085
INIT:00031085                 mov     edi, edi
INIT:00031087                 push    ebp
INIT:00031088                 mov     ebp, esp
INIT:0003108A                 mov     eax, ___se
INIT:0003108F                 test    eax, eax
INIT:00031091                 mov     ecx, 0BB40
INIT:00031096                 jz      short loc_
INIT:00031098                 cmp     eax, ecx
INIT:0003109A                 jnz     short loc_
INIT:0003109C
INIT:0003109C loc_3109C:
INIT:0003109C                 mov     edx, ds:__
INIT:000310A2                 mov     eax, offse
INIT:000310A7                 shr     eax, 8
INIT:000310AA                 xor     eax, [edx]
INIT:000310AC                 and     eax, 0FFFF
00021085 0000000000031085: GsDriver (Synchronized with Hex Vie
```

```
2022-01-21 15:41:55.719 Using FLIRT signature: Windows Driver Kit 7/10 32bit
2022-01-21 15:41:55.856 Propagating type information...
2022-01-21 15:41:55.997 Function argument information has been propagated
2022-01-21 15:41:56.106 The initial autoanalysis has been finished.
2022-01-21 15:42:13.274 Function _GsDriverEntry@8 at 0x31085
```

Logging to file

Logging of the messages in Output window to a file can be especially useful when using IDA in batch mode[3], but also in other situations (e.g. debugging scripts or plugins). The following options exist to enable it:

1. set environment variable[4] `IDALOG` to a filename. If the path is not absolute, the file will be created in the current directory. All IDA run afterwards will append output to the same file, so it can contain information from multiple runs.
2. pass the `-L<file>` command line[5] switch to IDA. Note that it has to precede the input filename.
3. On-demand, one-time saving can be done via "Save to file" context menu command (shortcut `Ctrl`+ `S`).

Note: if you have enabled timestamps in IDA, they will be added in the log file too (and in all future IDA sessions). There is currently no possibility to turn timestamps on or off via environment variable or command line switch.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-22-ida-desktop-layouts/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-48-searching-in-ida/
[3] https://hex-rays.com/blog/igor-tip-of-the-week-08-batch-mode-under-the-hood/
[4] https://www.hex-rays.com/products/ida/support/idadoc/1375.shtml
[5] https://www.hex-rays.com/products/ida/support/idadoc/417.shtml

#74: Parameter identification and tracking (PIT)

28 Jan 2022

https://hex-rays.com/blog/igors-tip-of-the-week-74-parameter-identification-and-tracking-pit/

Many features of IDA and other disassemblers are taken for granted nowadays but it’s not always been the case. As one example, let’s consider automatic variable naming.

A little bit of history

In the first versions[1], IDA did not differ much from a dumb disassembler with comments and renaming and showed pretty much raw instructions with numerical offsets. To keep track of them users often had to add manual comments.

A few versions later, support for stack variables appeared. They initially had dummy names (`var_4`, `var_C` etc.) but could be renamed by the user which eased the reverse engineering process. However, this could still be tedious in big programs.

Next, FLIRT[2] was added, which helped identify standard library functions. Now the user did not need to analyze boilerplate code from the compiler runtime libraries but only the code written by the programmer. Having identified library functions also helped in picking names for variables: most library functions had known prototypes so the variables used for their arguments could be renamed accordingly.

However, this process was still manual, could it not be automated?

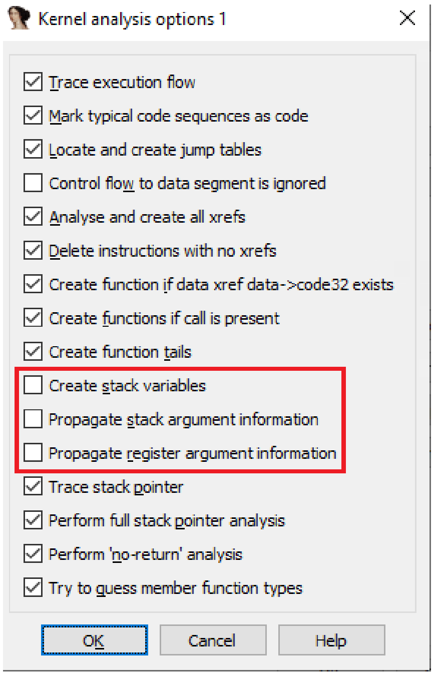
And indeed, this is what happened in IDA 4.10[3], with the addition of the type system and standard type libraries[4]. Now the identified library or imported functions could be matched to their prototypes in the type library and their arguments commented and/or renamed. For the arguments using a complex type (e.g. a structure), the stack variable could also be changed to use that type.

In practice

As a current example, let’s have a look at a Win32 program which calls `CreateWindowExA`.

First, with everything disabled:

```
mov     eax, [ebp-20h]
push    dword ptr [ebp+8]
sub     eax, [ebp-28h]
push    dword ptr [ebx+1Ch]
push    eax
mov     eax, [ebp-24h]
sub     eax, [ebp-2Ch]
push    eax
push    dword ptr [ebp-28h]
push    dword ptr [ebp-2Ch]
push    dword ptr [ebp-8]
push    edi
push    offset aEdit    ; “edit”
push    edi
call    ds:CreateWindowExA
```



Kernel analysis options 1

☑ Trace execution flow
☑ Mark typical code sequences as code
☑ Locate and create jump tables
□ Control flow to data segment is ignored
☑ Analyse and create all xrefs
☑ Delete instructions with no xrefs
☑ Create function if data xref data->code32 exists
☑ Create functions if call is present
☑ Create function tails
□ Create stack variables
□ Propagate stack argument information
□ Propagate register argument information
☑ Trace stack pointer
☑ Perform full stack pointer analysis
☑ Perform 'no-return' analysis
☑ Try to guess member function types

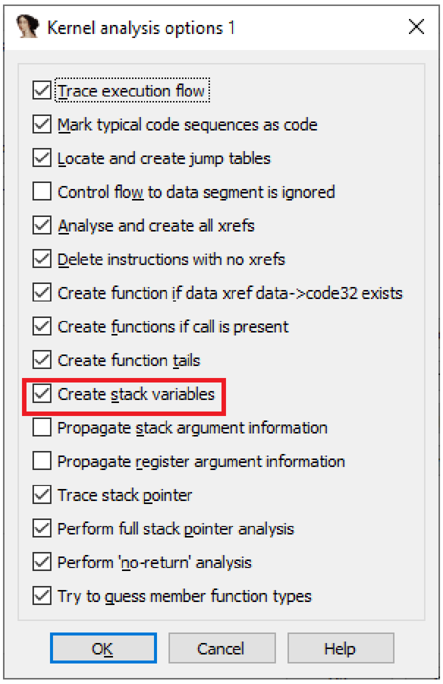
#74: Parameter identification and tracking (PIT)

28 Jan 2022

https://hex-rays.com/blog/igors-tip-of-the-week-74-parameter-identification-and-tracking-pit/

Next, with stack variables:

```
mov     eax, [ebp+var_20]
push    [ebp+arg_0]
sub     eax, [ebp+var_28]
push    dword ptr [ebx+1Ch]
push    eax
mov     eax, [ebp+var_24]
sub     eax, [ebp+var_2C]
push    eax
push    [ebp+var_28]
push    [ebp+var_2C]
push    [ebp+var_8]
push    edi
push    offset aEdit     ; “edit”
push    edi
call    ds:CreateWindowExA
```

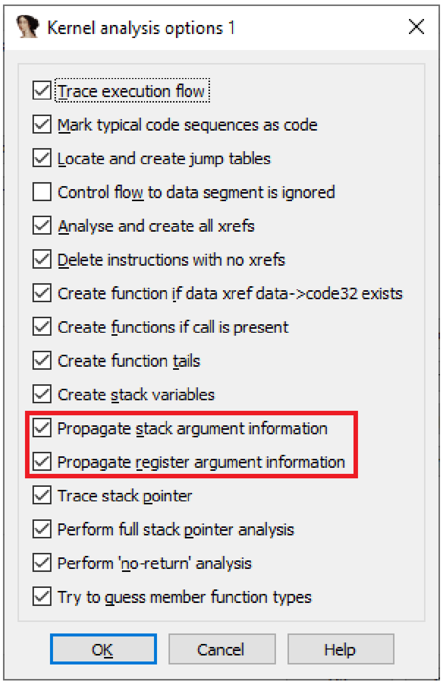


Kernel analysis options 1

☑ Trace execution flow
☑ Mark typical code sequences as code
☑ Locate and create jump tables
☐ Control flow to data segment is ignored
☑ Analyse and create all xrefs
☑ Delete instructions with no xrefs
☑ Create function if data xref data->code32 exists
☑ Create functions if call is present
☑ Create function tails
☑ Create stack variables
☐ Propagate stack argument information
☐ Propagate register argument information
☑ Trace stack pointer
☑ Perform full stack pointer analysis
☑ Perform 'no-return' analysis
☑ Try to guess member function types

Stack variables are created but use dummy names. We could consult the function’s documentation[5] and rename and retype them manually. But instead we can enable argument propagation and reanalyze the function[6]:

```
mov     eax, [ebp+Rect.bottom]
push    [ebp+hMenu]      ; hMenu
sub     eax, [ebp+Rect.top]
push    dword ptr [ebx+1Ch] ; hWndParent
push    eax              ; nHeight
mov     eax, [ebp+Rect.right]
sub     eax, [ebp+Rect.left]
push    eax              ; nWidth
push    [ebp+Rect.top]   ; Y
push    [ebp+Rect.left]  ; X
push    [ebp+dwStyle]    ; dwStyle
push    edi              ; lpWindowName
push    offset aEdit     ; “edit”
push    edi              ; dwExStyle
call    ds:CreateWindowExA
```



Kernel analysis options 1

☑ Trace execution flow
☑ Mark typical code sequences as code
☑ Locate and create jump tables
☐ Control flow to data segment is ignored
☑ Analyse and create all xrefs
☑ Delete instructions with no xrefs
☑ Create function if data xref data->code32 exists
☑ Create functions if call is present
☑ Create function tails
☑ Create stack variables
☑ Propagate stack argument information
☑ Propagate register argument information
☑ Trace stack pointer
☑ Perform full stack pointer analysis
☑ Perform 'no-return' analysis
☑ Try to guess member function types

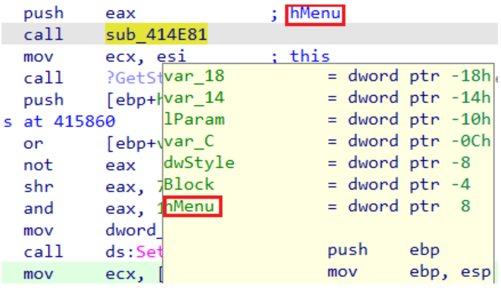
Now, all arguments are renamed and all instructions initializing them are commented. The `Rect` variable was renamed and typed thanks to another place in the same function:

```
    lea     eax, [ebp+Rect]
    push    eax             ; lpRect
    push    ebx             ; hWnd
    call    ds:GetClientRect
```

Here, IDA recognized that the `lea` instruction effectively takes an address of a struct so the stack variable should be the struct itself and not just a pointer. Thanks to this, the field references are clearly identified in the other snippet.

Recursive propagation

In fact, PIT is not limited to single functions: if any of the function's own arguments are renamed or retyped thanks to the type information, this information is propagated up the call tree. For example, `arg_0` from the second snippet is a function argument which was renamed to hMenu, so this information is used by the caller:



```
  push    eax                ; hMenu
  call    sub_414E81
  mov     ecx, esi           ; this
  call    ?GetStvar_18            = dword ptr -18h
  push    [ebp+Hvar_14            = dword ptr -14h
s at 415860     lParam            = dword ptr -10h
  or      [ebp+vvar_C             = dword ptr -0Ch
  not     eax   dwStyle           = dword ptr -8
  shr     eax, 7Block             = dword ptr -4
  and     eax, 1hMenu             = dword ptr  8
  mov     dword_
  call    ds:Set                  push    ebp
  mov     ecx, [                  mov     ebp, esp
```

[1] https://hex-rays.com/about-us/our-journey/
[2] https://hex-rays.com/products/ida/tech/flirt/
[3] https://hex-rays.com/products/ida/news/4_x/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-60-type-libraries/
[5] https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-createwindowexa
[6] https://hex-rays.com/blog/igor-tip-of-the-week-17-cross-references-2/

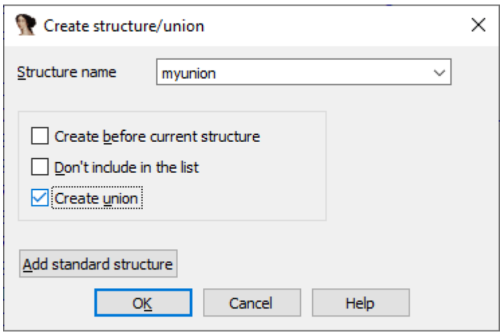
#75: Working with unions

04 Feb 2022
https://hex-rays.com/blog/igors-tip-of-the-week-75-working-with-unions/

In C, union[1] is a type similar to a struct but in which all members (possibly of different types) occupy the same memory, overlapping each other. They are used, for example, when there is a need to interpret the same data in different ways, or to save memory when storing data of different types (this is common in scripting engines, among others). IDA and the decompiler fully support unions and include definitions of commonly used ones in the standard type libraries[2], so they may be already present in the analyzed binaries.

Creating unions

Assembly-level unions can be created in the Structures window by enabling "create union" checkbox when adding a new "structure".

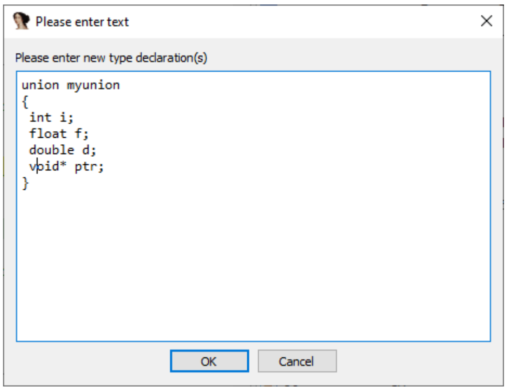


Create structure/union

Structure name: myunion

☐ Create before current structure
☐ Don't include in the list
☑ Create union

You can also use the Local Types[3] editor to create a union using C syntax.



Please enter text

Please enter new type declaration(s)

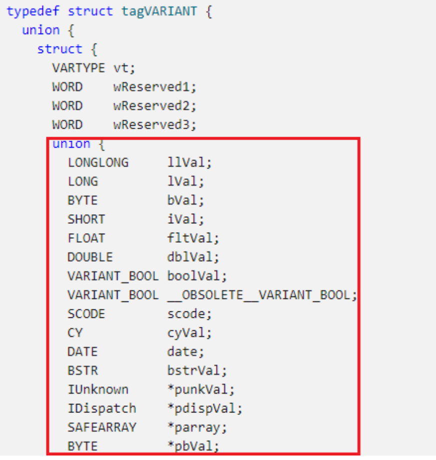
```
union myunion
{
 int i;
 float f;
 double d;
 void* ptr;
}
```

Using unions in disassembly

In disassembly, unions can be used similarly to structures. For example, when a member is referenced as an offset from a register, you can use the context menu's "Structure offset" submenu or the T hotkey. The difference is that you may see multiple "paths" for the same offset, representing alternative union members, so you can pick one most suitable for the specific use case.

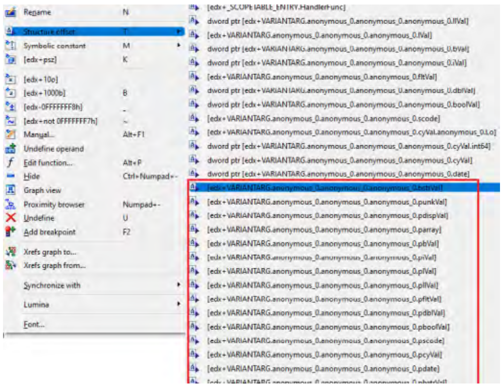
Example: OLE automation

OLE Automation is a COM-based set of APIs commonly used to implement scripting in Microsoft and other applications. One of the basic types used in it is the `VARIANT`[4] aka `VARIANTARG` structure, which can contain different types of values by embedding a union of different typed fields inside it.



```
typedef struct tagVARIANT {
  union {
    struct {
      VARTYPE vt;
      WORD    wReserved1;
      WORD    wReserved2;
      WORD    wReserved3;
      union {
        LONGLONG     llVal;
        LONG         lVal;
        BYTE         bVal;
        SHORT        iVal;
        FLOAT        fltVal;
        DOUBLE       dblVal;
        VARIANT_BOOL boolVal;
        VARIANT_BOOL __OBSOLETE__VARIANT_BOOL;
        SCODE        scode;
        CY           cyVal;
        DATE         date;
        BSTR         bstrVal;
        IUnknown     *punkVal;
        IDispatch    *pdispVal;
        SAFEARRAY    *parray;
        BYTE         *pbVal;
```

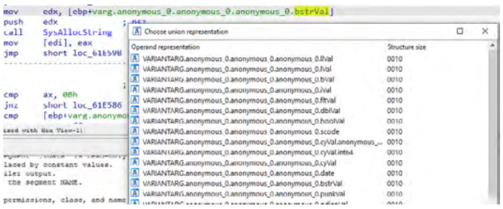
For example, if we have an instruction `mov eax, [edx+8]` and we know that `edx` points to an instance of `VARIANTARG`, using `T` on the second operand shows us multiple versions of the union field, so we can pick the most relevant to the specific code path take.



```
[edx+_SCOPETABLE_ENTRY.HandlerFunc]
dword ptr [edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.llVal]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.lVal]
dword ptr [edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.bVal]
dword ptr [edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.iVal]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.fltVal]
dword ptr [edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.dblVal]
dword ptr [edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.boolVal]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.scode]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.cyVal.anonymous_0.Lo]
dword ptr [edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.cyVal.int64]
dword ptr [edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.cyVal]
dword ptr [edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.date]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.bstrVal]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.punkVal]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.pdispVal]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.parray]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.pbVal]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.piVal]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.plVal]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.pllVal]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.pfltVal]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.pdblVal]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.pboolVal]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.pscode]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.pcyVal]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.pdate]
[edx+VARIANTARG.anonymous_0.anonymous_0.anonymous_0.pbstrVal]
```

Changing the union field used

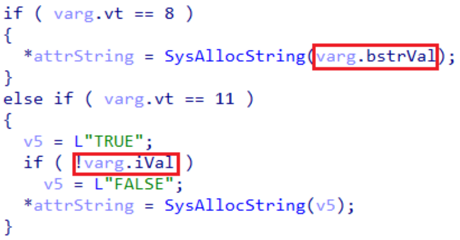
After you (or IDA) selected a union field, you can change it by going through the struct selection again (e.g. the `T` hotkey). But if the parent structure should remain the same, you can change only the union member by using the command Edit > Structs > Select union member… (hotkey `Ctrl–Y`). This can be especially useful when a structure with embedded union is placed on the stack, because you can't use the normal structure offset commands there (the offset inside the instruction is based on the stack or frame pointer which does not point to the beginning of the structure).


```
mov     edx, [ebp+varg.anonymous_0.anonymous_0.anonymous_0.bstrVal]
push    edx
call    SysAllocString
mov     [edi], eax
jmp     short loc_61E598

cmp     ax, 0Bh
jnz     short loc_61E586
cmp     [ebp+varg.anonymo
```

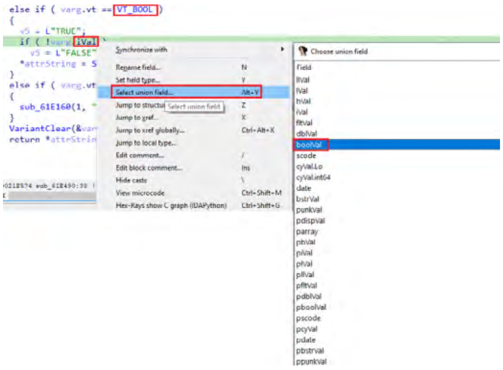
Operand representation	Structure size
VARIANTARG.anonymous_0.anonymous_0.anonymous_0.llVal	0010
VARIANTARG.anonymous_0.anonymous_0.anonymous_0.lVal	0010
VARIANTARG.anonymous_0.anonymous_0.anonymous_0.bVal	0010
VARIANTARG.anonymous_0.anonymous_0.anonymous_0.iVal	0010
VARIANTARG.anonymous_0.anonymous_0.anonymous_0.fltVal	0010
VARIANTARG.anonymous_0.anonymous_0.anonymous_0.dblVal	0010
VARIANTARG.anonymous_0.anonymous_0.anonymous_0.boolVal	0010
VARIANTARG.anonymous_0.anonymous_0.anonymous_0.scode	0010
VARIANTARG.anonymous_0.anonymous_0.anonymous_0.cyVal.anonymous_...	0010
VARIANTARG.anonymous_0.anonymous_0.anonymous_0.cyVal.int64	0010
VARIANTARG.anonymous_0.anonymous_0.anonymous_0.cyVal	0010
VARIANTARG.anonymous_0.anonymous_0.anonymous_0.date	0010
VARIANTARG.anonymous_0.anonymous_0.anonymous_0.bstrVal	0010
VARIANTARG.anonymous_0.anonymous_0.anonymous_0.punkVal	0010

Unions in decompiler

Because the decompiler can do dataflow analysis, in many cases it can pick up the most suitable union field by matching the expected type of the variable used by the code. For example, in the snippet below the decompiler picked the correct field for the argument passed to `SysAllocString`, because it knows that the function expects an argument of type `const OLECHAR *`, which is compatible with the `BSTR bstrVal` field of the union.


```
if ( varg.vt == 8 )
{
  *attrString = SysAllocString(varg.bstrVal);
}
else if ( varg.vt == 11 )
{
  v5 = L"TRUE";
  if ( !varg.iVal )
    v5 = L"FALSE";
  *attrString = SysAllocString(v5);
}
```

However, for the other reference the `iVal` filed was selected. While it is compatible for the use case (comparing against zero), by looking at the code it's obvious that the code is interpreting a boolean variant value (this can be made more clear by replacing the number 11 by the symbolic constant `VT_BOOL`). This means that boolVal is a more logical choice, and we can pick it by using "Select union field…" from the context menu, or the same `Alt– Y` hotkey as for disassembly.


```
else if ( varg.vt == VT_BOOL )
{
  v5 = L"TRUE";
  if ( !varg.iVal )
    v5 = L"FALSE"
  *attrString = S
}
else if ( varg.vt
{
  sub_61E160(1, "
}
VariantClear(&var
return *attrStrin
```

#75: Working with unions

04 Feb 2022

https://hex-rays.com/blog/igors-tip-of-the-week-75-working-with-unions/

More info:
IDA Help: Select union member[5]
Hex-Rays interactive operation: Select union field[6]

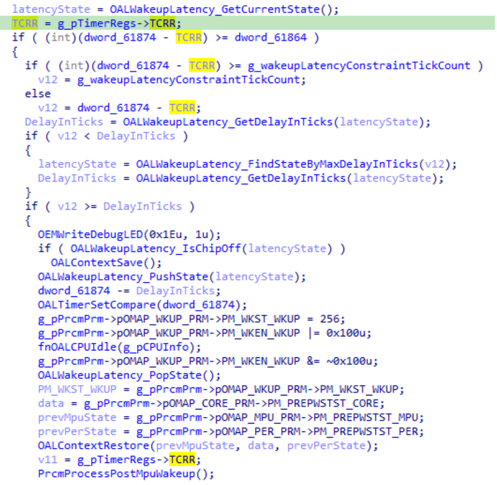
[1] https://en.cppreference.com/w/c/language/union
[2] https:// hex-rays.com/blog/igors-tip-of-the-week-60-type-libraries/
[3] https://hex-rays.com/blog/igor-tip-of-the-week-11-quickly-creating-structures/
[4] https://docs.microsoft.com/en-us/windows/win32/api/oaidl/ns-oaidl-variant
[5] https://www.hex-rays.com/products/ida/support/idadoc/498.shtml
[6] https://hex-rays.com/products/decompiler/manual/cmd_select_union_field.shtml

#76: Quick rename

11 Feb 2022

https://hex-rays.com/blog/igors-tip-of-the-week-76-quick-rename/

One of the features added in IDA 7.6[1] was automatic renaming of variables in the decompiler.



```
latencyState = OALWakeupLatency_GetCurrentState();
TCRR = g_pTimerRegs->TCRR;
if ( (int)(dword_61874 - TCRR) >= dword_61864 )
{
  if ( (int)(dword_61874 - TCRR) >= g_wakeupLatencyConstraintTickCount )
    v12 = g_wakeupLatencyConstraintTickCount;
  else
    v12 = dword_61874 - TCRR;
  DelayInTicks = OALWakeupLatency_GetDelayInTicks(latencyState);
  if ( v12 < DelayInTicks )
  {
    latencyState = OALWakeupLatency_FindStateByMaxDelayInTicks(v12);
    DelayInTicks = OALWakeupLatency_GetDelayInTicks(latencyState);
  }
  if ( v12 >= DelayInTicks )
  {
    OEMWriteDebugLED(0x1Eu, 1u);
    if ( OALWakeupLatency_IsChipOff(latencyState) )
      OALContextSave();
    OALWakeupLatency_PushState(latencyState);
    dword_61874 -= DelayInTicks;
    OALTimerSetCompare(dword_61874);
    g_pPrcmPrm->pOMAP_WKUP_PRM->PM_WKST_WKUP = 256;
    g_pPrcmPrm->pOMAP_WKUP_PRM->PM_WKEN_WKUP |= 0x100u;
    fnOALCPUIdle(g_pCPUInfo);
    g_pPrcmPrm->pOMAP_WKUP_PRM->PM_WKEN_WKUP &= ~0x100u;
    OALWakeupLatency_PopState();
    PM_WKST_WKUP = g_pPrcmPrm->pOMAP_WKUP_PRM->PM_WKST_WKUP;
    data = g_pPrcmPrm->pOMAP_CORE_PRM->PM_PREPWSTST_CORE;
    prevMpuState = g_pPrcmPrm->pOMAP_MPU_PRM->PM_PREPWSTST_MPU;
    prevPerState = g_pPrcmPrm->pOMAP_PER_PRM->PM_PREPWSTST_PER;
    OALContextRestore(prevMpuState, data, prevPerState);
    v11 = g_pTimerRegs->TCRR;
    PrcmProcessPostMpuWakeup();
```

Unlike PIT, it is not limited to stack variables but also handles variables stored in registers and not just calls but also assignments and some other expressions. It also tries to interpret function names which include a verb (get, make, fetch, query etc.) and rename the assigned result accordingly.

Triggering renaming manually

To cover situations where automatic renaming fails or insufficient, the decompiler also supports a manual action called "Quick Rename" with the default hotkey `Shift– N`. It can be used to propagate names across assignments and other expressions. Usually, it only renames dummy variables which were not explicitly named by the user (`v1`, `v2`, etc.). Here is an incomplete list of rules used by the action:

- by name of the opposite variable in assignments: `v1 = myvar`: rename **v1** -> **myvar1**
- by name of the opposite variable in comparisons: `offset < v1`: rename **v1** -> **offset1**
- as pointer to a well-named variable: `v1 = &Value`: rename **v1** -> **p_Value**
- by structure field in expressions: `v1 = x.Next`: rename **v1** -> **Next**
- as pointer to a structure field: `v1 = &x.left`: rename **v1** -> **p_left**
- by name of formal argument in a call: `close(v1)`: rename **v1** -> **fd**
- by name of a called function: `v1=create_table()`: rename **v1** -> **table**
- by return type of called function: `v1 = strchr(s, '1')`: rename **v1** -> **str**
- by a string constant: v1 = `fopen("/etc/fstab", "r")`: rename **v1** -> **etc_fstab**
- by variable type: `error_t v1`: rename **v1** -> **error**
- standard name for the result variable: `return v1`: rename **v1** -> **ok** if current function returns bool

Example: Windows driver

We'll inspect the driver used by Process Hacker[2] to perform actions requiring kernel mode access. On opening `kprocesshacker.sys`, IDA automatically applies well-known function prototype to the `DriverEntry` entrypoint and loads kernel mode type libraries, so the default decompilation is already decent:

```
NTSTATUS __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, PUNICODE_STRING RegistryPath)
{
  NTSTATUS result; // eax
  NTSTATUS v5; // r11d
  PDEVICE_OBJECT v6; // rax
  struct _UNICODE_STRING DestinationString; // [rsp+40h] [rbp-18h] BYREF
  PDEVICE_OBJECT DeviceObject; // [rsp+60h] [rbp+8h] BYREF

  qword_132C0 = (__int64)DriverObject;
  VersionInformation.dwOSVersionInfoSize = 284;
  result = RtlGetVersion(&VersionInformation);
  if ( result >= 0 )
  {
    result = sub_15100(RegistryPath);
    if ( result >= 0 )
    {
      RtlInitUnicodeString(&DestinationString, L"\\Device\\KProcessHacker3");
      result = IoCreateDevice(DriverObject, 0, &DestinationString, 0x22u, 0x100u, 0, &DeviceObject);
```

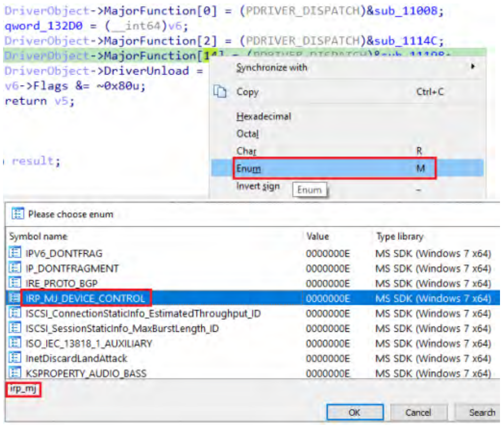
#76: Quick rename

11 Feb 2022

https://hex-rays.com/blog/igors-tip-of-the-week-76-quick-rename/

```
          v5 = result;
          if ( result >= 0 )
          {
            v6 = DeviceObject;
            DriverObject->MajorFunction[0] = (PDRIVER_DISPATCH)&sub_11008;
            qword_132D0 = (__int64)v6;
            DriverObject->MajorFunction[2] = (PDRIVER_DISPATCH)&sub_1114C;
            DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)&sub_11198;
            DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_150EC;
            v6->Flags &= ~0x80u;
            return v5;
          }
        }
      }
      return result;
    }
```

However, to make sense of it we need to make some changes. The indexes into the `MajorFunction` array are so-called IRP Major Function Codes[3] which have symbolic names starting with `IRP_MJ_`. So we can apply the Enum action (`M` hot-key) to convert numbers to the corresponding symbolic constants available in the type library.



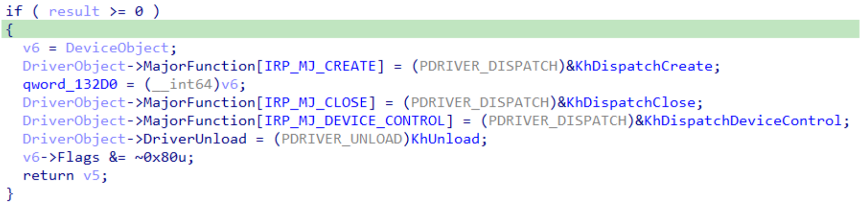
```
DriverObject->MajorFunction[0] = (PDRIVER_DISPATCH)&sub_11008;
qword_132D0 = (__int64)v6;
DriverObject->MajorFunction[2] = (PDRIVER_DISPATCH)&sub_1114C;
DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)&sub_11198;
DriverObject->DriverUnload =
v6->Flags &= ~0x80u;
return v5;

result;
```

Please choose enum

Symbol name	Value	Type library
IPV6_DONTFRAG	0000000E	MS SDK (Windows 7 x64)
IP_DONTFRAGMENT	0000000E	MS SDK (Windows 7 x64)
IRE_PROTO_BGP	0000000E	MS SDK (Windows 7 x64)
IRP_MJ_DEVICE_CONTROL	0000000E	MS SDK (Windows 7 x64)
ISCSI_ConnectionStaticInfo_EstimatedThroughput_ID	0000000E	MS SDK (Windows 7 x64)
ISCSI_SessionStaticInfo_MaxBurstLength_ID	0000000E	MS SDK (Windows 7 x64)
ISO_IEC_13818_1_AUXILIARY	0000000E	MS SDK (Windows 7 x64)
InetDiscardLandAttack	0000000E	MS SDK (Windows 7 x64)
KSPROPERTY_AUDIO_BASS	0000000E	MS SDK (Windows 7 x64)

irp_mj

Afterwards we can rename the corresponding routines and make the pseudocode look very similar to the standard DriverEntry[4]:

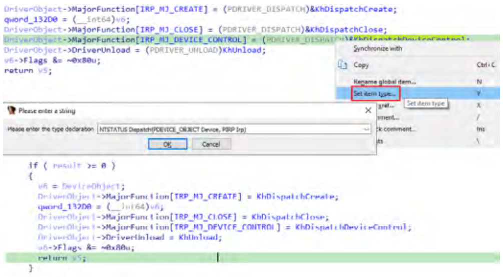


```
if ( result >= 0 )
{
  v6 = DeviceObject;
  DriverObject->MajorFunction[IRP_MJ_CREATE] = (PDRIVER_DISPATCH)&KhDispatchCreate;
  qword_132D0 = (__int64)v6;
  DriverObject->MajorFunction[IRP_MJ_CLOSE] = (PDRIVER_DISPATCH)&KhDispatchClose;
  DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = (PDRIVER_DISPATCH)&KhDispatchDeviceControl;
  DriverObject->DriverUnload = (PDRIVER_UNLOAD)KhUnload;
  v6->Flags &= ~0x80u;
  return v5;
}
```

To get rid of the casts, set the proper prototypes to the dispatch routines[5] using the "Set item type[6]" action (`Y` hotkey). We can use the same prototype string for all three routines:
`NTSTATUS Dispatch(PDEVICE_OBJECT Device, PIRP Irp)`

This works because function name is not considered to be a part of function prototype and is ignored by IDA. For the unload function, the prototype is different:
`void Unload(PDRIVER_OBJECT Driver)`

After setting the prototypes, no more casts:



```
DriverObject->MajorFunction[IRP_MJ_CREATE] = (PDRIVER_DISPATCH)&KhDispatchCreate;
qword_132D0 = (__int64)v6;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = (PDRIVER_DISPATCH)&KhDispatchClose;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = (PDRIVER_DISPATCH)&KhDispatchDeviceControl;
DriverObject->DriverUnload = (PDRIVER_UNLOAD)KhUnload;
v6->Flags &= ~0x80u;
return v5;
```

Please enter a string

Please enter the type declaration: NTSTATUS Dispatch(PDEVICE_OBJECT Device, PIRP Irp)

```
  if ( result >= 0 )
  {
    v6 = DeviceObject;
    DriverObject->MajorFunction[IRP_MJ_CREATE] = KhDispatchCreate;
    qword_132D0 = (__int64)v6;
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = KhDispatchClose;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = KhDispatchDeviceControl;
    DriverObject->DriverUnload = KhUnload;
    v6->Flags &= ~0x80u;
    return v5;
  }
```

#76: Quick rename

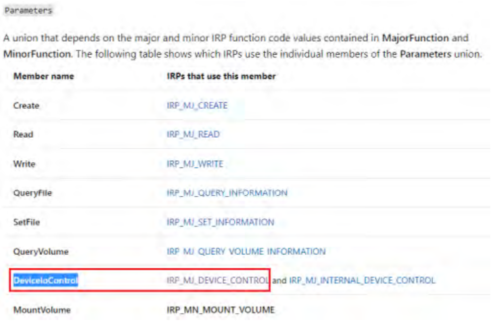
11 Feb 2022

https://hex-rays.com/blog/igors-tip-of-the-week-76-quick-rename/

Now we can go into `KhDispatchDeviceControl` to investigate how it works. Thanks to the preset prototype, the initial pseudocode looks plausible at the first glance:

```
NTSTATUS __stdcall KhDispatchDeviceControl(PDEVICE_OBJECT Device, PIRP Irp)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-”+” TO EXPAND]

  v13 = Irp;
  CurrentStackLocation = Irp->Tail.Overlay.CurrentStackLocation;
  FsContext = CurrentStackLocation->FileObject->FsContext;
  Parameters = CurrentStackLocation->Parameters.CreatePipe.Parameters;
  Options = CurrentStackLocation->Parameters.Create.Options;
  LowPart = CurrentStackLocation->Parameters.Read.ByteOffset.LowPart;
  AccessMode = Irp->RequestorMode;
  if ( !FsContext )
  {
    v9 = -1073741595;
    goto LABEL_105;
  }
  if ( LowPart != -1718018045
    && LowPart != -1718018041
    && (dword_132CC == 2 || dword_132CC == 3)
    && (*FsContext & 2) == 0 )
```

But on closer inspection, some oddities become apparent. The `Parameters` member of the `_IO_STACK_LOCATION`[7] structure is a union which contains the request-specific parameters. With insufficient information, the decompiler picked the first matching members, but they do not make sense for the request we’re dealing with. For `IRP_MJ_DEVICE_CONTROL`, the `DeviceIoControl` struct should be used.

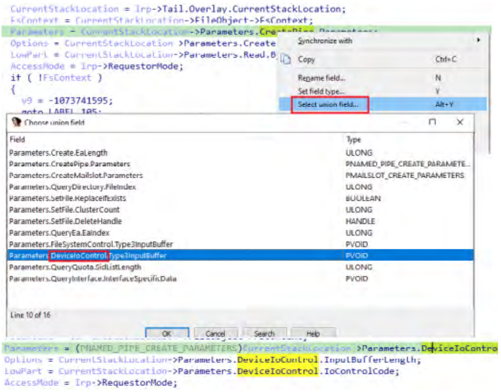


`Parameters`

A union that depends on the major and minor IRP function code values contained in **MajorFunction** and **MinorFunction**. The following table shows which IRPs use the individual members of the **Parameters** union.

Member name	IRPs that use this member
Create	IRP_MJ_CREATE
Read	IRP_MJ_READ
Write	IRP_MJ_WRITE
QueryFile	IRP_MJ_QUERY_INFORMATION
SetFile	IRP_MJ_SET_INFORMATION
QueryVolume	IRP_MJ_QUERY_VOLUME_INFORMATION
DeviceIoControl	IRP_MJ_DEVICE_CONTROL and IRP_MJ_INTERNAL_DEVICE_CONTROL
MountVolume	IRP_MN_MOUNT_VOLUME

Thus, we can use the “Select union field[8]” action (`Alt–Y` hotkey) to choose `DeviceIoControl` on the three references to `CurrentStackLocation->Parameters` to see which parameters are actually being used.

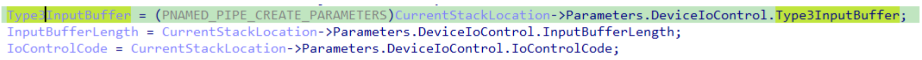


```
CurrentStackLocation = Irp->Tail.Overlay.CurrentStackLocation;
FsContext = CurrentStackLocation->FileObject->FsContext;
Parameters = CurrentStackLocation->Parameters.CreatePipe.Parameters;
Options = CurrentStackLocation->Parameters.Create
LowPart = CurrentStackLocation->Parameters.Read.B
AccessMode = Irp->RequestorMode;
if ( !FsContext )
{
  v9 = -1073741595;
  goto LABEL_105;
```

Field	Type
Parameters.Create.EaLength	ULONG
Parameters.CreatePipe.Parameters	PNAMED_PIPE_CREATE_PARAMETE...
Parameters.CreateMailslot.Parameters	PMAILSLOT_CREATE_PARAMETERS
Parameters.QueryDirectory.FileIndex	ULONG
Parameters.SetFile.ReplaceIfExists	BOOLEAN
Parameters.SetFile.ClusterCount	ULONG
Parameters.SetFile.DeleteHandle	HANDLE
Parameters.QueryEa.EaIndex	ULONG
Parameters.FileSystemControl.Type3InputBuffer	PVOID
Parameters.DeviceIoControl.Type3InputBuffer	PVOID
Parameters.QueryQuota.SidListLength	ULONG
Parameters.QueryInterface.InterfaceSpecificData	PVOID

```
Parameters = (PNAMED_PIPE_CREATE_PARAMETERS)CurrentStackLocation->Parameters.DeviceIoContro
Options = CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength;
LowPart = CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode;
AccessMode = Irp->RequestorMode;
```

The references have been changed, but the variable names and types remain. In such situation, we can update the names by using Quick rename (`Shift–N`) on the assignments.



```
Type3InputBuffer = (PNAMED_PIPE_CREATE_PARAMETERS)CurrentStackLocation->Parameters.DeviceIoControl.Type3InputBuffer;
InputBufferLength = CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength;
IoControlCode = CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode;
```

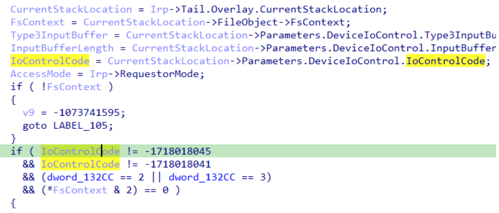
#76: Quick rename

11 Feb 2022

https://hex-rays.com/blog/igors-tip-of-the-week-76-quick-rename/

To get rid of the cast, we can either change the `Type3InputBuffer` variable type to `void*` manually, or simply refresh the decompilation (F5). This causes the decompiler to rerun the type derivation algorithm and update types of automatically typed variables.

Now the pseudocode more closely reflects what is going on. In particular, we can see that the first comparisons are checking the `IoControlCode` against some expected values, which makes more sense than the original `LowPart`.



```
CurrentStackLocation = Irp->Tail.Overlay.CurrentStackLocation;
FsContext = CurrentStackLocation->FileObject->FsContext;
Type3InputBuffer = CurrentStackLocation->Parameters.DeviceIoControl.Type3InputBu
InputBufferLength = CurrentStackLocation->Parameters.DeviceIoControl.InputBuffer
IoControlCode = CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode;
AccessMode = Irp->RequestorMode;
if ( !FsContext )
{
  v9 = -1073741595;
  goto LABEL_105;
}
if ( IoControlCode != -1718018045
  && IoControlCode != -1718018041
  && (dword_132CC == 2 || dword_132CC == 3)
  && (*FsContext & 2) == 0 )
{
```

Other uses

Quick rename can be useful when automatic renaming fails due to a name conflict. For example, if we go back to `DriverEntry`, we can see that `DeviceObject` is copied to a temporary variable `v6`:

```
v6 = DeviceObject;
DriverObject->MajorFunction[IRP_MJ_CREATE] = KhDispatchCreate;
qword_132D0 = (__int64)v6;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = KhDispatchClose;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = KhDispatchDeviceControl;
DriverObject->DriverUnload = KhUnload;
v6->Flags &= ~0x80u;
```

We can rename `v6` manually, or simply press `Shift`– `N` on the assignment and the decompiler will reuse the name with a numerical suffix to resolve the conflict:

```
DeviceObject1 = DeviceObject;
DriverObject->MajorFunction[IRP_MJ_CREATE] = KhDispatchCreate;
qword_132D0 = (__int64)DeviceObject1;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = KhDispatchClose;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = KhDispatchDeviceControl;
DriverObject->DriverUnload = KhUnload;
DeviceObject1->Flags &= ~0x80u;
```

[1] https://hex-rays.com/products/ida/news/7_6/
[2] https://processhacker.sourceforge.io/
[3] https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-major-function-codes
[4] https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/driverentry-s-required-responsibilities
[5] https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nc-wdm-driver_dispatch
[6] https://hex-rays.com/products/decompiler/manual/cmd_settype.shtml
[7] https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_io_stack_location
[8] https://hex-rays.com/products/decompiler/manual/cmd_select_union_field.shtml

#77: Mapped variables

18 Feb 2022
https://hex-rays.com/blog/igors-tip-of-the-week-77-mapped-variables/

Quick rename[1] can be useful when you have code which copies data around so the variable names stay the same or similar. However, sometimes there is a way to get rid of duplicate variables altogether.

Reasons for duplicate variables

Even if in the source code a specific variable may appear only once, on the machine code level it is not always possible. For example, most arithmetic operations use machine registers, so the values have to be moved from memory to registers to perform them. Conversely, sometimes a value has to be moved to memory from a register, for example:

- taking a reference/address of a variable requires that it resides in memory;
- when there are too few available registers, some variables have to be spilled[2] to the stack;
- when a calling convention uses stack for passing arguments;
- recursive calls or closures are usually implemented by storing the current variables on the stack;
- some other situations.

All this means that the same variable may be present in different locations during the lifetime of the function. Although the decompiler tries its best to merge these different locations into a single variable, it is not always possible, so extra variables may appear in the pseudocode.

Example

For a simple example, we can go back to `DriverEntry` in `kprocesshacker.sys` from the last post[3]. The initial output looks like this:

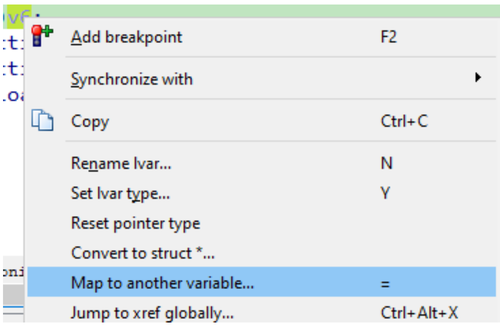
```
NTSTATUS __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, PUNICODE_STRING RegistryPath)
{
  NTSTATUS result; // eax
  NTSTATUS v5; // r11d
  PDEVICE_OBJECT v6; // rax
  struct _UNICODE_STRING DestinationString; // [rsp+40h] [rbp-18h] BYREF
  PDEVICE_OBJECT DeviceObject; // [rsp+60h] [rbp+8h] BYREF

  qword_132C0 = (__int64)DriverObject;
  VersionInformation.dwOSVersionInfoSize = 284;
  result = RtlGetVersion(&VersionInformation);
  if ( result >= 0 )
  {
    result = sub_15100(RegistryPath);
    if ( result >= 0 )
    {
      RtlInitUnicodeString(&DestinationString, L”\\Device\\KProcessHacker3”);
      result = IoCreateDevice(DriverObject, 0, &DestinationString, 0x22u, 0x100u, 0, &DeviceObject);
      v5 = result;
      if ( result >= 0 )
      {
        v6 = DeviceObject;
        DriverObject->MajorFunction[0] = (PDRIVER_DISPATCH)&sub_11008;
        qword_132D0 = (__int64)v6;
        DriverObject->MajorFunction[2] = (PDRIVER_DISPATCH)&sub_1114C;
        DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)&sub_11198;
        DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_150EC;
        v6->Flags &= ~0x80u;
        return v5;
      }
    }
  }
  return result;
}
```

We can see that there are two variables which look redundant: `v5` and `v6`. `v5` is a copy of result which resides in `r11d` and `v6` is a copy of `DeviceObject` which resides in rax. It seems they were introduced for related reasons:

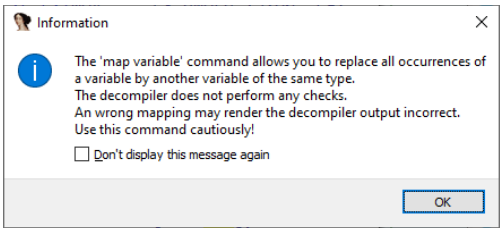
1. The compiler had to move `DeviceObject` from the stack to a register to initialize the global variable qword_132D0 and also modify the `Flags` member. It picked the register `rax` for that;
2. Because `rax` already contained the `result` variable (in the lower part of it: `eax`), it had to be saved elsewhere in the meantime (and moved back to `eax` at the end of manipulations with `DeviceObject`);
3. The decompiler could not automatically merge `DeviceObject` with `v6` because they use different storage types (stack vs register) and because in theory the writes to `DriverObject->MajorFunction` could have changed the stack variable, so the values would not be the same anymore.

Mapping variables

After looking at the code closely, it seems that `v5` and `v6` can be replaced correspondingly by result and `DeviceObject` in all cases. To ask the decompiler do it, we can use "Map to another variable[4]" action from the context menu.



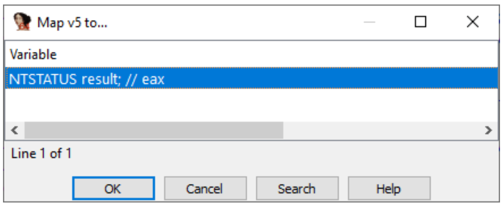
When you use it for the first time, the following warning appears:



The 'map variable' command allows you to replace all occurrences of a variable by another variable of the same type.
The decompiler does not perform any checks.
An wrong mapping may render the decompiler output incorrect.
Use this command cautiously!

□ Don't display this message again

Alternatively, you can use the hotkey = (equals sign); it's best to use it on the initial assignment such as `v6 = DeviceObject` because then the best match (the other side of assignment) will be preselected in the list of replacement candidates. In our case we get only one candidate, but in big functions you may have several variables of the same type, so triggering the action on an assignment helps ensure that you pick the correct one.



Variable
NTSTATUS result; // eax

Line 1 of 1

After mapping both variables, the output no longer mentions them:

```
NTSTATUS __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, PUNICODE_STRING RegistryPath)
{
  NTSTATUS result; // eax MAPDST
  struct _UNICODE_STRING DestinationString; // [rsp+40h] [rbp-18h] BYREF
  PDEVICE_OBJECT DeviceObject; // [rsp+60h] [rbp+8h] MAPDST BYREF

  qword_132C0 = (__int64)DriverObject;
  VersionInformation.dwOSVersionInfoSize = 284;
  result = RtlGetVersion(&VersionInformation);
  if ( result >= 0 )
  {
    result = sub_15100(RegistryPath);
    if ( result >= 0 )
    {
      RtlInitUnicodeString(&DestinationString, L”\\Device\\KProcessHacker3”);
      result = IoCreateDevice(DriverObject, 0, &DestinationString, 0x22u, 0x100u, 0, &DeviceObject);
      if ( result >= 0 )
```

#77: Mapped variables

18 Feb 2022

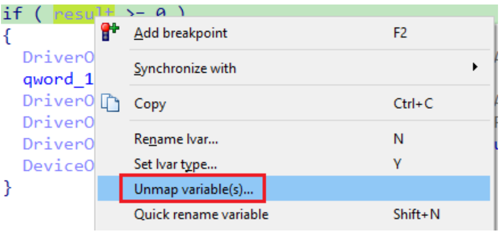
https://hex-rays.com/blog/igors-tip-of-the-week-77-mapped-variables/

```
          {
            DriverObject->MajorFunction[0] = (PDRIVER_DISPATCH)&sub_11008;
            qword_132D0 = (__int64)DeviceObject;
            DriverObject->MajorFunction[2] = (PDRIVER_DISPATCH)&sub_1114C;
            DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)&sub_11198;
            DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_150EC;
            DeviceObject->Flags &= ~0x80u;
          }
        }
      }
      return result;
    }
```

You can see that `result` and `DeviceObject` variables now have a new annotation[5]: `MAPDST`. This means that some other variable(s) have been mapped to them.

Unmapping variables

If you've changed your mind and want to see how the original pseudocode looked like, or observe something suspicious in the output involving mapped variables, you can remove the mapping by right-clicking a mapped variable (marked with MAPDST) and choosing "Unmap variable(s)".



```
if ( result >= 0 )
{
  DriverO
  qword_1
  DriverO
  DriverO
  DriverO
  DeviceO
}
```

More info: Hex-Rays interactive operation: Map to another variable[6]

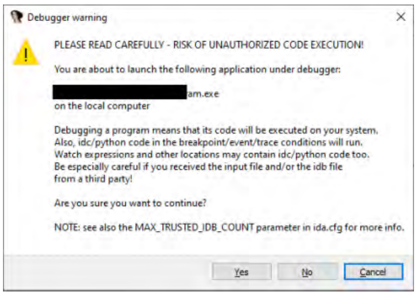
[1] https://hex-rays.com/blog/igors-tip-of-the-week-76-quick-rename/
[2] https://en.wikipedia.org/wiki/Register_allocation#Components_of_register_allocation
[3] https://hex-rays.com/blog/igors-tip-of-the-week-76-quick-rename/
[4] https://hex-rays.com/products/decompiler/manual/cmd_map_lvar.shtml
[5] https://hex-rays.com/blog/igors-tip-of-the-week-66-decompiler-annotations/
[6] https://hex-rays.com/products/decompiler/manual/cmd_map_lvar.shtml

#78: Auto-hidden messages

25 Feb 2022

https://hex-rays.com/blog/igors-tip-of-the-week-78-auto-hidden-messages/

During the work with binaries, IDA sometimes shows warnings to inform the user about unusual or potentially dangerous behavior or asks questions:



Debugger warning

PLEASE READ CAREFULLY - RISK OF UNAUTHORIZED CODE EXECUTION!

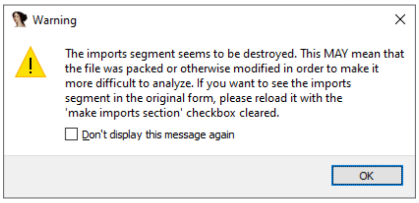
You are about to launch the following application under debugger:

[redacted]am.exe
on the local computer

Debugging a program means that its code will be executed on your system.
Also, idc/python code in the breakpoint/event/trace conditions will run.
Watch expressions and other locations may contain idc/python code too.
Be especially careful if you received the input file and/or the idb file
from a third party!

Are you sure you want to continue?

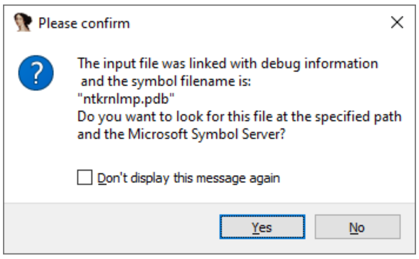
NOTE: see also the MAX_TRUSTED_IDB_COUNT parameter in ida.cfg for more info.



Warning

The imports segment seems to be destroyed. This MAY mean that
the file was packed or otherwise modified in order to make it
more difficult to analyze. If you want to see the imports
segment in the original form, please reload it with the
'make imports section' checkbox cleared.

□ Don't display this message again



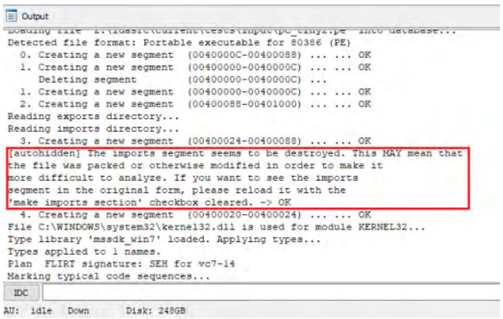
Please confirm

The input file was linked with debug information
and the symbol filename is:
"ntkrnlmp.pdb"
Do you want to look for this file at the specified path
and the Microsoft Symbol Server?

□ Don't display this message again

Hiding messages

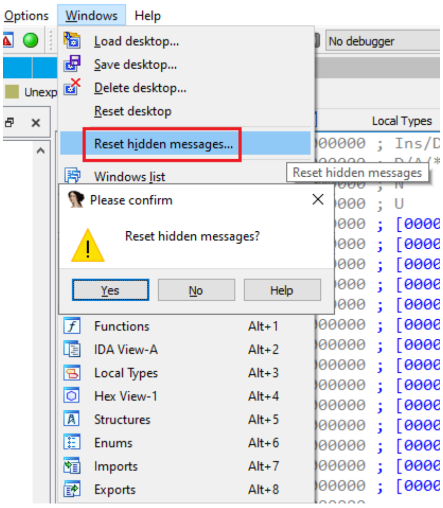
For some of such messages there is a checkbox "Don't Display this message again". If you enable it before answering or confirming the message (hint: you can press 'D' to toggle it without the mouse[1]), IDA will remember your answer and use it the next time automatically. This can be observed in the log of the Output window[2]:



```
Loading file 'Z:\idasrc\current\tests\input\pc_cinyz.pe' into database...
Detected file format: Portable executable for 80386 (PE)
  0. Creating a new segment  (0040000C-00400088) ... ... OK
  1. Creating a new segment  (00400000-0040000C) ... ... OK
     Deleting segment        (00400000-0040000C) ...
  1. Creating a new segment  (00400000-0040000C) ... ... OK
  2. Creating a new segment  (00400088-00401000) ... ... OK
Reading exports directory...
Reading imports directory...
  3. Creating a new segment  (00400024-00400088) ... ... OK
[autohidden] The imports segment seems to be destroyed. This MAY mean that
the file was packed or otherwise modified in order to make it
more difficult to analyze. If you want to see the imports
segment in the original form, please reload it with the
'make imports section' checkbox cleared. -> OK
  4. Creating a new segment  (00400020-00400024) ... ... OK
File C:\WINDOWS\system32\kernel32.dll is used for module KERNEL32...
Type library 'mssdk_win7' loaded. Applying types...
Types applied to 1 names.
Plan  FLIRT signature: SEH for vc7-14
Marking typical code sequences...
```

Changing the automatic answer

Sometimes you may change your mind and want to pick a different answer. For example, you've answered "No" to the PDB symbols questions but later you do need to load PDB symbols for a file at load time (Note: it is still possible to do it[3] after the fact using the File menu). Currently, there is no per message option but you can reset automatic answers for all of them using the menu Windows > Reset hidden messages...



Please confirm

Reset hidden messages?

After this, IDA will revert to the default settings and once again show all prompts and warnings, giving you a chance to answer differently.

IDA Help: Reset Hidden Messages[4]

[1] https://hex-rays.com/blog/igor-tip-of-the-week-01-lesser-known-keyboard-shortcuts-in-ida/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-43-annotating-the-decompiler-output/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-55-using-debug-symbols/
[4] https://www.hex-rays.com/products/ida/support/idadoc/1464.shtml

#79: Handling variable

04 Mar 2022

https://hex-rays.com/blog/igors-tip-of-the-week-79-handling-variable-reuse/

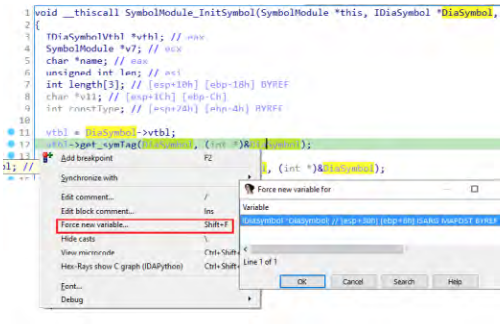
Previously we've discussed how to reduce the number of variables used in pseudocode by mapping[1] copies of a variable to one. However, sometimes you may run into an opposite problem: a single variable can be used for different purposes.

Reused stack slots

One common situation is when the compiler reuses a stack location of either a local variable or even an incoming stack argument for a different purpose. For example, in a snippet like this:

```
vtbl = DiaSymbol->vtbl;
vtbl->get_symTag(DiaSymbol, (int *)&DiaSymbol);
Symbol->Tag = (int)DiaSymbol;
```

The second argument of the call is clearly an output argument and has a different meaning and type from DiaSymbol before the call. In such case, you can use the "Force new variable" command (shortcut `Shift– F`). Due to implementation details, sometimes the option is not displayed if you right-click on the variable itself; in that case try right-clicking on the start of the pseudocode line.



```
void __thiscall SymbolModule_InitSymbol(SymbolModule *this, IDiaSymbol *DiaSymbol,
{
  IDiaSymbolVtbl *vtbl; // eax
  SymbolModule *v7; // ecx
  char *name; // eax
  unsigned int len; // esi
  int length[3]; // [esp+10h] [ebp-18h] BYREF
  char *v11; // [esp+1Ch] [ebp-Ch]
  int constType; // [esp+24h] [ebp-4h] BYREF

  vtbl = DiaSymbol->vtbl;
  vtbl->get_symTag(DiaSymbol, (int *)&DiaSymbol);
```

Force new variable for

Variable

IDiaSymbol *DiaSymbol; // [esp+30h] [ebp+8h] ISARG MAPDST BYREF

Line 1 of 1

Add breakpoint F2
Synchronize with
Edit comment... /
Edit block comment... Ins
Force new variable... Shift+F
Hide casts \
View microcode Ctrl+Shift+
Hex-Rays show C graph (IDAPython) Ctrl+Shift+
Font...
Debug
OK
Cancel
Search
Help

The decompiler creates a new variable at the same stack location, initially with the same type:

```
IDiaSymbol *v14; // [esp+30h] [ebp+8h] FORCED BYREF

vtbl = DiaSymbol->vtbl;
vtbl->get_symTag(DiaSymbol, (int *)&v14);
Symbol->Tag = (int)v14;
```

Naturally, you can change its type and name to a better one:

```
int tag; // [esp+30h] [ebp+8h] FORCED BYREF

vtbl = DiaSymbol->vtbl;
vtbl->get_symTag(DiaSymbol, &tag);
Symbol->Tag = tag;
```

Using a union to represent a polymorphic variable

Unfortunately, "Force new variable" is not available for register variables (as of IDA 7.7). In such case, using a union may work. For example, consider this snippet from `ntdll.dll`'s `LdrRelocateImage` function:

```
int v6; // esi
int v7; // eax
int v8; // edi
int v9; // eax

v6 = 0;
v20 = 0;
v7 = RtlImageNtHeader(a1);
v8 = v7;
if ( !v7 )
  return -1073741701;
v9 = *(unsigned __int16 *)(v7 + 24);
if ( v9 == IMAGE_NT_OPTIONAL_HDR32_MAGIC )
{
  v18 = *(_DWORD *)(v8 + 52);
  v16 = 0;
}
```

```
else
{
  if ( v9 != IMAGE_NT_OPTIONAL_HDR64_MAGIC )
    return -1073741701;
  v18 = *(_DWORD *)(v8 + 48);
  v16 = *(_DWORD *)(v8 + 52);
}
```

The function `RtlImageNtHeader` returns a pointer to the `IMAGE_NT_HEADERS` structure of the PE image at the given address. After changing its prototype and types of the variables, the code becomes a little more readable:

```
int v6; // esi
PIMAGE_NT_HEADERS v7; // eax
PIMAGE_NT_HEADERS v8; // edi
int Magic; // eax
int v10; // edx
int v11; // eax
unsigned int v12; // ecx
int v13; // ecx
int v15; // [esp+Ch] [ebp-10h]
unsigned int v16; // [esp+10h] [ebp-Ch]
int v17; // [esp+10h] [ebp-Ch]
unsigned int v18; // [esp+14h] [ebp-8h]
char *v19; // [esp+14h] [ebp-8h]
int v20; // [esp+18h] [ebp-4h] BYREF

v6 = 0;
v20 = 0;
v7 = RtlImageNtHeader(a1);
v8 = v7;
if ( !v7 )
  return -1073741701;
Magic = v7->OptionalHeader.Magic;
if ( Magic == IMAGE_NT_OPTIONAL_HDR32_MAGIC )
{
  v18 = v8->OptionalHeader.ImageBase;
  v16 = 0;
}
else
{
  if ( Magic != IMAGE_NT_OPTIONAL_HDR64_MAGIC )
    return -1073741701;
  v18 = v8->OptionalHeader.BaseOfData;
  v16 = v8->OptionalHeader.ImageBase;
}
```

However, there is a small problem. Judging by the checks of the magic value, the code can handle both 32-bit and 64-bit images, however the current `PIMAGE_NT_HEADERS` type is 32-bit (`PIMAGE_NT_HEADERS32`) so the code in the `else` clause is likely incorrect. If we change v8 to `PIMAGE_NT_HEADERS64`, then the if clause becomes incorrect:

```
if ( Magic == IMAGE_NT_OPTIONAL_HDR32_MAGIC )
{
  ImageBase = HIDWORD(v8->OptionalHeader.ImageBase);
  v16 = 0;
}
else
{
  if ( Magic != IMAGE_NT_OPTIONAL_HDR64_MAGIC )
    return -1073741701;
  ImageBase = v8->OptionalHeader.ImageBase;
  v16 = HIDWORD(v8->OptionalHeader.ImageBase);
}
```

#79: Handling variable

04 Mar 2022

https://hex-rays.com/blog/igors-tip-of-the-week-79-handling-variable-reuse/

We can't force a new variable because `v8` is allocated in a register and not on stack. Can we still use both types at once?

The answer is yes: we can use a union which combines both types. Here's how it can be done in this example:

1. Open Local Types (Shift-F1);
2. Add a new type (Ins);
3. Enter this definition:

```
union nt_headers
{
 PIMAGE_NT_HEADERS32 hdr32;
 PIMAGE_NT_HEADERS64 hdr64;
};
```

4. change type of `v8` to `nt_headers` and use "Select Union Field[2]" to pick the correct field in each branch of the if:

```
Magic = v7->OptionalHeader.Magic;
if ( Magic == IMAGE_NT_OPTIONAL_HDR32_MAGIC )
{
  ImageBase = v8.hdr32->OptionalHeader.ImageBase;
  ImageBaseHigh = 0;
}
else
{
  if ( Magic != IMAGE_NT_OPTIONAL_HDR64_MAGIC )
    return -1073741701;
  ImageBase = v8.hdr64->OptionalHeader.ImageBase;
  ImageBaseHigh = HIDWORD(v8.hdr64->OptionalHeader.ImageBase);
}
```

In this specific example the difference is minor and you could probably get by with some comments, but there may be situations where it makes a real difference. Note that this approach can be used for stack variables too.

See also: Hex-Rays interactive operation: Force new variable[3]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-77-mapped-variables/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-75-working-with-unions/
[3] https://www.hex-rays.com/products/decompiler/manual/cmd_force_lvar.shtml

#80: Bookmarks

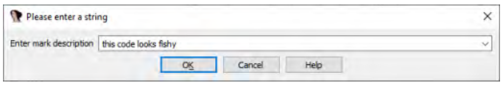
11 Mar 2022

https://hex-rays.com/blog/igors-tip-of-the-week-80-bookmarks/

In addition to comments[1], IDA offers a few more features for annotating and quickly navigating in the database. Today we'll cover bookmarks.

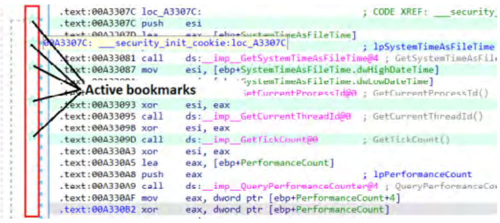
Adding bookmarks

Bookmarks can be added at most locations in the address-based views (disassembly listing, Hex View, Pseudocode), as well as Structures and Enums. This can be done via the Jump > Mark position... menu, or the hotkey `Alt– M`. You can enter a short text to describe the bookmark which will then be displayed in the bookmark list.



Enter mark description: this code looks fishy

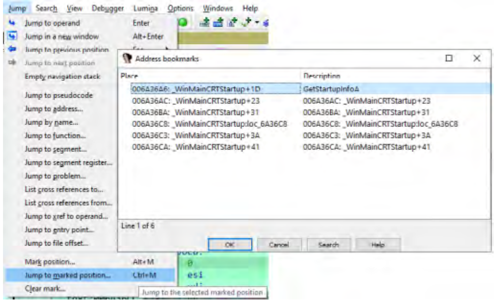
In the disassembly listing, bookmarks can be quickly added while in the text view by clicking to the left of the breakpoint circles in the execution flow arrows panel. In that case, the bookmark description will contain only the address and the label, if any. Active bookmarks are marked with the an icon which can be clicked again to remove them. Hover the mouse over the icon to see the bookmark's description.



```
.text:00A3307C loc_A3307C:                          ; CODE XREF: ___security_
.text:00A3307C push    esi
.text:00A3307D lea     eax, [ebp+SystemTimeAsFileTime]
00A3307C: ___security_init_cookie:loc_A3307C        ; lpSystemTimeAsFileTime
.text:00A33081 call    ds:__imp__GetSystemTimeAsFileTime@4 ; GetSystemTimeAsFil
.text:00A33087 mov     esi, [ebp+SystemTimeAsFileTime.dwHighDateTime]
                            SystemTimeAsFileTime.dwLowDateTime]
                       etCurrentProcessId@0 ; GetCurrentProcessId()
.text:00A33093 xor     esi, eax
.text:00A33095 call    ds:__imp__GetCurrentThreadId@0  ; GetCurrentThreadId()
.text:00A3309B xor     esi, eax
.text:00A3309D call    ds:__imp__GetTickCount@0        ; GetTickCount()
.text:00A330A3 xor     esi, eax
.text:00A330A5 lea     eax, [ebp+PerformanceCount]
.text:00A330A8 push    eax                             ; lpPerformanceCount
.text:00A330A9 call    ds:__imp__QueryPerformanceCounter@4 ; QueryPerformanceCo
.text:00A330AF mov     eax, dword ptr [ebp+PerformanceCount+4]
.text:00A330B2 xor     eax, dword ptr [ebp+PerformanceCount]
```

Managing and navigating bookmarks

To see the list of bookmarks and quickly jump to any of them, use Jump > Jump to &marked position... menu, or the `Ctrl– M` hotkey. This dialog can also be used to delete or edit bookmarks via the context menu or hotkeys (`Del` and `Ctrl– E`, respectively).

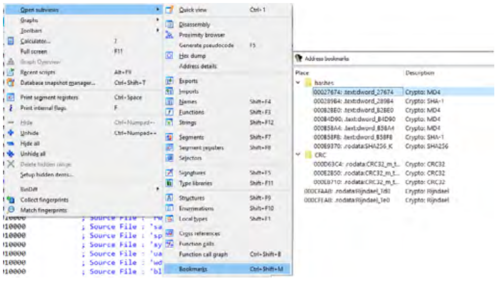


Address bookmarks

Place	Description
006A36A6: _WinMainCRTStartup+1D	GetStartupInfoA
006A36AC: _WinMainCRTStartup+23	006A36AC: _WinMainCRTStartup+23
006A36BA: _WinMainCRTStartup+31	006A36BA: _WinMainCRTStartup+31
006A36C8: _WinMainCRTStartup:loc_6A36C8	006A36C8: _WinMainCRTStartup:loc_6A36C8
006A36C3: _WinMainCRTStartup+3A	006A36C3: _WinMainCRTStartup+3A
006A36CA: _WinMainCRTStartup+41	006A36CA: _WinMainCRTStartup+41

Line 1 of 6

However, if you add many bookmarks, it can get difficult to find the one you need, so in IDA 7.6[2] we've added a dedicated bookmarks view as well as the possibility to group bookmarks into folders. The new view is available via View > Open subviews > Bookmarks, or `Ctrl– Shift– M` shortcut. The window is non-modal and can be docked.



Place	Description
hashes	
00027674: .text:dword_27674	Crypto: MD4
00028984: .text:dword_28984	Crypto: SHA-1
000B28E0: .text:dword_B28E0	Crypto: MD4
000B4D90: .text:dword_B4D90	Crypto: MD4
000B58A4: .text:dword_B58A4	Crypto: MD4
000B58F8: .text:dword_B58F8	Crypto: SHA-1
000E9370: .rodata:SHA256_K	Crypto: SHA256
CRC	
000D63C4: .rodata:CRC32_m_t...	Crypto: CRC32
000E2850: .rodata:CRC32_m_t...	Crypto: CRC32
000EB710: .rodata:CRC32_m_t...	Crypto: CRC32
000CFAA8: .rodata:Rijndael_Td0	Crypto: Rijndael
000CFEA8: .rodata:Rijndael_Te0	Crypto: Rijndael

```
10000                ; Source File : 'sa
10000                ; Source File : 'sp
10000                ; Source File : 'sy
10000                ; Source File : 'ua
10000                ; Source File : 'wd
10000                ; Source File : 'bl
```

[1] https://hex-rays.com/blog/igors-tip-of-the-week-50-execution-flow-arrows/
[2] https://hex-rays.com/products/ida/news/7_6/

#81: Database notepad

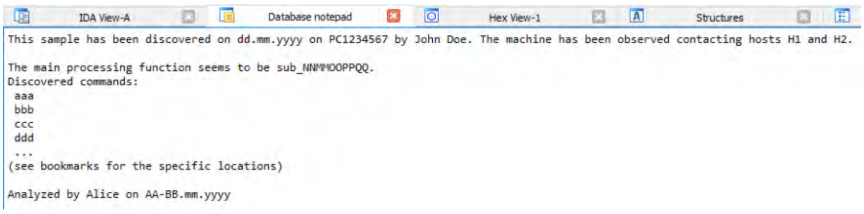
18 Mar 2022

https://hex-rays.com/blog/igors-tip-of-the-week-81-database-notepad/

There are multiple ways of annotating IDA databases: renaming, commenting[1], or adding bookmarks[2]. However, sometimes there is a need for general notes for the database as a whole, not tied to specific locations.

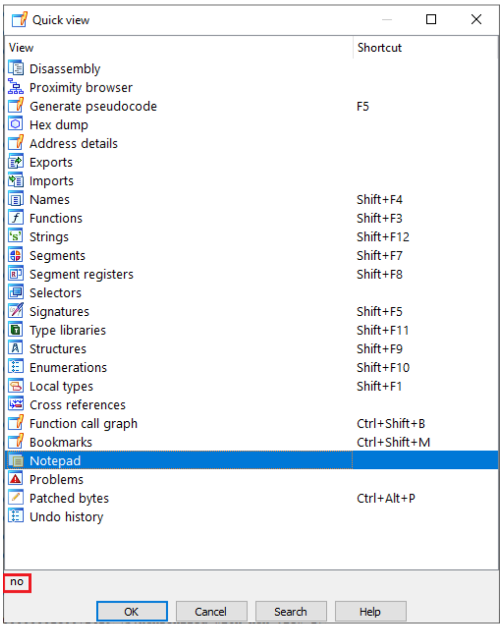
Notepad window

The database notepad is a text input box which can store arbitrary text within the database, so you can add your notes and thoughts there instead of using separate software.
It can be opened using the menu View > Open subview > Notepad.



```
This sample has been discovered on dd.mm.yyyy on PC1234567 by John Doe. The machine has been observed contacting hosts H1 and H2.

The main processing function seems to be sub_NNMMOOPPQQ.
Discovered commands:
 aaa
 bbb
 ccc
 ddd
 ...
(see bookmarks for the specific locations)

Analyzed by Alice on AA-BB.mm.yyyy
```

There is no built-in shortcut for quick access, but you can add one[3], or open the Quick view[4] (`Ctrl– 1`), type “no” to select the entry and `Enter` to activate it.



View	Shortcut
Disassembly	
Proximity browser	
Generate pseudocode	F5
Hex dump	
Address details	
Exports	
Imports	
Names	Shift+F4
Functions	Shift+F3
Strings	Shift+F12
Segments	Shift+F7
Segment registers	Shift+F8
Selectors	
Signatures	Shift+F5
Type libraries	Shift+F11
Structures	Shift+F9
Enumerations	Shift+F10
Local types	Shift+F1
Cross references	
Function call graph	Ctrl+Shift+B
Bookmarks	Ctrl+Shift+M
Notepad	
Problems	
Patched bytes	Ctrl+Alt+P
Undo history	

no

[1] https://hex-rays.com/blog/igor-tip-of-the-week-14-comments-in-ida/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-80-bookmarks/
[3] https://hex-rays.com/blog/igor-tip-of-the-week-02-ida-ui-actions-and-where-to-find-them/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-30-quick-views/

#82: Decompiler options: pseudocode formatting

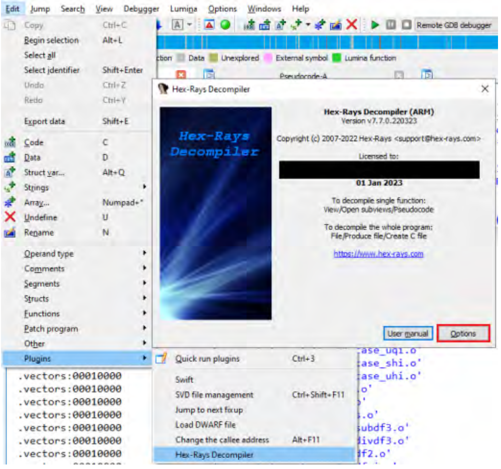
25 Mar 2022

https://hex-rays.com/blog/igors-tip-of-the-week-82-decompiler-options-pseudocode-formatting/

The default output of the Hex-Rays decompiler tries to strike a balance between conciseness and readability. However, everyone has different preferences so it offers a few options to control the layout and formatting of the pseudocode.

Accessing the options

Because of its origins as a third-party plugin for IDA, the decompiler options are accessible not through IDA's Options menu, but via Edit > Plugins > Hex-Rays Decompiler, Options button



Hex-Rays Decompiler (ARM)
Version v7.7.0.220323

Copyright (c) 2007-2022 Hex-Rays <support@hex-rays.com>

Licensed to:
[redacted]
01 Jan 2023

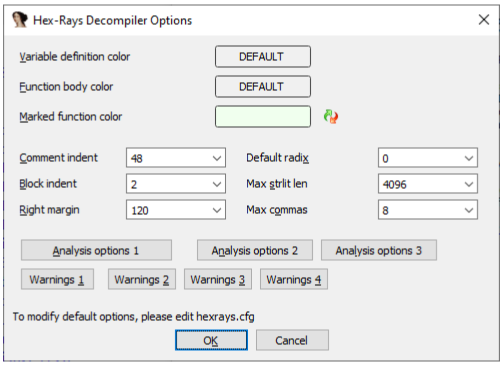
To decompile single function:
View/Open subviews/Pseudocode

To decompile the whole program:
File/Produce file/Create C file

https://www.hex-rays.com

Pseudocode formatting options

Formatting options are available on the main page of the options dialog.

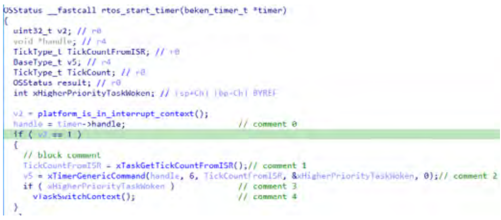


Variable definition color: DEFAULT
Function body color: DEFAULT
Marked function color:

Comment indent: 48
Block indent: 2
Right margin: 120
Default radix: 0
Max strlit len: 4096
Max commas: 8

To modify default options, please edit hexrays.cfg

- **Comment indent**: starting position for regular (end-of-line) comments. Obviously, for longer lines the comment will be shifted further to the right. Block comments[1] are aligned to the statement they're attached to so this setting does not apply to them.



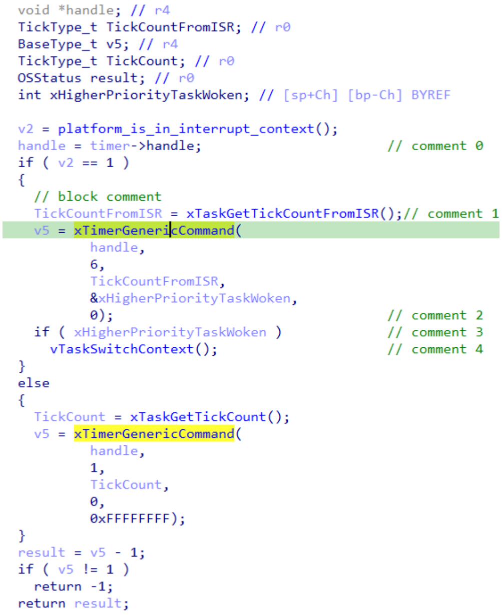
```
OSStatus __fastcall rtos_start_timer(beken_timer_t *timer)
{
  uint32_t v2; // r0
  void *handle; // r4
  TickType_t TickCountFromISR; // r0
  BaseType_t v5; // r4
  TickType_t TickCount; // r0
  OSStatus result; // r0
  int xHigherPriorityTaskWoken; // [sp+Ch] [bp-Ch] BYREF

  v2 = platform_is_in_interrupt_context();
  handle = timer->handle;                        // comment 0
  if ( v2 == 1 )
  {
    // block comment
    TickCountFromISR = xTaskGetTickCountFromISR();// comment 1
    v5 = xTimerGenericCommand(handle, 6, TickCountFromISR, &xHigherPriorityTaskWoken, 0);// comment 2
    if ( xHigherPriorityTaskWoken )              // comment 3
      vTaskSwitchContext();                      // comment 4
  }
```

- **Block indent**: indentation for nested statements, e.g. inside `if` statements or `for`/`do`/`while` loop bodies.
- **Right margin**: the decompiler tries to keep the pseudocode line length under the specified length. For example, it will try to split a function call with many arguments by putting them on separate lines:

#82: Decompiler options: pseudocode formatting

25 Mar 2022

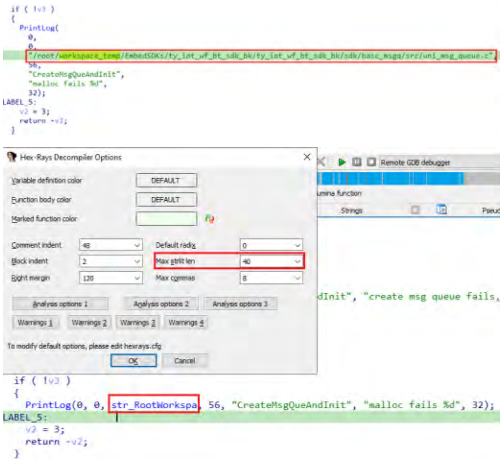
https://hex-rays.com/blog/igors-tip-of-the-week-82-decompiler-options-pseudocode-formatting/



```
void *handle; // r4
TickType_t TickCountFromISR; // r0
BaseType_t v5; // r4
TickType_t TickCount; // r0
OSStatus result; // r0
int xHigherPriorityTaskWoken; // [sp+Ch] [bp-Ch] BYREF

v2 = platform_is_in_interrupt_context();
handle = timer->handle;                       // comment 0
if ( v2 == 1 )
{
  // block comment
  TickCountFromISR = xTaskGetTickCountFromISR();// comment 1
  v5 = xTimerGenericCommand(
         handle,
         6,
         TickCountFromISR,
         &xHigherPriorityTaskWoken,
         0);                                  // comment 2
  if ( xHigherPriorityTaskWoken )             // comment 3
    vTaskSwitchContext();                     // comment 4
}
else
{
  TickCount = xTaskGetTickCount();
  v5 = xTimerGenericCommand(
         handle,
         1,
         TickCount,
         0,
         0xFFFFFFFF);
}
result = v5 - 1;
if ( v5 != 1 )
  return -1;
return result;
```

Note that in some cases you may still see lines longer than the specified margin because it's not always possible to break a long line so that it remains valid C.

- **Max strlit len**: maximum length of a string constant displayed directly (inline) in the pseudocode. A constant longer than this value will be replaced by a name referring to it.



```
if ( !v3 )
{
  PrintLog(
    0,
    0,
    "/root/workspace_temp/EmbedSDKs/ty_iot_wf_bt_sdk_bk/ty_iot_wf_bt_sdk_bk/sdk/base_msgq/src/uni_msg_queue.c",
    56,
    "CreateMsgQueAndInit",
    "malloc fails %d",
    32);
LABEL_5:
  v2 = 3;
  return -v2;
}
```

Hex-Rays Decompiler Options

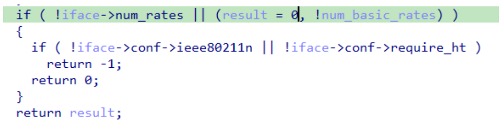
Variable definition color: DEFAULT
Function body color: DEFAULT
Marked function color:
Comment indent: 48
Block indent: 2
Right margin: 120
Default radix: 0
Max strlit len: 40
Max commas: 8

To modify default options, please edit hexrays.cfg

```
dInit", "create msg queue fails,
```

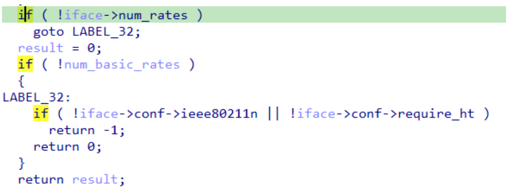
```
if ( !v3 )
{
  PrintLog(0, 0, str_RootWorkspa, 56, "CreateMsgQueAndInit", "malloc fails %d", 32);
LABEL_5:
  v2 = 3;
  return -v2;
}
```

- **Max commas**: how many comma operators[2] the decompiler can use in one expression to make the code more compact. By reducing this value, you should see simpler expressions at the cost of more lines of code: more/deeper nested `if` statements, extra variables for intermediate results, or even additional `goto` statements.
  For example, here's a fragment of pseudocode with a comma statement inside the `if` condition:



```
if ( !iface->num_rates || (result = 0, !num_basic_rates) )
{
  if ( !iface->conf->ieee80211n || !iface->conf->require_ht )
    return -1;
  return 0;
}
return result;
```

After changing max commas to 1, the comma disappears at the cost of additional `if` and `goto` statements:



```
if ( !iface->num_rates )
  goto LABEL_32;
result = 0;
if ( !num_basic_rates )
{
LABEL_32:
  if ( !iface->conf->ieee80211n || !iface->conf->require_ht )
    return -1;
  return 0;
}
return result;
```

Changing the defaults

When changing the settings from inside IDA using the UI described above, they apply only to the current database. To change the defaults for all new databases, either edit `cfg/hexrays.cfg` in IDA's install directory, or create one in the user directory[3] with the options you want to override.

Related options

Extra empty lines can be added to the pseudocode to improve readability. This feature was described in the tip #43 (Annotating the decompiler output[4]).

More info: Configuration (Hex-Rays Decompiler User Manual)[5][2]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-43-annotating-the-decompiler-output/
[2] https://en.cppreference.com/w/c/language/operator_other#Comma_operator
[3] https://hex-rays.com/blog/igors-tip-of-the-week-33-idas-user-directory-idausr/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-43-annotating-the-decompiler-output/
[5] https://www.hex-rays.com/products/decompiler/manual/config.shtml

#83: Decompiler options: default radix

01 Apr 2022

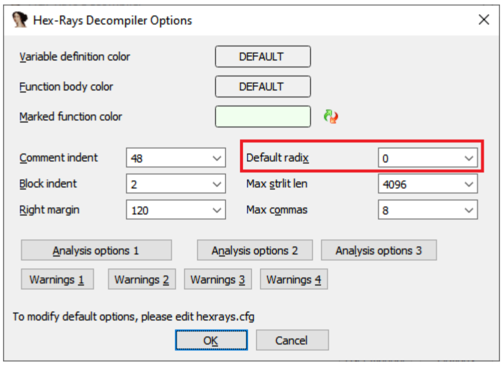
https://hex-rays.com/blog/igors-tip-of-the-week-83-decompiler-options-default-radix/

We've covered the major pseudocode formatting options previously but there is one more option which can influence the output. It is the radix used for printing numbers in the pseudocode.

> *In a positional numeral system, the **radix** or **base** is the number of unique digits, including the digit zero, used to represent numbers. For example, for the decimal/denary system (the most common system in use today) the radix (base number) is ten, because it uses the ten digits from 0 through 9.*
> *(from Wikipedia[1])*

Automatic radix

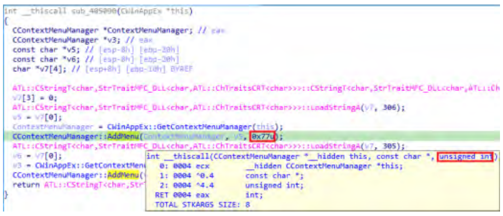
The default radix setting is 0, which means "automatic".



Hex-Rays Decompiler Options

Variable definition color	DEFAULT		
Function body color	DEFAULT		
Marked function color			
Comment indent	48	Default radix	0
Block indent	2	Max strlit len	4096
Right margin	120	Max commas	8

With this setting, the decompiler uses hexadecimal radix for values detected as unsigned, and decimal otherwise. For example, in the below screenshot, arguments to `CContextMenuManager::AddMenu()` are shown in hex because the function prototype specifies the last argument type as "unsigned int", while those for `LoadStringA()` are in decimal because the decompiler used a guessed prototype with the type `_DWORD`[2] which behaves like a signed type.



```
int __thiscall sub_405090(CWinAppEx *this)
{
  CContextMenuManager *ContextMenuManager; // eax
  CContextMenuManager *v3; // eax
  const char *v5; // [esp-8h] [ebp-20h]
  const char *v6; // [esp-8h] [ebp-20h]
  char *v7[4]; // [esp+8h] [ebp-10h] BYREF

  ATL::CStringT<char,StrTraitMFC_DLL<char,ATL::ChTraitsCRT<char>>>::CStringT<char,StrTraitMFC_DLL<char,ATL::Ch
  v7[3] = 0;
  ATL::CStringT<char,StrTraitMFC_DLL<char,ATL::ChTraitsCRT<char>>>::LoadStringA(v7, 306);
  v5 = v7[0];
  ContextMenuManager = CWinAppEx::GetContextMenuManager(this);
  CContextMenuManager::AddMenu(ContextMenuManager, v5, 0x77u);
  ATL::CStringT<char,StrTraitMFC_DLL<char,ATL::ChTraitsCRT<char>>>::LoadStringA(v7, 305);
  v6 = v7[0];
  v3 = CWinAppEx::GetContextMenu
  CContextMenuManager::AddMenu(
  return ATL::CStringT<char,Str
}
```

```
int __thiscall(CContextMenuManager *__hidden this, const char *, unsigned int)
  0: 0004 ecx        __hidden CContextMenuManager *this;
  1: 0004 ^0.4       const char *;
  2: 0004 ^4.4       unsigned int;
 RET 0004 eax        int;
 TOTAL STKARGS SIZE: 8
```

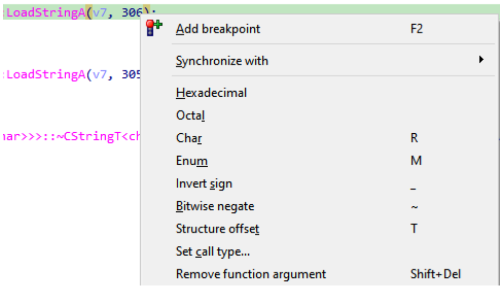
"Nice" numbers

In some cases, the decompiler may use hex even for signed numbers if it makes the number look "nice". Currently (as of IDA 7.7), the following rules are used:

1. values matching 2^n and 2^n-1 (typical bitmasks) are printed as hexadecimal.
2. 64-bit values which have not all-zero or all-one high 32 bits are printed as hexadecimal unless they end with more than 3 zeroes in decimal representation.
3. -1 is printed as decimal.

Changing the radix manually

You can always change the representation for a specific number in pseudocode from the context menu or via a hotkey.



```
LoadStringA(v7, 306);

LoadStringA(v7, 305

iar>>>::~CStringT<ch
```

To toggle between decimal and hex, use the `H` hotkey. Octal is available only via the context menu by default, but it's possible to add a custom hotkey[3] for the action name `hx:Oct`.

#83: Decompiler options: default radix

01 Apr 2022

https://hex-rays.com/blog/igors-tip-of-the-week-83-decompiler-options-default-radix/

Setting preferred radix

By changing Default radix in the decompiler options, you can have decompiler always use decimal (10) or hexadecimal(16) for all numbers without an explicitly set radix. Note that in this case the “nice” number detection will be disabled.

To change the default for all new databases, set the value **DEFAULT_RADIX** in `hexrays.cfg` as described in the previous post[4].

More info: Configuration (Hex-Rays Decompiler User Manual)[5]

[1] https://en.wikipedia.org/wiki/Radix
[2] https://hex-rays.com/blog/igors-tip-of-the-week-45-decompiler-types/
[3] https://hex-rays.com/blog/igor-tip-of-the-week-02-ida-ui-actions-and-where-to-find-them/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-82-decompiler-options-pseudocode-formatting/
[5] https://www.hex-rays.com/products/decompiler/manual/config.shtml

#84: Array indexes

08 Apr 2022

https://hex-rays.com/blog/igors-tip-of-the-week-84-array-indexes/

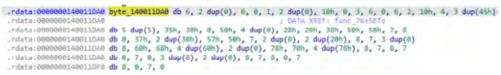
We've covered arrays previously[1], but one feature briefly mentioned there is worth a separate highlight.

Complex programs may use arrays of data, either of items such as integers or floats, or of complex items such as structures. When the arrays are small, it's not too difficult to make sense of them, but what to do if your task requires, for example, to find the value of the item #567 in a 3000-item array?

You can of course try to count the items manually or copy the array into a text editor (Export Data[2] can help here) and import into a spreadsheet but there are ways to do it inside IDA without too much trouble.

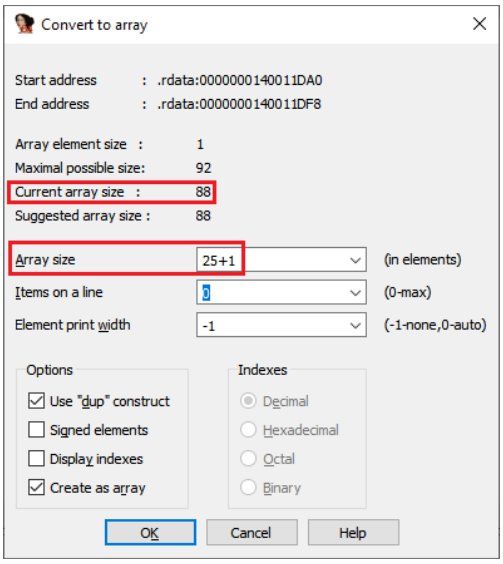
Resizing the array

Let's say we have an array of 88 items:



```
.rdata:0000000140011DA0 byte_140011DA0 db 6, 2 dup(0), 6, 0, 1, 2 dup(0), 10h, 0, 3, 6, 0, 6, 2, 10h, 4, 3 dup(45h)
.rdata:0000000140011DA0                                         ; DATA XREF: func_76+5E↑o
.rdata:0000000140011DA0 db 5 dup(5), 35h, 30h, 0, 50h, 4 dup(0), 28h, 20h, 38h, 50h, 58h, 7, 8
.rdata:0000000140011DA0 db 0, 37h, 2 dup(30h), 57h, 50h, 7, 2 dup(0), 2 dup(20h), 8, 7, 3 dup(0)
.rdata:0000000140011DA0 db 8, 60h, 68h, 4 dup(60h), 2 dup(0), 78h, 70h, 4 dup(78h), 8, 7, 8, 7
.rdata:0000000140011DA0 db 0, 7, 0, 3 dup(8), 2 dup(0), 8, 7, 8, 0, 7
.rdata:0000000140011DF8 db 8, 0, 7, 0
```

and we need the item #25. Manual counting is possible but tedious, especially because we need to account for the repeated items in the dup expressions. There is a different approach to solve this. Because the items are counted from 0 and we have 88 of them, the last one has index 87. To make it so that the last item is number 25, we can resize the array to 26(25+1) items. For this, press * to open the array parameters dialog and change the Array size field:



Convert to array

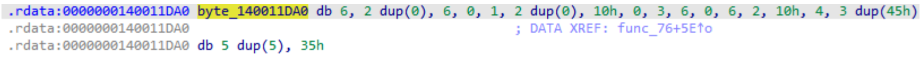
Start address : .rdata:0000000140011DA0
End address : .rdata:0000000140011DF8

Array element size : 1
Maximal possible size: 92
Current array size : 88
Suggested array size : 88

Array size 25+1 (in elements)
Items on a line 0 (0-max)
Element print width -1 (-1-none,0-auto)

Options
☑ Use "dup" construct
☐ Signed elements
☐ Display indexes
☑ Create as array

Indexes
● Decimal
○ Hexadecimal
○ Octal
○ Binary

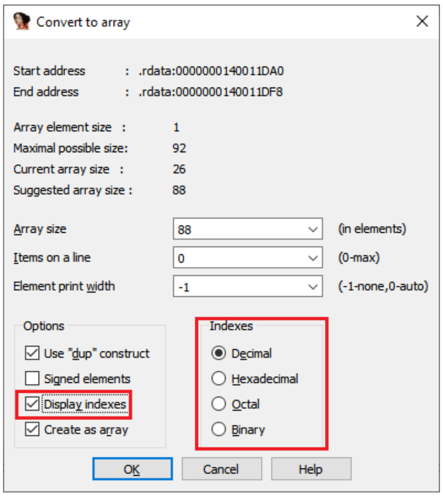


```
.rdata:0000000140011DA0 byte_140011DA0 db 6, 2 dup(0), 6, 0, 1, 2 dup(0), 10h, 0, 3, 6, 0, 6, 2, 10h, 4, 3 dup(45h)
.rdata:0000000140011DA0                                         ; DATA XREF: func_76+5E↑o
.rdata:0000000140011DA0 db 5 dup(5), 35h
```

Now the array contains 26 items from #0 to #25 so we can see that the item we needed has the value `35h`.

Array index display

Alternatively, we can enable the Display indexes option.



Convert to array

Start address : .rdata:0000000140011DA0
End address : .rdata:0000000140011DF8

Array element size : 1
Maximal possible size: 92
Current array size : 26
Suggested array size : 88

Array size 88 (in elements)
Items on a line 0 (0-max)
Element print width -1 (-1-none,0-auto)

Options
☑ Use "dup" construct
☐ Signed elements
☑ Display indexes
☑ Create as array

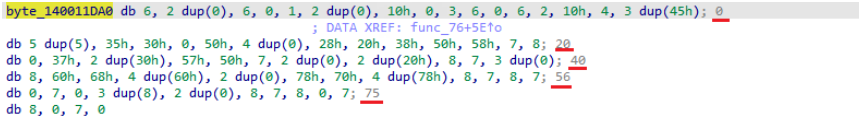
Indexes
● Decimal
○ Hexadecimal
○ Octal
○ Binary

#84: Array indexes

08 Apr 2022

https://hex-rays.com/blog/igors-tip-of-the-week-84-array-indexes/

With the option on, index of the first element is displayed as a comment for each line:

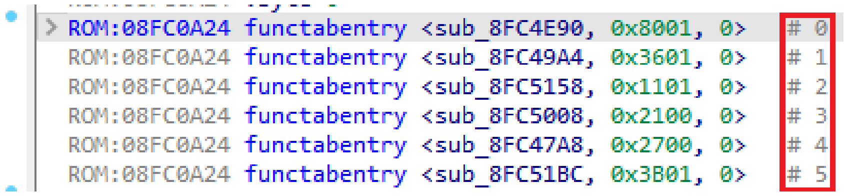


```
byte_140011DA0 db 6, 2 dup(0), 6, 0, 1, 2 dup(0), 10h, 0, 3, 6, 0, 6, 2, 10h, 4, 3 dup(45h); 0
                                          ; DATA XREF: func_76+5E↑o
db 5 dup(5), 35h, 30h, 0, 50h, 4 dup(0), 28h, 20h, 38h, 50h, 58h, 7, 8; 20
db 0, 37h, 2 dup(30h), 57h, 50h, 7, 2 dup(0), 2 dup(20h), 8, 7, 3 dup(0); 40
db 8, 60h, 68h, 4 dup(60h), 2 dup(0), 78h, 70h, 4 dup(78h), 8, 7, 8, 7; 56
db 0, 7, 0, 3 dup(8), 2 dup(0), 8, 7, 8, 0, 7; 75
db 8, 0, 7, 0
```

While still not very obvious, it is a little easier to find the necessary element by counting from the start of a line. You can also set the Items on a line value to 1 or another small value so that each line contains fewer elements and it's easier to find the necessary one.

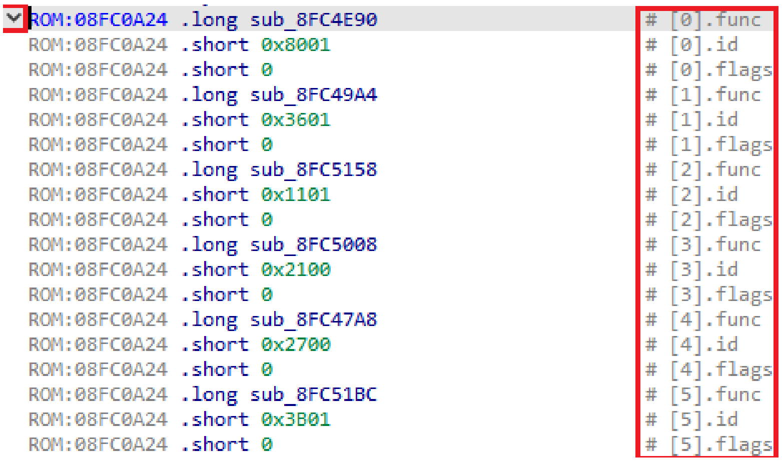
Indexes and arrays of structures

When you have an array of structures and they can be displayed in terse form[3], the indexes are printed for each line similarly to the array of simple values.



```
ROM:08FC0A24 functabentry <sub_8FC4E90, 0x8001, 0>  # 0
ROM:08FC0A24 functabentry <sub_8FC49A4, 0x3601, 0>  # 1
ROM:08FC0A24 functabentry <sub_8FC5158, 0x1101, 0>  # 2
ROM:08FC0A24 functabentry <sub_8FC5008, 0x2100, 0>  # 3
ROM:08FC0A24 functabentry <sub_8FC47A8, 0x2700, 0>  # 4
ROM:08FC0A24 functabentry <sub_8FC51BC, 0x3B01, 0>  # 5
```

However, if you unhide/uncollapse the array to show the structs in verbose form, each field gets a comment with an array notation:



```
ROM:08FC0A24 .long sub_8FC4E90       # [0].func
ROM:08FC0A24 .short 0x8001           # [0].id
ROM:08FC0A24 .short 0                # [0].flags
ROM:08FC0A24 .long sub_8FC49A4       # [1].func
ROM:08FC0A24 .short 0x3601           # [1].id
ROM:08FC0A24 .short 0                # [1].flags
ROM:08FC0A24 .long sub_8FC5158       # [2].func
ROM:08FC0A24 .short 0x1101           # [2].id
ROM:08FC0A24 .short 0                # [2].flags
ROM:08FC0A24 .long sub_8FC5008       # [3].func
ROM:08FC0A24 .short 0x2100           # [3].id
ROM:08FC0A24 .short 0                # [3].flags
ROM:08FC0A24 .long sub_8FC47A8       # [4].func
ROM:08FC0A24 .short 0x2700           # [4].id
ROM:08FC0A24 .short 0                # [4].flags
ROM:08FC0A24 .long sub_8FC51BC       # [5].func
ROM:08FC0A24 .short 0x3B01           # [5].id
ROM:08FC0A24 .short 0                # [5].flags
```

See also: IDA Help: Convert to array[4]

[1] https://hex-rays.com/blog/igor-tip-of-the-week-10-working-with-arrays/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-39-export-data/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-31-hiding-and-collapsing/
[4] https://www.hex-rays.com/products/ida/support/idadoc/455.shtml

#85: Source-level debugging

15 Apr 2022

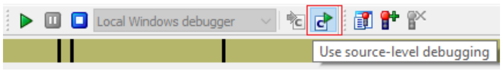
https://hex-rays.com/blog/igors-tip-of-the-week-85-source-level-debugging/

Although IDA has been created first and foremost to analyze binaries in “black box” mode, i.e. without any symbols or debug information, it does have the ability to consume such information when available[1].

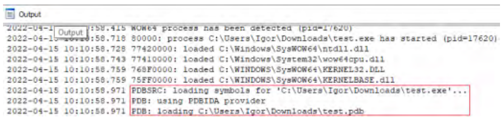
The debugger functionality was also initially optimized to debug binaries on the assembly level, but nowadays can work with source code too.

Source-level debugging

Source-level debugging is enabled by default but can be turned off or on manually via Debugger > Use source-level debugging menu item, or the button on the Debug toolbar.



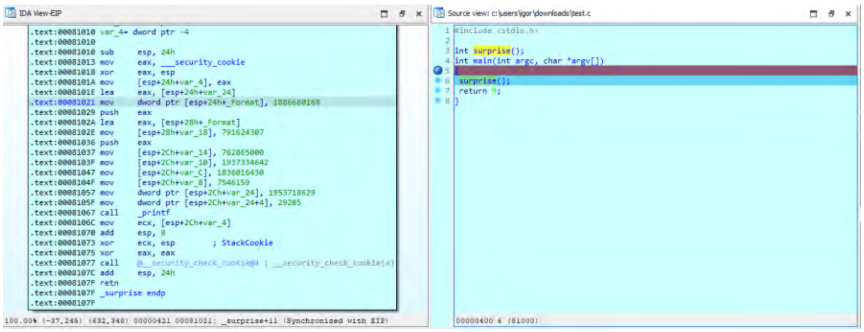
If the input file has debugging info in the format supported by IDA (e.g. PDB or DWARF), it will be automatically used when the debugging starts and the code being executed is covered by the debug info.



```
2022-04-1      :58.415 WOW64 process has been detected (pid=17620)
2022-04-1      :58.718 80000: process C:\Users\Igor\Downloads\test.exe has started (pid=17620)
2022-04-15 10:10:58.728 77420000: loaded C:\Windows\SysWOW64\ntdll.dll
2022-04-15 10:10:58.743 77410000: loaded C:\Windows\System32\wow64cpu.dll
2022-04-15 10:10:58.759 768F0000: loaded C:\WINDOWS\SysWOW64\KERNEL32.DLL
2022-04-15 10:10:58.759 75FF0000: loaded C:\WINDOWS\SysWOW64\KERNELBASE.dll
2022-04-15 10:10:58.971 PDBSRC: loading symbols for 'C:\Users\Igor\Downloads\test.exe'...
2022-04-15 10:10:58.971 PDB: using PDBIDA provider
2022-04-15 10:10:58.971 PDB: loading C:\Users\Igor\Downloads\test.pdb
```

Source code files

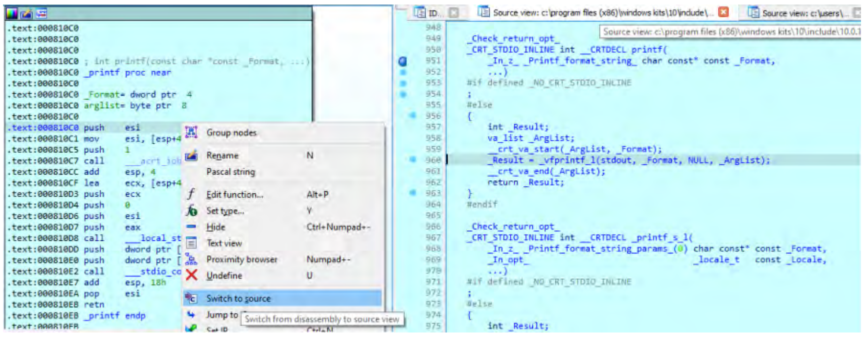
If source files are present in the original locations, IDA will open them in separate source view windows and highlight the currently executing line. The assembly instructions are still shown in the IDA View, and you can continue to use it for analysis independently of source code view. Note that IDA may automatically switch to disassembly when stepping through instructions which do not have a correspondence in the source code (for example, compiler helper functions, or auxiliary code such as prolog or epilog instructions).



```
.text:00081010 var_4= dword ptr -4
.text:00081010
.text:00081010 sub     esp, 24h
.text:00081013 mov     eax, ___security_cookie
.text:00081018 xor     eax, esp
.text:0008101A mov     [esp+24h+var_4], eax
.text:0008101E lea     eax, [esp+24h+var_24]
.text:00081021 mov     dword ptr [esp+24h+_Format], 1886680168
.text:00081029 push    eax
.text:0008102A lea     eax, [esp+28h+_Format]
.text:0008102E mov     [esp+28h+var_18], 791624307
.text:00081036 push    eax
.text:00081037 mov     [esp+2Ch+var_14], 762865000
.text:0008103F mov     [esp+2Ch+var_10], 1937334642
.text:00081047 mov     [esp+2Ch+var_C], 1836016430
.text:0008104F mov     [esp+2Ch+var_8], 7546159
.text:00081057 mov     dword ptr [esp+2Ch+var_24], 1953718629
.text:0008105F mov     dword ptr [esp+2Ch+var_24+4], 29285
.text:00081067 call    _printf
.text:0008106C mov     ecx, [esp+2Ch+var_4]
.text:00081070 add     esp, 8
.text:00081073 xor     ecx, esp        ; StackCookie
.text:00081075 xor     eax, eax
.text:00081077 call    @__security_check_cookie@4 ; __security_check_cookie(x)
.text:0008107C add     esp, 24h
.text:0008107F retn
.text:0008107F _surprise endp
.text:0008107F
100.00% (-37,245) (632,348) 00000421 00081021: _surprise+11 (Synchronized with EIP)
```

```
#include <stdio.h>

int surprise();
int main(int argc, char *argv[])
{
 surprise();
 return 0;
}
00000400 6 (81000)
```

When, after stepping through disassembly, the execution returns to the area covered by the source code, you can ask IDA to show the corresponding source code again via the action Debugger > Switch to source (also available in context menu and toolbar). This action can be used even for code away from the current execution point.


ID...
Source view: c:\program files (x86)\windows kits\10\include\...
Source view: c:\users\...
Source view: c:\program files (x86)\windows kits\10\include\10.0.1
Group nodes
Rename N
Pascal string
Edit function... Alt+P
Set type... Y
Hide Ctrl+Numpad+-
Text view
Proximity browser Numpad+-
Undefine U
Switch to source
Jump to
Switch from disassembly to source view
Set IP Ctrl+N

```
.text:000810C0
.text:000810C0
.text:000810C0
.text:000810C0 ; int printf(const char *const _Format, ...)
.text:000810C0 _printf proc near
.text:000810C0
.text:000810C0 _Format= dword ptr  4
.text:000810C0 arglist= byte ptr  8
.text:000810C0
.text:000810C0 push    esi
.text:000810C1 mov     esi, [esp+4
.text:000810C5 push    1
.text:000810C7 call    ___acrt_iob
.text:000810CC add     esp, 4
.text:000810CF lea     ecx, [esp+4
.text:000810D3 push    ecx
.text:000810D4 push    0
.text:000810D6 push    esi
.text:000810D7 push    eax
.text:000810D8 call    ___local_st
.text:000810DD push    dword ptr [
.text:000810E0 push    dword ptr [
.text:000810E2 call    ___stdio_co
.text:000810E7 add     esp, 18h
.text:000810EA pop     esi
.text:000810EB retn
.text:000810EB _printf endp
.text:000810EB
```

```

_Check_return_opt_
_CRT_STDIO_INLINE int __CRTDECL printf(
    _In_z_ _Printf_format_string_ char const* const _Format,
    ...)
#if defined _NO_CRT_STDIO_INLINE
;
#else
{
    int _Result;
    va_list _ArgList;
    __crt_va_start(_ArgList, _Format);
    _Result = _vfprintf_l(stdout, _Format, NULL, _ArgList);
    __crt_va_end(_ArgList);
    return _Result;
}
#endif

_Check_return_opt_
_CRT_STDIO_INLINE int __CRTDECL _printf_s_l(
    _In_z_ _Printf_format_string_params_(0) char const* const _Format,
    _In_opt_                                 _locale_t   const _Locale,
    ...)
#if defined _NO_CRT_STDIO_INLINE
;
#else
{
    int _Result;
```

Source path mappings

Sometimes the source code corresponding to the binary may be available but in a different location from what is recorded in the debug info (e.g. you may be debugging the binary on a different machine or even remotely from a different OS). The files which were not found in expected location are printed in Output window:

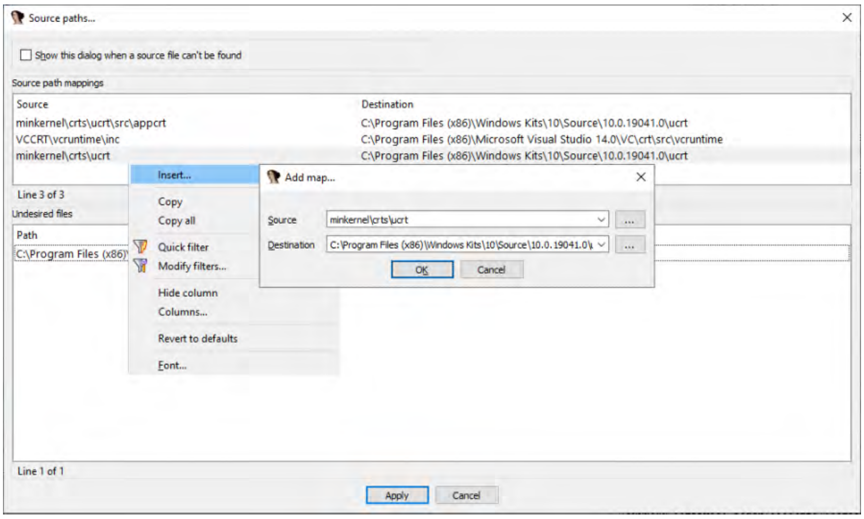
#85: Source-level debugging

15 Apr 2022

https://hex-rays.com/blog/igors-tip-of-the-week-85-source-level-debugging/

```
warning: read_file failed: source file not found: minkernel\crts\ucrt\inc\corecrt_internal_stdio_output.h
warning: read_file failed: source file not found: minkernel\crts\ucrt\inc\corecrt_internal_stdio_output.h
warning: read_file failed: source file not found: minkernel\crts\ucrt\inc\corecrt_internal_stdio_output.h
warning: read_file failed: source file not found: minkernel\crts\ucrt\inc\corecrt_internal_stdio_output.h
warning: read_file failed: source file not found: minkernel\crts\ucrt\inc\corecrt_internal_stdio_output.h
```

Using Options > Source paths..., you can set up mappings for IDA to find the source files in new locations.



Source	Destination
minkernel\crts\ucrt\src\appcrt	C:\Program Files (x86)\Windows Kits\10\Source\10.0.19041.0\ucrt
VCCRT\vcruntime\inc	C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\crt\src\vcruntime
minkernel\crts\ucrt	C:\Program Files (x86)\Windows Kits\10\Source\10.0.19041.0\ucrt

Line 3 of 3

Undesired files

Path
C:\Program Files (x86)\

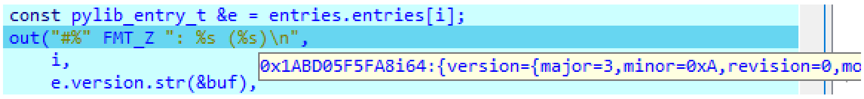
Line 1 of 1

Source: minkernel\crts\ucrt

Destination: C:\Program Files (x86)\Windows Kits\10\Source\10.0.19041.0\

Locals

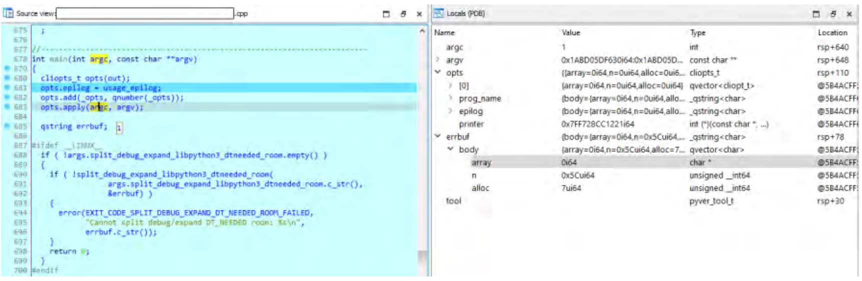
When the debug info includes information about local variables, IDA can use it to show their values. A short, one-line version is shown when you hover mouse over the variables in source code view.



```
const pylib_entry_t &e = entries.entries[i];
out("#%" FMT_Z ": %s (%s)\n",
      i,
      e.version.str(&buf),
```

```
0x1ABD05F5FA8i64:{version={major=3,minor=0xA,revision=0,mo
```

For more complex objects it may be more convenient to open the dedicated view where you can expand and inspect fields and sub-objects. This view is available via the menu Debugger > Debugger windows > Locals.

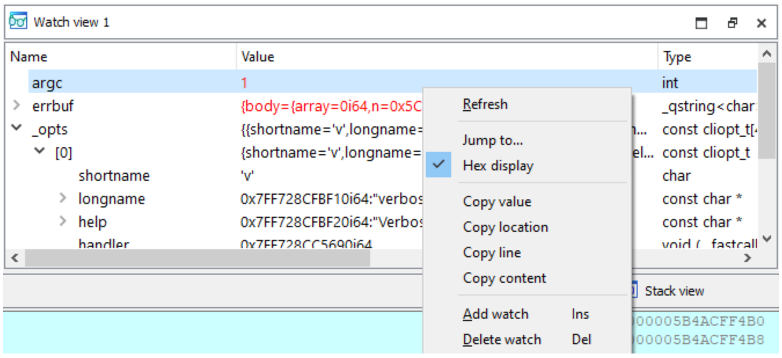


```
  ;

//----------------------------------------------------------------------
int main(int argc, const char **argv)
{
  cliopts_t opts(out);
  opts.epilog = usage_epilog;
  opts.add(_opts, qnumber(_opts));
  opts.apply(argc, argv);

  qstring errbuf;

#ifdef __LINUX__
  if ( !args.split_debug_expand_libpython3_dtneeded_room.empty() )
  {
    if ( !split_debug_expand_libpython3_dtneeded_room(
                args.split_debug_expand_libpython3_dtneeded_room.c_str(),
                &errbuf) )
    {
      error(EXIT_CODE_SPLIT_DEBUG_EXPAND_DT_NEEDED_ROOM_FAILED,
            "Cannot split debug/expand DT_NEEDED room: %s\n",
            errbuf.c_str());
    }
    return 0;
  }
#endif
```

Name	Value	Type	Location
argc	1	int	rsp+640
argv	0x1ABD05DF630i64:0x1ABD05D...	const char **	rsp+648
opts	{{array=0i64,n=0ui64,alloc=0ui6...	cliopts_t	rsp+110
[0]	{array=0i64,n=0ui64,alloc=0ui64}	qvector<cliopt_t>	@5B4ACFF
prog_name	{body={array=0i64,n=0ui64,allo...	_qstring<char>	@5B4ACFF
epilog	{body={array=0i64,n=0ui64,allo...	_qstring<char>	@5B4ACFF
printer	0x7FF728CC1221i64	int (*)(const char *, ...)	@5B4ACFF
errbuf	{body={array=0i64,n=0x5Cui64,...	_qstring<char>	rsp+78
body	{array=0i64,n=0x5Cui64,alloc=7...	qvector<char>	@5B4ACFF
array	0i64	char *	@5B4ACFF
n	0x5Cui64	unsigned __int64	@5B4ACFF
alloc	7ui64	unsigned __int64	@5B4ACFF
tool		pyver_tool_t	rsp+30

Watches

In addition to locals, you can also watch only specific variables you need instead of all locals, or values of global variables. This view can be opened via Debugger > Debugger windows > Watch view. Variables can be added using `Ins` or context menu.



Name	Value	Type
argc	1	int
errbuf	{body={array=0i64,n=0x5C	_qstring<char
_opts	{{shortname='v',longname=...n...	const cliopt_t[
[0]	{shortname='v',longname=...el...	const cliopt_t
shortname	'v'	char
longname	0x7FF728CFBF10i64:"verbos	const char *
help	0x7FF728CFBF20i64:"Verbos	const char *
handler	0x7FF728CC5690i64	void (__fastcall

```
00005B4ACFF4B0
00005B4ACFF4B8
```

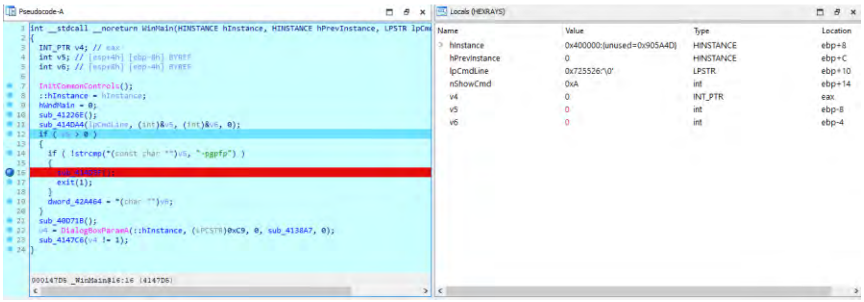
#85: Source-level debugging

15 Apr 2022

https://hex-rays.com/blog/igors-tip-of-the-week-85-source-level-debugging/

Debugging pseudocode

Even if you don't have debug info for the code you're debugging but do have a decompiler, it is possible to debug pseudocode as if it were a source file. IDA will automatically use pseudocode if source-level debugging is enabled but there is no debug info for the specific code fragment you're stepping through. You can also always switch to pseudocode during debugging using the usual `Tab` hotkey. Locals, Watches, and source-level breakpoints are available when debugging pseudocode in the same way as with "real" source code.



```
int __stdcall __noreturn WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCm
{
  INT_PTR v4; // eax
  int v5; // [esp+4h] [ebp-8h] BYREF
  int v6; // [esp+8h] [ebp-4h] BYREF

  InitCommonControls();
  ::hInstance = hInstance;
  hWndMain = 0;
  sub_41226E();
  sub_414DA4(lpCmdLine, (int)&v5, (int)&v6, 0);
  if ( v5 > 0 )
  {
    if ( !strcmp(*(const char **)v5, "-pgpfp") )
    {
      sub_414D3F();
      exit(1);
    }
    dword_42A464 = *(char **)v6;
  }
  sub_40D71B();
  v4 = DialogBoxParamA(::hInstance, (LPCSTR)0xC9, 0, sub_4138A7, 0);
  sub_4147C6(v4 != 1);
}

000147D6 _WinMain@16:16 (4147D6)
```

Name	Value	Type	Location
hInstance	0x400000:{unused=0x905A4D}	HINSTANCE	ebp+8
hPrevInstance	0	HINSTANCE	ebp+C
lpCmdLine	0x725526:'\0'	LPSTR	ebp+10
nShowCmd	0xA	int	ebp+14
v4	0	INT_PTR	eax
v5	0	int	ebp-8
v6	0	int	ebp-4

P.S. attentive reader may discover an additional surprise in this post. Happy Easter!

[1] https://hex-rays.com/blog/igors-tip-of-the-week-55-using-debug-symbols

#86: Function chunks

22 Apr 2022

https://hex-rays.com/blog/igors-tip-of-the-week-86-function-chunks/

In IDA, function is a sequence of instructions grouped together. Usually it corresponds to a high-level function or subroutine[1]:

1. it can be called from other places in the program, usually using a dedicated processor instruction;
2. it has an entry and one or more exits (instruction(s) which return to the caller);
3. it can accept arguments (in registers or on the stack) and optionally return values;
4. it can use local (stack) variables.

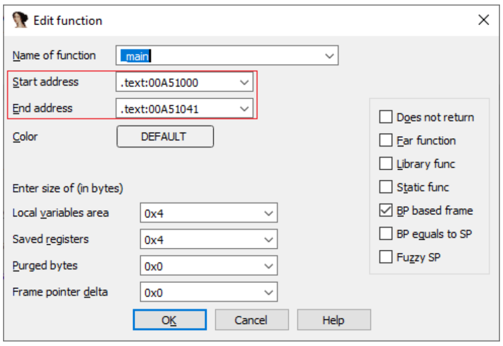
However, IDA's functions can group any arbitrary sequence of instructions, even those not matching the above criteria. The only hard requirement is that the function must start with a valid instruction.

Creating functions

IDA usually creates functions automatically, based on the call instructions or debug information, but they can also be created manually using the Create Function action (under Edit > Functions or from context menu), or `P` shortcut. This can be done only for instructions not already belonging to functions. By default IDA follows the cross-references and tries to determine the function boundaries automatically, but you can also select[2] a range beforehand to force creation of a function, for example, if there are some invalid instructions or embedded data.

Function range

In the most common case, a function occupies a contiguous address range, from the entry to the last return instruction. This is the start and end address specified in function properties available via the Edit Function dialog (`Alt– P`).



Name of function: main

Start address: .text:00A51000

End address: .text:00A51041

Color: DEFAULT

Enter size of (in bytes)

Local variables area: 0x4

Saved registers: 0x4

Purged bytes: 0x0

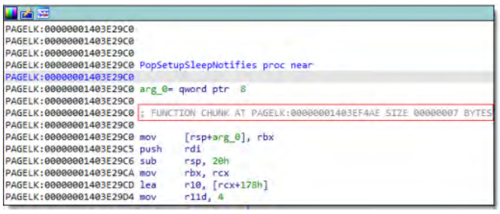
Frame pointer delta: 0x0

□ Does not return
□ Far function
□ Library func
□ Static func
☑ BP based frame
□ BP equals to SP
□ Fuzzy SP

Chunked functions

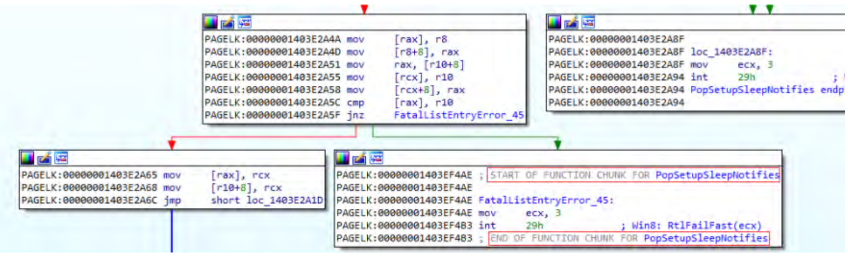
A single-range function is not the only option supported by IDA. In real-life programs, a function may be split into several disjoint ranges. For example, this may happen as a result of profile-guided optimization[3], which can put cold (rarely executed) basic blocks into a separate part of binary from hot (often executed) ones. In IDA, such functions are considered to consist of multiple chunks (each chunk being a single contiguous range of instructions). The chunk containing the function entry is known as entry chunk, while the others are called tail chunks or simply tails.

In disassembly view, the functions which have additional chunks have additional annotations near the function's entry, listing the tail chunks which belong to the function.



```
PAGELK:00000001403E29C0
PAGELK:00000001403E29C0
PAGELK:00000001403E29C0
PAGELK:00000001403E29C0 PopSetupSleepNotifies proc near
PAGELK:00000001403E29C0
PAGELK:00000001403E29C0 arg_0= qword ptr  8
PAGELK:00000001403E29C0
PAGELK:00000001403E29C0 ; FUNCTION CHUNK AT PAGELK:00000001403EF4AE SIZE 00000007 BYTES
PAGELK:00000001403E29C0
PAGELK:00000001403E29C0 mov     [rsp+arg_0], rbx
PAGELK:00000001403E29C5 push    rdi
PAGELK:00000001403E29C6 sub     rsp, 20h
PAGELK:00000001403E29CA mov     rbx, rcx
PAGELK:00000001403E29CD lea     r10, [rcx+178h]
PAGELK:00000001403E29D4 mov     r11d, 4
```

The tail chunks themselves are marked with "START OF FUNCTION CHUNK" and "END OF FUNCTION CHUNK" annotations, mentioning which function they belong to. This is mostly useful in text view, as in the graph view they are displayed as part of the overall function graph.



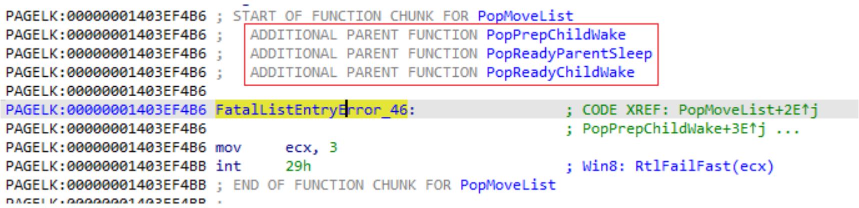
```
PAGELK:00000001403E2A4A mov     [rax], r8
PAGELK:00000001403E2A4D mov     [r8+8], rax
PAGELK:00000001403E2A51 mov     rax, [r10+8]
PAGELK:00000001403E2A55 mov     [rcx], r10
PAGELK:00000001403E2A58 mov     [rcx+8], rax
PAGELK:00000001403E2A5C cmp     [rax], r10
PAGELK:00000001403E2A5F jnz     FatalListEntryError_45
```

```
PAGELK:00000001403E2A8F
PAGELK:00000001403E2A8F loc_1403E2A8F:
PAGELK:00000001403E2A8F mov     ecx, 3
PAGELK:00000001403E2A94 int     29h                 ;
PAGELK:00000001403E2A94 PopSetupSleepNotifies endp
PAGELK:00000001403E2A94
```

```
PAGELK:00000001403E2A65 mov     [rax], rcx
PAGELK:00000001403E2A68 mov     [r10+8], rcx
PAGELK:00000001403E2A6C jmp     short loc_1403E2A1D
```

```
PAGELK:00000001403EF4AE ; START OF FUNCTION CHUNK FOR PopSetupSleepNotifies
PAGELK:00000001403EF4AE
PAGELK:00000001403EF4AE FatalListEntryError_45:
PAGELK:00000001403EF4AE mov     ecx, 3
PAGELK:00000001403EF4B3 int     29h                 ; Win8: RtlFailFast(ecx)
PAGELK:00000001403EF4B3 ; END OF FUNCTION CHUNK FOR PopSetupSleepNotifies
```

Sometimes a tail chunk may be shared by multiple functions. In that case, one of them is designated tail owner and others are considered additional parents. Such chunk will appear in the function graph for every function it belongs to.

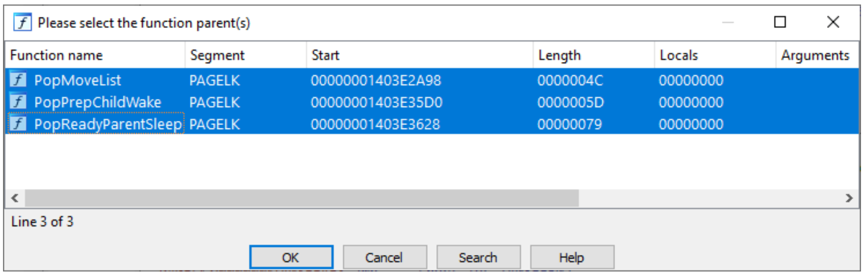


```
PAGELK:00000001403EF4B6 ; START OF FUNCTION CHUNK FOR PopMoveList
PAGELK:00000001403EF4B6 ;   ADDITIONAL PARENT FUNCTION PopPrepChildWake
PAGELK:00000001403EF4B6 ;   ADDITIONAL PARENT FUNCTION PopReadyParentSleep
PAGELK:00000001403EF4B6 ;   ADDITIONAL PARENT FUNCTION PopReadyChildWake
PAGELK:00000001403EF4B6
PAGELK:00000001403EF4B6 FatalListEntryError_46:                   ; CODE XREF: PopMoveList+2E↑j
PAGELK:00000001403EF4B6                                           ; PopPrepChildWake+3E↑j ...
PAGELK:00000001403EF4B6 mov     ecx, 3
PAGELK:00000001403EF4BB int     29h                               ; Win8: RtlFailFast(ecx)
PAGELK:00000001403EF4BB ; END OF FUNCTION CHUNK FOR PopMoveList
PAGELK:00000001403EF4BB ;
```

Managing chunks manually

Usually IDA handles chunked functions automatically, either detecting them during autoanalysis or by making use of other function range metadata (such as `.pdata` function descriptors in x64 PE files, or debug information). However, there may be situations where you need to add or remove chunks manually, for example to fix a false positive or handle an unusual compiler optimization.

To remove (detach) a tail chunk, position cursor inside it and invoke Edit > Functions > Remove function tail. If the tail has only one owner, it will be removed immediately and converted to standalone instructions (not belonging to any function). If it has multiple owners, IDA will offer you to choose from which function(s) it should be detached.



Please select the function parent(s)

Function name	Segment	Start	Length	Locals	Arguments
PopMoveList	PAGELK	00000001403E2A98	0000004C	00000000	
PopPrepChildWake	PAGELK	00000001403E35D0	0000005D	00000000	
PopReadyParentSleep	PAGELK	00000001403E3628	00000079	00000000	

Line 3 of 3

To add a range of instructions as a tail to a function, select the range and invoke Edit > Functions > Append function tail, then select a function to which it should be added. This can be done multiple times to attach a tail to several functions (whole tail must be selected again in such case).

More info:
IDA Help: Append Function Tail[4]
IDA Help: Remove Function Tail[5]

[1] https://en.wikipedia.org/wiki/Subroutine
[2] https://hex-rays.com/blog/igor-tip-of-the-week-03-selection-in-ida/
[3] https://devblogs.microsoft.com/cppblog/profile-guided-optimization-pgo-under-the-hood/
[4] https://www.hex-rays.com/products/ida/support/idadoc/687.shtml
[5] https://www.hex-rays.com/products/ida/support/idadoc/688.shtml

#87: Function chunks and the decompiler

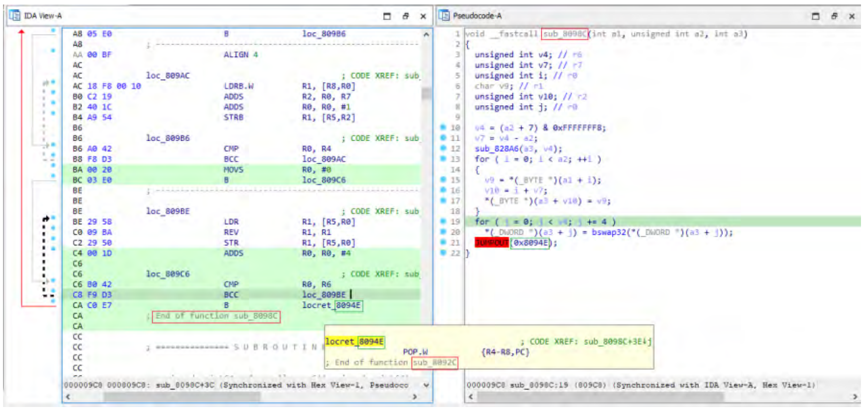
29 Apr 2022

https://hex-rays.com/blog/igors-tip-of-the-week-87-function-chunks-and-the-decompiler/

We've covered function chunks last week[1] and today we'll show an example of how to use them in practice to handle a common compiler optimization.

Shared function tail optimization

When working with some ARM firmware, you may sometimes run into the following situation:



```
A8 05 E0                B               loc_809B6
A8              ; ---------------------------------------------------------
AA 00 BF                ALIGN 4
AC
AC              loc_809AC                               ; CODE XREF: sub_
AC 18 F8 00 10          LDRB.W          R1, [R8,R0]
B0 C2 19                ADDS            R2, R0, R7
B2 40 1C                ADDS            R0, R0, #1
B4 A9 54                STRB            R1, [R5,R2]
B6
B6              loc_809B6                               ; CODE XREF: sub_
B6 A0 42                CMP             R0, R4
B8 F8 D3                BCC             loc_809AC
BA 00 20                MOVS            R0, #0
BC 03 E0                B               loc_809C6
BE              ; ---------------------------------------------------------
BE
BE              loc_809BE                               ; CODE XREF: sub_
BE 29 58                LDR             R1, [R5,R0]
C0 09 BA                REV             R1, R1
C2 29 50                STR             R1, [R5,R0]
C4 00 1D                ADDS            R0, R0, #4
C6
C6              loc_809C6                               ; CODE XREF: sub_
C6 B0 42                CMP             R0, R6
C8 F9 D3                BCC             loc_809BE
CA C0 E7                B               locret_8094E
CA              ; End of function sub_8098C
CA
CC
CC              ; =============== S U B R O U T I N
CC
CC
000009C8 000809C8: sub_8098C+3C (Synchronized with Hex View-1, Pseudoco
```

```
locret_8094E                            ; CODE XREF: sub_8098C+3E↓j
                POP.W           {R4-R8,PC}
; End of function sub_8092C
```

```
void __fastcall sub_8098C(int a1, unsigned int a2, int a3)
{
  unsigned int v4; // r6
  unsigned int v7; // r7
  unsigned int i; // r0
  char v9; // r1
  unsigned int v10; // r2
  unsigned int j; // r0

  v4 = (a2 + 7) & 0xFFFFFFF8;
  v7 = v4 - a2;
  sub_828A6(a3, v4);
  for ( i = 0; i < a2; ++i )
  {
    v9 = *(_BYTE *)(a1 + i);
    v10 = i + v7;
    *(_BYTE *)(a3 + v10) = v9;
  }
  for ( j = 0; j < v4; j += 4 )
    *(_DWORD *)(a3 + j) = bswap32(*(_DWORD *)(a3 + j));
  JUMPOUT(0x8094E);
}
000009C8 sub_8098C:19 (809C8) (Synchronized with IDA View-A, Hex View-1)
```

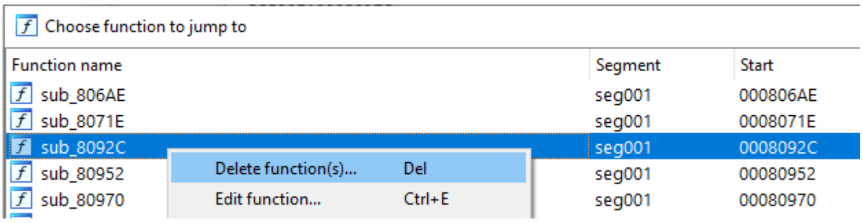
We have decompilation of `sub_8098C` which ends with a strange `JUMPOUT` statement and if we look at the disassembly, we can see that it corresponds to a branch to a `POP.W` instruction in **another function** (`sub_8092C`). What happened here?

This is an example of a code size optimization. The `POP.W` instruction is 4 bytes long, while the `B` branch is only two, so by reusing it the compiler saves two bytes. It may not sound like much, but such savings can accumulate to something substantial over all functions of the binary. Also, sometimes longer sequences of several instructions may be reused, leading to bigger savings.

Can we fix the database to get clean decompilation and get rid of `JUMPOUT`? Of course, the answer is yes, but the specific steps may be not too obvious, so let's describe some approaches.

Creating a chunk for the shared tail instructions

First we need to create a chunk for the shared instructions (in our example, the `POP.W` instruction). A chunk can be created only from instructions which do not yet belong to any function, thus the easiest way is to delete the function so that instructions become "free". This can be done either from the Functions window, via Edit > Functions > Delete function menu entry, or from the modal "jump to function" list (`Ctrl– P`, `Del`).

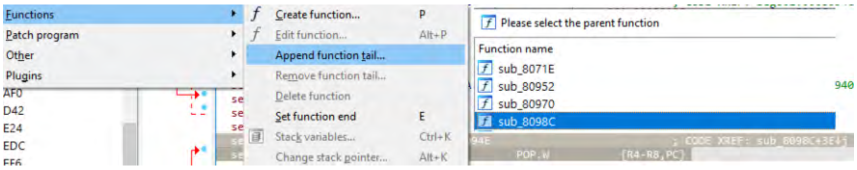


Choose function to jump to

Function name	Segment	Start
sub_806AE	seg001	000806AE
sub_8071E	seg001	0008071E
sub_8092C	seg001	0008092C
sub_80952	seg001	00080952
sub_80970	seg001	00080970

Once deleted, the shared tail instructions can be added as a chunk to the other function. This can be done manually:

1. select the instruction(s),
2. invoke Edit > Functions > Append function tail...
3. pick the referencing function (in our case, `sub_8098C`). Normally IDA should suggest it automatically.



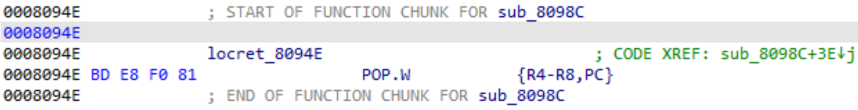
Please select the parent function

Function name
sub_8071E
sub_80952
sub_80970
sub_8098C

Or (semi)automatically:

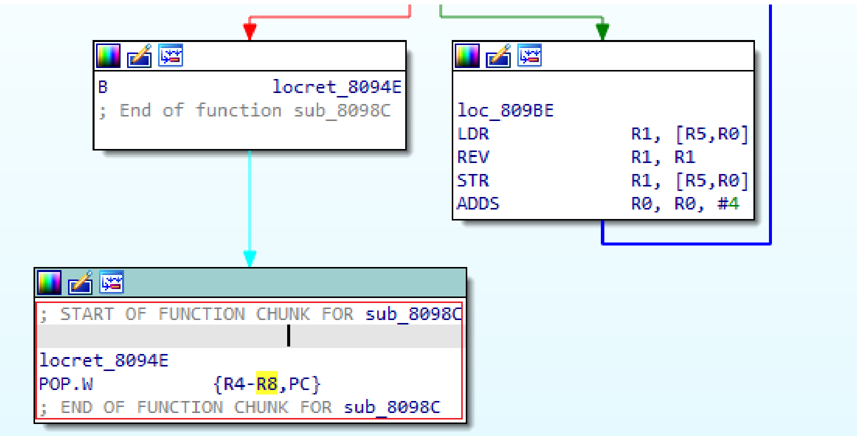
1. jump to the referencing branch (e.g. by double-clicking the `CODE XREF: sub_8098C+3E↓j` comment)
2. reanalyze[2] the branch (press `C`). IDA will detect that execution continues outside the current function bounds and automatically create and add the chunk for the shared tail instructions.

Either solution will create the chunk and mark it as belonging to the referencing function.



```
0008094E             ; START OF FUNCTION CHUNK FOR sub_8098C
0008094E
0008094E             locret_8094E                          ; CODE XREF: sub_8098C+3E↓j
0008094E BD E8 F0 81                 POP.W           {R4-R8,PC}
0008094E             ; END OF FUNCTION CHUNK FOR sub_8098C
```

We can check that it is contained in the function graph:

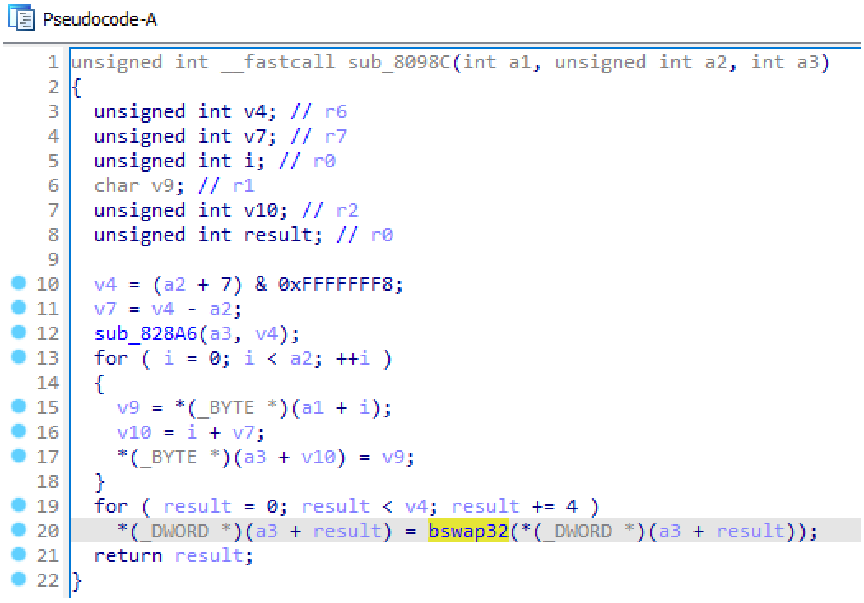


```
B                    locret_8094E
; End of function sub_8098C
```

```
loc_809BE
LDR             R1, [R5,R0]
REV             R1, R1
STR             R1, [R5,R0]
ADDS            R0, R0, #4
```

```
; START OF FUNCTION CHUNK FOR sub_8098C

locret_8094E
POP.W           {R4-R8,PC}
; END OF FUNCTION CHUNK FOR sub_8098C
```

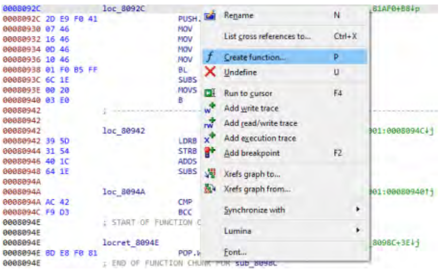
And the pseudocode no longer has a JUMPOUT:



```
unsigned int __fastcall sub_8098C(int a1, unsigned int a2, int a3)
{
  unsigned int v4; // r6
  unsigned int v7; // r7
  unsigned int i; // r0
  char v9; // r1
  unsigned int v10; // r2
  unsigned int result; // r0

  v4 = (a2 + 7) & 0xFFFFFFF8;
  v7 = v4 - a2;
  sub_828A6(a3, v4);
  for ( i = 0; i < a2; ++i )
  {
    v9 = *(_BYTE *)(a1 + i);
    v10 = i + v7;
    *(_BYTE *)(a3 + v10) = v9;
  }
  for ( result = 0; result < v4; result += 4 )
    *(_DWORD *)(a3 + result) = bswap32(*(_DWORD *)(a3 + result));
  return result;
}
```

Attaching the chunk to the original function

We “solved” the problem for one function, but in the process we’ve destroyed the function which contained the shared tail. If we need to decompile it too, we can try to recreate it:

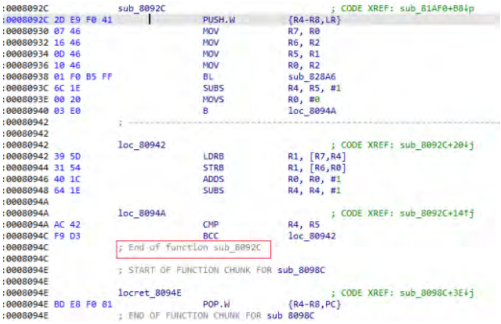


```
0008092C             loc_8092C                                 81AF0+B8↓p
0008092C 2D E9 F0 41             PUSH.
00080930 07 46                   MOV
00080932 16 46                   MOV
00080934 0D 46                   MOV
00080936 10 46                   MOV
00080938 01 F0 B5 FF             BL
0008093C 6C 1E                   SUBS
0008093E 00 20                   MOVS
00080940 03 E0                   B
00080942             ; ---------------
00080942
00080942             loc_80942                                 01:0008094C↓j
00080942 39 5D                   LDRB
00080944 31 54                   STRB
00080946 40 1C                   ADDS
00080948 64 1E                   SUBS
0008094A
0008094A             loc_8094A                                 01:00080940↑j
0008094A AC 42                   CMP
0008094C F9 D3                   BCC
0008094E             ; START OF FUNCTION C
0008094E
0008094E             locret_8094E                              8098C+3E↓j
0008094E BD E8 F0 81             POP.W
0008094E             ; END OF FUNCTION CHUNK FOR sub_8098C
```

29 Apr 2022

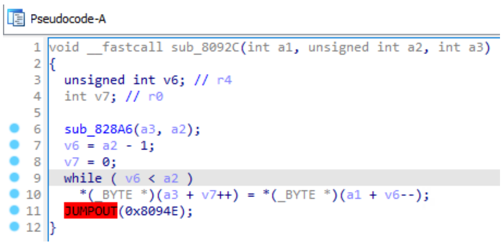
https://hex-rays.com/blog/igors-tip-of-the-week-87-function-chunks-and-the-decompiler/

However, IDA ends it before the chunk, because it's now a part of another function:



```
:0008092C             sub_8092C                             ; CODE XREF: sub_81AF0+B8↓p
:0008092C 2D E9 F0 41                 PUSH.W          {R4-R8,LR}
:00080930 07 46                       MOV             R7, R0
:00080932 16 46                       MOV             R6, R2
:00080934 0D 46                       MOV             R5, R1
:00080936 10 46                       MOV             R0, R2
:00080938 01 F0 B5 FF                 BL              sub_828A6
:0008093C 6C 1E                       SUBS            R4, R5, #1
:0008093E 00 20                       MOVS            R0, #0
:00080940 03 E0                       B               loc_8094A
:00080942             ; ---------------------------------------------------------------------------
:00080942
:00080942             loc_80942                             ; CODE XREF: sub_8092C+20↓j
:00080942 39 5D                       LDRB            R1, [R7,R4]
:00080944 31 54                       STRB            R1, [R6,R0]
:00080946 40 1C                       ADDS            R0, R0, #1
:00080948 64 1E                       SUBS            R4, R4, #1
:0008094A
:0008094A             loc_8094A                             ; CODE XREF: sub_8092C+14↑j
:0008094A AC 42                       CMP             R4, R5
:0008094C F9 D3                       BCC             loc_80942
:0008094C             ; End of function sub_8092C
:0008094C
:0008094E             ; START OF FUNCTION CHUNK FOR sub_8098C
:0008094E
:0008094E             locret_8094E                          ; CODE XREF: sub_8098C+3E↓j
:0008094E BD E8 F0 81                 POP.W           {R4-R8,PC}
:0008094E             ; END OF FUNCTION CHUNK FOR sub 8098C
```

And if we decompile it, we get the same `JUMPOUT` issue:

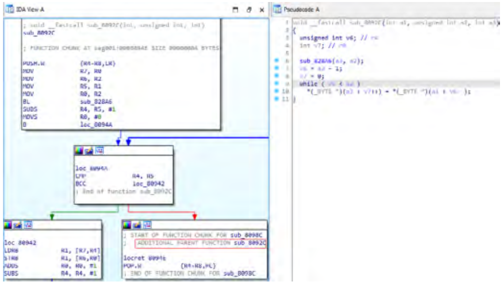


```
void __fastcall sub_8092C(int a1, unsigned int a2, int a3)
{
  unsigned int v6; // r4
  int v7; // r0

  sub_828A6(a3, a2);
  v6 = a2 - 1;
  v7 = 0;
  while ( v6 < a2 )
    *(_BYTE *)(a3 + v7++) = *(_BYTE *)(a1 + v6--);
  JUMPOUT(0x8094E);
}
```

The solution is simple: as mentioned in the previous post, a chunk may belong to multiple functions, so we just need to attach the chunk to this function too:

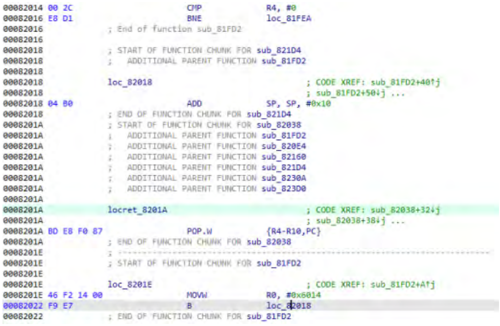
1. Select the instructions of the tail;
2. invoke Edit > Functions > Append function tail…
3. select the recreated function (in our example, sub_8092C).

The chunk gains one more owner, appears in the function graph, and the decompilation is fixed:


IDA View-A
Pseudocode-A
; START OF FUNCTION CHUNK FOR sub_8098C
; ADDITIONAL PARENT FUNCTION sub_8092C
locret_8094E
POP.W {R4-R8,PC}
; END OF FUNCTION CHUNK FOR sub_8098C

Complex situations

The above example had a tail shared by two functions, but of course this is not the limit. Consider this example:



```
00082014 00 2C                   CMP             R4, #0
00082016 E8 D1                   BNE             loc_81FEA
00082016             ; End of function sub_81FD2
00082016
00082018             ; START OF FUNCTION CHUNK FOR sub_821D4
00082018             ;   ADDITIONAL PARENT FUNCTION sub_81FD2
00082018
00082018             loc_82018                          ; CODE XREF: sub_81FD2+40↑j
00082018                                                ; sub_81FD2+50↓j ...
00082018 04 B0                   ADD             SP, SP, #0x10
00082018             ; END OF FUNCTION CHUNK FOR sub_821D4
0008201A             ; START OF FUNCTION CHUNK FOR sub_82038
0008201A             ;   ADDITIONAL PARENT FUNCTION sub_81FD2
0008201A             ;   ADDITIONAL PARENT FUNCTION sub_820E4
0008201A             ;   ADDITIONAL PARENT FUNCTION sub_82160
0008201A             ;   ADDITIONAL PARENT FUNCTION sub_821D4
0008201A             ;   ADDITIONAL PARENT FUNCTION sub_8230A
0008201A             ;   ADDITIONAL PARENT FUNCTION sub_823D0
0008201A
0008201A             locret_8201A                       ; CODE XREF: sub_82038+32↓j
0008201A                                                ; sub_82038+38↓j ...
0008201A BD E8 F0 87             POP.W           {R4-R10,PC}
0008201A             ; END OF FUNCTION CHUNK FOR sub_82038
0008201E             ; ---------------------------------------------------------------------------
0008201E             ; START OF FUNCTION CHUNK FOR sub_81FD2
0008201E
0008201E             loc_8201E                          ; CODE XREF: sub_81FD2+A↑j
0008201E 46 F2 14 00             MOVW            R0, #0x6014
00082022 F9 E7                   B               loc_82018
00082022             ; END OF FUNCTION CHUNK FOR sub_81FD2
```

Here, the `POP.W` instruction is shared by seven functions, and two of them also reuse the `ADD SP, SP, #0x10` instruction preceding it. There is also a chunk which belongs only to one function but it had to be separated because the function was no longer contiguous. Still, IDA's approach to fragmented functions was flexible enough to handle it with some manual help and all involved functions have proper control flow graphs and nice decompilation.

To summarize, the suggested algorithm of handling shared tail optimization is as follows:

1. Delete the function containing the shared tail instructions.
2. Attach the shared tail instructions to the other function(s) (manually or by reanalyzing the branches to the tail).
3. Recreate the deleted function and attach the shared tail(s) to it too.

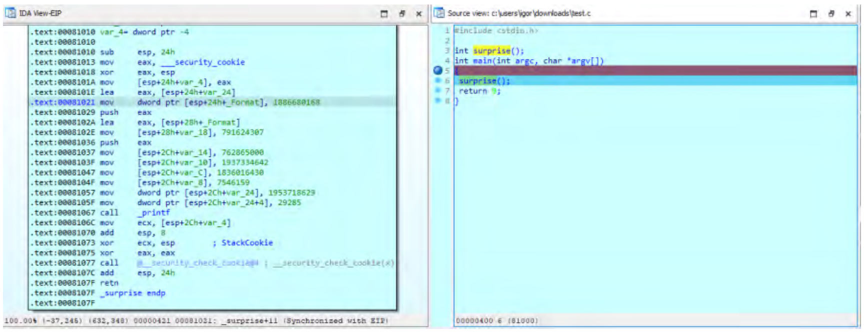
[1] https://hex-rays.com/blog/igors-tip-of-the-week-86-function-chunks/

#88: Character operand type and stack strings

06 May 2022
https://hex-rays.com/blog/igors-tip-of-the-week-88-character-operand-type-and-stack-strings/

We've mentioned operand representation[1] before but today we'll use a specific one to find the Easter egg hidden in the post #85[2].

More specifically, it was this screenshot:



```
.text:00081010 var_4= dword ptr -4
.text:00081010
.text:00081010 sub     esp, 24h
.text:00081013 mov     eax, ___security_cookie
.text:00081018 xor     eax, esp
.text:0008101A mov     [esp+24h+var_4], eax
.text:0008101E lea     eax, [esp+24h+var_24]
.text:00081021 mov     dword ptr [esp+24h+_Format], 1886680168
.text:00081029 push    eax
.text:0008102A lea     eax, [esp+28h+_Format]
.text:0008102E mov     [esp+28h+var_18], 791624307
.text:00081036 push    eax
.text:00081037 mov     [esp+2Ch+var_14], 762865000
.text:0008103F mov     [esp+2Ch+var_10], 1937334642
.text:00081047 mov     [esp+2Ch+var_C], 1836016430
.text:0008104F mov     [esp+2Ch+var_8], 7546159
.text:00081057 mov     dword ptr [esp+2Ch+var_24], 1953718629
.text:0008105F mov     dword ptr [esp+2Ch+var_24+4], 29285
.text:00081067 call    _printf
.text:0008106C mov     ecx, [esp+2Ch+var_4]
.text:00081070 add     esp, 8
.text:00081073 xor     ecx, esp        ; StackCookie
.text:00081075 xor     eax, eax
.text:00081077 call    @__security_check_cookie@4 ; __security_check_cookie(x)
.text:0008107C add     esp, 24h
.text:0008107F retn
.text:0008107F _surprise endp
.text:0008107F
100.00% (-37,245) (632,348) 00000421 00081021: _surprise+11 (Synchronized with EIP)
```

```
#include <stdio.h>

int surprise();
int main(int argc, char *argv[])
{
 surprise();
 return 0;
}
00000400 6 (81000)
```

The function `surprise` calls `printf`, but the arguments being passed to it seem to all be numbers. Doesn't `printf()` usually work with strings? What's going on?

Numbers and characters

As you probably know, computers do not actually distinguish numbers from characters – to them they're all just a set of bits. So it's all a matter of *interpretation* or *representation*. For example, all of the following are represented by the same bit pattern:

1. `65` (decimal number)
2. `0x41`, `41h`, `H'41` (hexadecimal number)
3. `0101` or `101o` (octal number)
4. `1000001b` or `0b1000001` (binary number)
5. `'A'` (ASCII character)
6. `WM_COMPACTING` (Win32 API constant)
7. (and many other variations)

Character operand representation

In fact, listing in the screenshot has been modified from the defaults to make the Easter egg less obvious. Here's the original version as text:

```
.text:00401010 ; int surprise(...)
.text:00401010 _surprise proc near                     ; CODE XREF: _main↑p
.text:00401010
.text:00401010 var_24= dword ptr -24h
.text:00401010 var_20= dword ptr -20h
.text:00401010 _Format= byte ptr -1Ch
.text:00401010 var_18= dword ptr -18h
.text:00401010 var_14= dword ptr -14h
.text:00401010 var_10= dword ptr -10h
.text:00401010 var_C= dword ptr -0Ch
.text:00401010 var_8= dword ptr -8
.text:00401010 var_4= dword ptr -4
.text:00401010
.text:00401010 sub     esp, 24h
.text:00401013 mov     eax, ___security_cookie
.text:00401018 xor     eax, esp
.text:0040101A mov     [esp+24h+var_4], eax
.text:0040101E lea     eax, [esp+24h+var_24]
.text:00401021 mov     dword ptr [esp+24h+_Format], 70747468h
.text:00401029 push    eax
.text:0040102A lea     eax, [esp+28h+_Format]
.text:0040102E mov     [esp+28h+var_18], 2F2F3A73h
.text:00401036 push    eax                             ; _Format
.text:00401037 mov     [esp+2Ch+var_14], 2D786568h
.text:0040103F mov     [esp+2Ch+var_10], 73796172h
.text:00401047 mov     [esp+2Ch+var_C], 6D6F632Eh
.text:0040104F mov     [esp+2Ch+var_8], 73252Fh
```

#88: Character operand type and stack strings

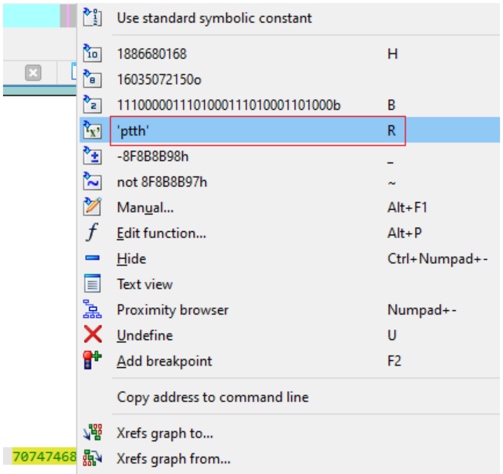
06 May 2022

https://hex-rays.com/blog/igors-tip-of-the-week-88-character-operand-type-and-stack-strings/

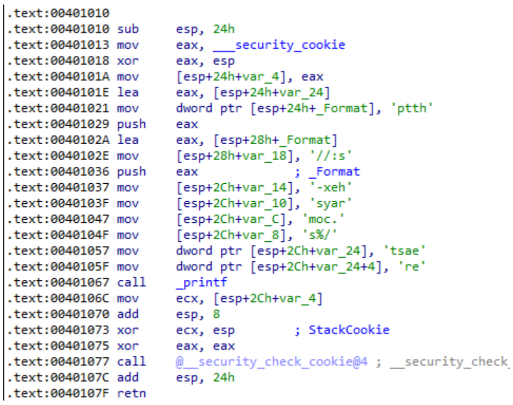
```
.text:00401057 mov     [esp+2Ch+var_24], 74736165h
.text:0040105F mov     [esp+2Ch+var_20], 7265h
.text:00401067 call    _printf
.text:0040106C mov     ecx, [esp+2Ch+var_4]
.text:00401070 add     esp, 8
.text:00401073 xor     ecx, esp                       ; StackCookie
.text:00401075 xor     eax, eax
.text:00401077 call    @__security_check_cookie@4      ; __security_check_cookie(x)
.text:0040107C add     esp, 24h
.text:0040107F retn
.text:0040107F _surprise endp
```

In hexadecimal it's almost immediately obvious: the "numbers" are actually short fragments of ASCII text. The code is building strings on the stack piece by piece. This can be made more explicit by converting numbers to the character operand type (shortcut `R`).

To help you decide whether such operand type makes sense, IDA shows a preview in the context menu:



This way it's pretty clear that the "number" is actually a text fragment. After converting all "numbers" to character constant, a pattern begins to emerge:



```
.text:00401010
.text:00401010 sub     esp, 24h
.text:00401013 mov     eax, ___security_cookie
.text:00401018 xor     eax, esp
.text:0040101A mov     [esp+24h+var_4], eax
.text:0040101E lea     eax, [esp+24h+var_24]
.text:00401021 mov     dword ptr [esp+24h+_Format], 'ptth'
.text:00401029 push    eax
.text:0040102A lea     eax, [esp+28h+_Format]
.text:0040102E mov     [esp+28h+var_18], '//:s'
.text:00401036 push    eax             ; _Format
.text:00401037 mov     [esp+2Ch+var_14], '-xeh'
.text:0040103F mov     [esp+2Ch+var_10], 'syar'
.text:00401047 mov     [esp+2Ch+var_C], 'moc.'
.text:0040104F mov     [esp+2Ch+var_8], 's%/'
.text:00401057 mov     dword ptr [esp+2Ch+var_24], 'tsae'
.text:0040105F mov     dword ptr [esp+2Ch+var_24+4], 're'
.text:00401067 call    _printf
.text:0040106C mov     ecx, [esp+2Ch+var_4]
.text:00401070 add     esp, 8
.text:00401073 xor     ecx, esp        ; StackCookie
.text:00401075 xor     eax, eax
.text:00401077 call    @__security_check_cookie@4 ; __security_check
.text:0040107C add     esp, 24h
.text:0040107F retn
```

Due to the little-endian memory organization of the x86 processor family, the individual fragments have to be read backwards (i.e. character literal `'ptth'` corresponds to the string fragment "`http`").

Decompiler and optimized string operations

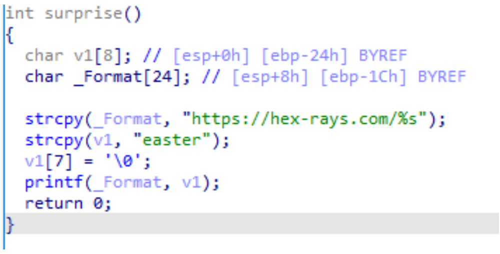
Now it's almost obvious what the result is supposed to be but there's in fact an even easier way to discover it.

Because the approach of processing short strings in register-sized chunks is often used by compilers to implement common C runtime functions inline instead of calling the library function, the decompiler uses heuristics to detect such code patterns and show them as equivalent function calls again. If we decompile this function, the decompiler reassembles the strings and shows them as if they were like that in the pseudocode:

#88: Character operand type and stack strings

06 May 2022

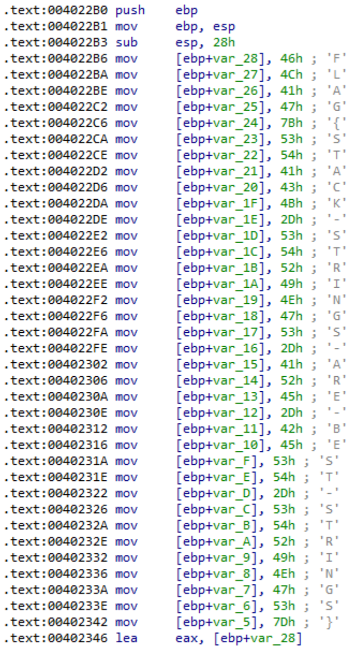
https://hex-rays.com/blog/igors-tip-of-the-week-88-character-operand-type-and-stack-strings/



```
int surprise()
{
  char v1[8]; // [esp+0h] [ebp-24h] BYREF
  char _Format[24]; // [esp+8h] [ebp-1Ch] BYREF

  strcpy(_Format, "https://hex-rays.com/%s");
  strcpy(v1, "easter");
  v1[7] = '\0';
  printf(_Format, v1);
  return 0;
}
```

Stack strings

Malware often uses a similar approach of building strings by small pieces (most often character by character) on the stack because this way the complete string does not appear in the binary and can’t be discovered by simply searching for it. Thanks to the automatic comments shown by IDA for operands not having explicitly assigned type, they are usually obvious in the disassembly:



```
.text:004022B0 push    ebp
.text:004022B1 mov     ebp, esp
.text:004022B3 sub     esp, 28h
.text:004022B6 mov     [ebp+var_28], 46h ; 'F'
.text:004022BA mov     [ebp+var_27], 4Ch ; 'L'
.text:004022BE mov     [ebp+var_26], 41h ; 'A'
.text:004022C2 mov     [ebp+var_25], 47h ; 'G'
.text:004022C6 mov     [ebp+var_24], 7Bh ; '{'
.text:004022CA mov     [ebp+var_23], 53h ; 'S'
.text:004022CE mov     [ebp+var_22], 54h ; 'T'
.text:004022D2 mov     [ebp+var_21], 41h ; 'A'
.text:004022D6 mov     [ebp+var_20], 43h ; 'C'
.text:004022DA mov     [ebp+var_1F], 4Bh ; 'K'
.text:004022DE mov     [ebp+var_1E], 2Dh ; '-'
.text:004022E2 mov     [ebp+var_1D], 53h ; 'S'
.text:004022E6 mov     [ebp+var_1C], 54h ; 'T'
.text:004022EA mov     [ebp+var_1B], 52h ; 'R'
.text:004022EE mov     [ebp+var_1A], 49h ; 'I'
.text:004022F2 mov     [ebp+var_19], 4Eh ; 'N'
.text:004022F6 mov     [ebp+var_18], 47h ; 'G'
.text:004022FA mov     [ebp+var_17], 53h ; 'S'
.text:004022FE mov     [ebp+var_16], 2Dh ; '-'
.text:00402302 mov     [ebp+var_15], 41h ; 'A'
.text:00402306 mov     [ebp+var_14], 52h ; 'R'
.text:0040230A mov     [ebp+var_13], 45h ; 'E'
.text:0040230E mov     [ebp+var_12], 2Dh ; '-'
.text:00402312 mov     [ebp+var_11], 42h ; 'B'
.text:00402316 mov     [ebp+var_10], 45h ; 'E'
.text:0040231A mov     [ebp+var_F], 53h ; 'S'
.text:0040231E mov     [ebp+var_E], 54h ; 'T'
.text:00402322 mov     [ebp+var_D], 2Dh ; '-'
.text:00402326 mov     [ebp+var_C], 53h ; 'S'
.text:0040232A mov     [ebp+var_B], 54h ; 'T'
.text:0040232E mov     [ebp+var_A], 52h ; 'R'
.text:00402332 mov     [ebp+var_9], 49h ; 'I'
.text:00402336 mov     [ebp+var_8], 4Eh ; 'N'
.text:0040233A mov     [ebp+var_7], 47h ; 'G'
.text:0040233E mov     [ebp+var_6], 53h ; 'S'
.text:00402342 mov     [ebp+var_5], 7Dh ; '}'
.text:00402346 lea     eax, [ebp+var_28]
```

And the decompiler easily recovers the complete string:

```
void __noreturn start()
{
  char v0[36]; // [esp+0h] [ebp-28h] BYREF
  qmemcpy(v0, “FLAG{STACK-STRINGS-ARE-BEST-STRINGS}”, sizeof(v0));
  [...]
 }
```

P.S. If you want to play with the Easter egg binary and reproduce the results in this post, download it here:easter2022.zip[3]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-46-disassembly-operand-representation/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-85-source-level-debugging/
[3] https://hex-rays.com/wp-content/uploads/2022/05/easter2022.zip

#89: En masse operations

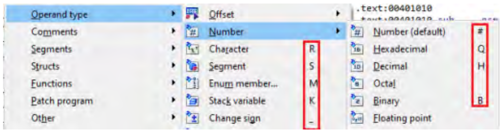
13 May 2022

https://hex-rays.com/blog/igors-tip-of-the-week-89-en-masse-operations/

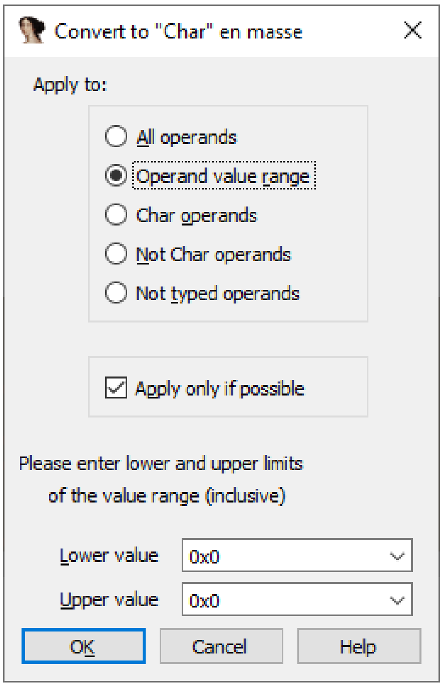
Last time we used operand types to make a function more readable and understand its behavior better. Converting operands one by one is fine if you need to do it a few times, but can quickly get tedious if you need to do it for a long piece of code.

En masse operation

To convert operands of several instruction at once, select them[1] before triggering the operation (either using the corresponding hotkey (e.g. R), or from the Edit > Operand type menu.



If you have a selection when triggering one of these actions, it won't be performed immediately but another dialog will pop up first:


Convert to "Char" en masse

Apply to:

○ All operands
● Operand value range
○ Char operands
○ Not Char operands
○ Not typed operands

☑ Apply only if possible

Please enter lower and upper limits of the value range (inclusive)

Lower value 0x0

Upper value 0x0

Here, you can tell IDA which operands you want to actually convert. The following options are available:

- All operands: all operands of selected instructions will be converted to the selected type (or back to the default/ number type if they already had the chosen type);
- Operand value range: only operands with values between Lower value and Upper value below will be converted. For example, you could enter '0x20' and '0x7F' to have IDA only consider single ASCII characters like the last example from the previous post[2];
- <type> operands: only convert operands which already have the selected type (they will be converted back to the default/number type);
- Not <type> operands: only convert operands not already having the selected type. Both untyped and having another type (e.g. decimal/enum/offset) operands will be converted to the desired type;
- Not typed operands: only convert operands not assigned a specific type (default/number). All operands already having an assigned type will be left as is.

P.S. you can use this feature not only with instructions but also data. For example, for converting several separate integers in the data section to decimal or octal. In such case, the 'operands' will be the data items.

See also: IDA Help: Perform en masse operation[3]

[1] https://hex-rays.com/blog/igor-tip-of-the-week-03-selection-in-ida/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-88-character-operand-type-and-stack-strings/
[3] https://hex-rays.com/products/ida/support/idadoc/459.shtml

#90: Suspicious operand limits

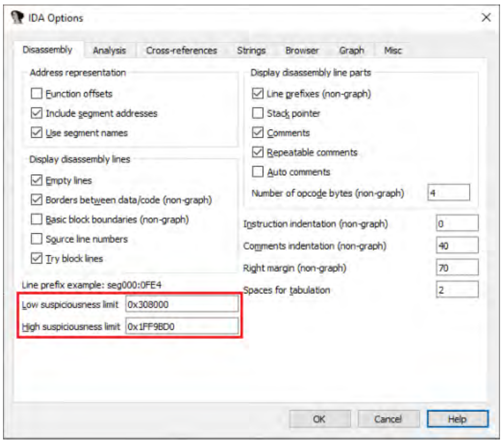
20 May 2022

https://hex-rays.com/blog/igors-tip-of-the-week-90-suspicious-operand-limits/

Although in general case the problem of correct disassembly is unsolvable, in practice it can get pretty close. IDA uses various heuristics to improve the disassembly and make it more readable, such as converting numerical values to offsets when it "looks plausible". However, this is not always reliable or successful and it may miss some. To help you improve things manually, in some cases IDA can give you a hint.

Suspiciousness Limits

In IDA's Options dialog on the Disassembly tab, there are two fields: Low suspiciousness limit and High suspiciousness limit. What do they mean?



Address representation
□ Function offsets
☑ Include segment addresses
☑ Use segment names

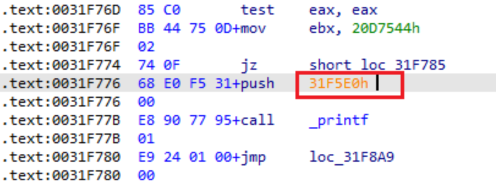
Display disassembly lines
☑ Empty lines
☑ Borders between data/code (non-graph)
□ Basic block boundaries (non-graph)
□ Source line numbers
☑ Try block lines

Line prefix example: seg000:0FE4
Low suspiciousness limit: 0x308000
High suspiciousness limit: 0x1FF9BD0

Display disassembly line parts
☑ Line prefixes (non-graph)
□ Stack pointer
☑ Comments
☑ Repeatable comments
□ Auto comments
Number of opcode bytes (non-graph): 4

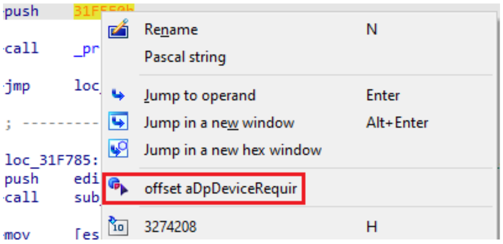
Instruction indentation (non-graph): 0
Comments indentation (non-graph): 40
Right margin (non-graph): 70
Spaces for tabulation: 2

Whenever IDA outputs an instruction operand with the numerical value in that range, and it does not yet have an explicitly set type (i.e. it has the default AKA void type), it will use a special color (orange in the default color scheme):



```
.text:0031F76D  85 C0       test    eax, eax
.text:0031F76F  BB 44 75 0D+mov     ebx, 20D7544h
.text:0031F76F  02
.text:0031F774  74 0F       jz      short loc_31F785
.text:0031F776  68 E0 F5 31+push    31F5E0h
.text:0031F776  00
.text:0031F77B  E8 90 77 95+call    _printf
.text:0031F77B  01
.text:0031F780  E9 24 01 00+jmp     loc_31F8A9
.text:0031F780  00
```

In such situation, you could, for example, hover your mouse[1] over the value to see if the target looks like a valid destination, and convert it to an offset either using a hotkey (O) or via the context menu.



```
push    31F5E0h
call    _pr
jmp     loc
; ---------
loc_31F785:
push    edi
call    sub
mov     [es
```

Changing the Suspiciousness Limits

Initial values of the limits are taken from the input file's loaded address range. If the valid address range changes (for example, if you rebase the database or create additional segments), it may make sense to update the ranges so you can see more of potential addresses. Conversely, you can also change the values to exclude some ranges which are unlikely to be valid addresses to reduce the false positives.

See also: IDA Help: Low & High Suspicious Operand Limits[2]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-47-hints-in-ida/

[2] https://www.hex-rays.com/products/ida/support/idadoc/606.shtml

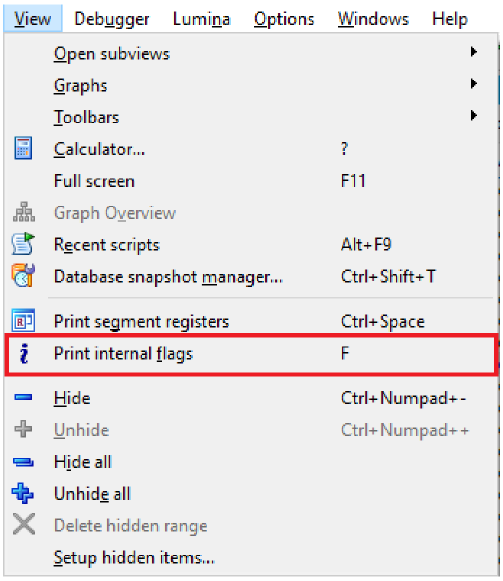
#91: Item flags

27 May 2022

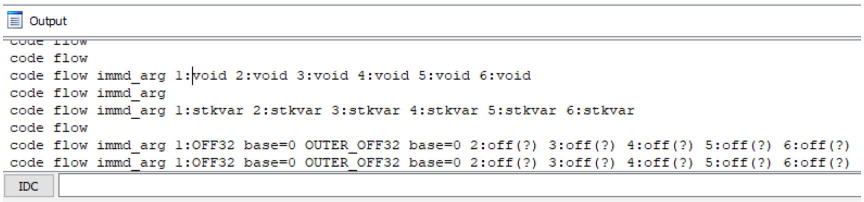
https://hex-rays.com/blog/igors-tip-of-the-week-91-item-flags/

When changing operand representation[1], you may need to check what are the operand types currently used by IDA for a specific instruction. In some cases it is obvious (e.g. for offset or character type), but the hex and default, for example, look exactly the same in most processors so it's not easy to tell them apart just by look.

Internally, this information is stored by IDA in the item flags. To check the current flags of an instruction (or any other address) in the database, use View > Print internal flags (hotkey `F`) .



When you invoke this action, IDA prints flags for the current address to the Output window. It only prints info about non-default operand types – the default ones are omitted (except for suspicious operands[2] which are printed as void).

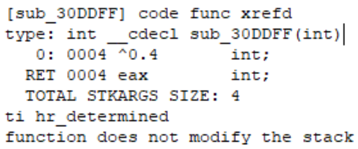


```
code flow
code flow
code flow immd_arg 1:void 2:void 3:void 4:void 5:void 6:void
code flow immd_arg
code flow immd_arg 1:stkvar 2:stkvar 3:stkvar 4:stkvar 5:stkvar 6:stkvar
code flow
code flow immd_arg 1:OFF32 base=0 OUTER_OFF32 base=0 2:off(?) 3:off(?) 4:off(?) 5:off(?) 6:off(?)
code flow immd_arg 1:OFF32 base=0 OUTER_OFF32 base=0 2:off(?) 3:off(?) 4:off(?) 5:off(?) 6:off(?)
```

code and *flow* are generic instruction flags: they mean that the current item is marked as code (instruction) and the execution reaches it from the previous address (this is the case for most instructions in the program).

Whenever IDA prints information about the second operand (number 1 since they are counted from 0), the operands 2,3...6 (even if they do not actually exist) are also printed as having the same type. This happens because of a limitation in IDA: it originally supported user-specified representation only for two operands (0 and 1) and this limitation is not completely lifted yet as of IDA 7.7.

Besides operand types, the feature may show other low-level info about the current address: for example, the type information if it's set for current location, the function arguments layout similarly to what you can see in decompiler annotations[3], structure name for structure data items, and so on.



```
[sub_30DDFF] code func xrefd
type: int __cdecl sub_30DDFF(int)
   0: 0004 ^0.4        int;
  RET 0004 eax         int;
  TOTAL STKARGS SIZE: 4
ti hr_determined
function does not modify the stack
```

[1] https://hex-rays.com/blog/igors-tip-of-the-week-46-disassembly-operand-representation/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-77-mapped-variables/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-66-decompiler-annotations/

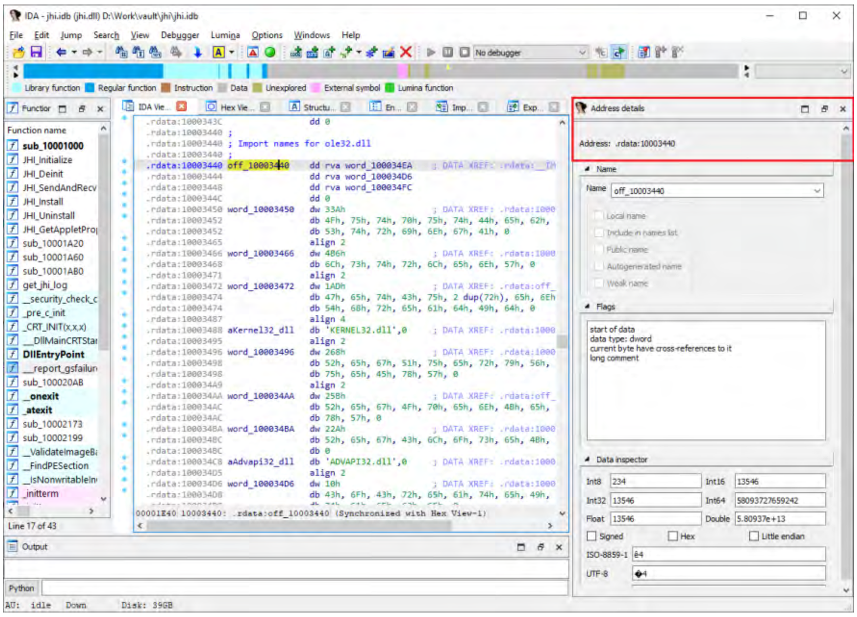
03 Jun 2022

https://hex-rays.com/blog/igors-tip-of-the-week-92-address-details/

The address details pane is a rather recent addition to IDA so probably not many users are familiar with it yet. However, it can be a quite useful addition to the standard workflow, permitting you to perform some common tasks faster.

Address details view

On invoking View > Open subview > Address details (you can also use the Quick view selector), a new pane appears, by default on the right side of the main window. Obviously, it can be moved and docked elsewhere if you prefer. It will automatically update itself using the current address in any address-based view (IDA View, Hex View, Pseudocode).


IDA - jhi.idb (jhi.dll) D:\Work\vault\jhi\jhi.idb
File Edit Jump Search View Debugger Lumina Options Windows Help
No debugger
Library function Regular function Instruction Data Unexplored External symbol Lumina function
Function name
IDA Vie... Hex Vie... Structu... En... Imp... Exp...
Output
Python
AU: idle Down Disk: 39GB

```
.rdata:1000343C                 dd 0
.rdata:10003440 ;
.rdata:10003440 ; Import names for ole32.dll
.rdata:10003440 ;
.rdata:10003440 off_10003440    dd rva word_100034EA    ; DATA XREF: .rdata:__IM
.rdata:10003444                 dd rva word_100034D6
.rdata:10003448                 dd rva word_100034FC
.rdata:1000344C                 dd 0
.rdata:10003450 word_10003450   dw 33Ah                 ; DATA XREF: .rdata:1000
.rdata:10003452                 db 4Fh, 75h, 74h, 70h, 75h, 74h, 44h, 65h, 62h,
.rdata:10003452                 db 53h, 74h, 72h, 69h, 6Eh, 67h, 41h, 0
.rdata:10003465                 align 2
.rdata:10003466 word_10003466   dw 486h                 ; DATA XREF: .rdata:1000
.rdata:10003468                 db 6Ch, 73h, 74h, 72h, 6Ch, 65h, 6Eh, 57h, 0
.rdata:10003471                 align 2
.rdata:10003472 word_10003472   dw 1ADh                 ; DATA XREF: .rdata:off_
.rdata:10003474                 db 47h, 65h, 74h, 43h, 75h, 2 dup(72h), 65h, 6Eh
.rdata:10003474                 db 54h, 68h, 72h, 65h, 61h, 64h, 49h, 64h, 0
.rdata:10003487                 align 4
.rdata:10003488 aKernel32_dll   db 'KERNEL32.dll',0     ; DATA XREF: .rdata:1000
.rdata:10003495                 align 2
.rdata:10003496 word_10003496   dw 268h                 ; DATA XREF: .rdata:1000
.rdata:10003498                 db 52h, 65h, 67h, 51h, 75h, 65h, 72h, 79h, 56h,
.rdata:10003498                 db 75h, 65h, 45h, 78h, 57h, 0
.rdata:100034A9                 align 2
.rdata:100034AA word_100034AA   dw 25Bh                 ; DATA XREF: .rdata:off_
.rdata:100034AC                 db 52h, 65h, 67h, 4Fh, 70h, 65h, 6Eh, 4Bh, 65h,
.rdata:100034AC                 db 78h, 57h, 0
.rdata:100034BA word_100034BA   dw 22Ah                 ; DATA XREF: .rdata:1000
.rdata:100034BC                 db 52h, 65h, 67h, 43h, 6Ch, 6Fh, 73h, 65h, 4Bh,
.rdata:100034BC                 db 0
.rdata:100034C8 aAdvapi32_dll   db 'ADVAPI32.dll',0     ; DATA XREF: .rdata:1000
.rdata:100034D5                 align 2
.rdata:100034D6 word_100034D6   dw 10h                  ; DATA XREF: .rdata:1000
.rdata:100034D8                 db 43h, 6Fh, 43h, 72h, 65h, 61h, 74h, 65h, 49h,
00001E40 10003440: .rdata:off_10003440 (Synchronized with Hex View-1)
```

Functions: sub_10001000, JHI_Initialize, JHI_Deinit, JHI_SendAndRecv, JHI_Install, JHI_Uninstall, JHI_GetAppletPro, sub_10001A20, sub_10001A60, sub_10001AB0, get_jhi_log, __security_check_c, _pre_c_init, _CRT_INIT(x,x,x), __DllMainCRTStar, DllEntryPoint, __report_gsfailur, sub_100020AB, _onexit, _atexit, sub_10002173, sub_10002199, _ValidateImageB, _FindPESection, _IsNonwritableIn, _initterm

Line 17 of 43

Address details

Address: .rdata:10003440

Name

Name: off_10003440

- □ Local name
- □ Include in names list
- □ Public name
- □ Autogenerated name
- □ Weak name

Flags

start of data
data type: dword
current byte have cross-references to it
long comment

Data inspector

Int8	234	Int16	13546
Int32	13546	Int64	58093727659242
Float	13546	Double	5.80937e+13

□ Signed □ Hex □ Little endian

ISO-8859-1: ê4

UTF-8: �4

The pane consists of three sections, each of which can be collapsed and expanded using the triangle icon in the top left corner.

Name section

This is basically a non-modal version of the standard Rename address dialog (`N` hotkey). It allows you to quickly rename locations by entering a new name in the edit box as well as view and change various name attributes.

Flags section

This is an expanded version of the Print internal flags[1] action, however currently it does not provide details of instruction operands.

Data inspector section

This section shows how the bytes at the current address can be interpreted in various formats: integers, floating-point values, or string literals. It can be especially useful when exploring unknown file formats or arrays of unknown data in programs.

[1] https://hex-rays.com/blog/igors-tip-of-the-week-91-item-flags/

#93: COM reverse engineering and COM Helper

10 Jun 2022

https://hex-rays.com/blog/igors-tip-of-the-week-93-com-reverse-engineering-and-com-helper/

COM aka Component Object Model is the technology used by Microsoft (and others) to create and use reusable software components in a manner independent from the specific language or vendor. It uses a stable and well-defined ABI which is mostly compatible with Microsoft C++ ABI, allowing easy implementation and usage of COM components in C++.

COM basics

COM components and their interfaces are identified by UUID aka GUID[1] – unique 128-bit IDs usually represented as string of several hexadecimal number groups. For example, `{00000000-0000-0000-C000-000000000046}` represents `IUnknown` – the base interface which must be implemented by any COM-conforming component.

Each COM interface provides a set of functions in a way similar to a C++ class. On the binary level, this is represented by a structure with function pointers, commonly named <name>Vtbl. For example, here's how the t is laid out:

```
struct IUnknownVtbl
{
  HRESULT (__stdcall *QueryInterface)(IUnknown *This, const IID *const riid, void **ppvObject);
  ULONG (__stdcall *AddRef)(IUnknown *This);
  ULONG (__stdcall *Release)(IUnknown *This);
};
struct IUnknown
{
  struct IUnknownVtbl *lpVtbl;
};
```

IDA's standard type libraries include most of the COM interfaces defined by the Windows SDKs, so you can import these structures from them. Here's how to do it manually:

1. Open the Structures window (`Shift– F9`);
2. Use "Add struct type..." from the context menu, or `Ins`;
3. Type the name of the interface and/or its vtable (e.g. `IUnknownVtbl`) and click OK. If the interface is known, it will be imported from the type library automatically. If you are not sure it is available, you can click "Add standard structure" and use incremental search (start typing the name) to check if it's present in the list of available types.

Once imported, the struct can be used, for example, to label indirect calls performed using the interface pointer.

How to know which interface is being used in the code? There are multiple ways it can be done, but one common approach is to use the `CoCreateInstance`[2] API. It returns a pointer to the interface defined by the interface ID (IID) which is a kind of GUID. You can check what IID is used, then search for it in Windows SDK headers and hopefully find the interface name.

For example, consider this call:

```
.text:30961A4D push    eax                             ; ppv
.text:30961A4E push    offset riid                     ; riid
.text:30961A53 push    1                               ; dwClsContext
.text:30961A55 push    esi                             ; pUnkOuter
.text:30961A56 push    offset rclsid                   ; rclsid
.text:30961A5B mov     [ebp+ppv], esi
.text:30961A5E call    ds:CoCreateInstance
```

If we follow riid, we can see that it's been formatted by IDA nicely as an instance of the IID structure:

```
.text:30961C18 riid dd 0EC5EC8A9h                           ; Data1
.text:30961C18                                         ; DATA XREF: sub_30961A2E+20↑o
.text:30961C18                                         ; sub_30DECD76+1D↓o
.text:30961C18 dw 0C395h                               ; Data2
.text:30961C18 dw 4314h                                ; Data3
.text:30961C18 db 9Ch, 77h, 54h, 0D7h, 0A9h, 35h, 0FFh, 70h; Data4
```

In the text form, this corresponds to `EC5EC8A9-C395-4314-9C77-54D7A935FF70`, but since it's quite awkward to convert from the struct representation, a quick way is to search for EC5EC8A9 and see if you can find a match.

There is one in `wincodec.h`:

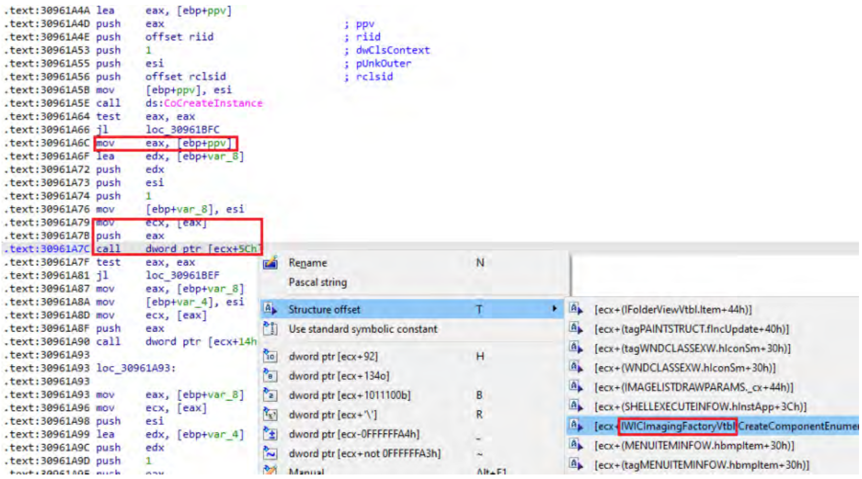
#93: COM reverse engineering and COM Helper

10 Jun 2022

https://hex-rays.com/blog/igors-tip-of-the-week-93-com-reverse-engineering-and-com-helper/

```
MIDL_INTERFACE(“ec5ec8a9-c395-4314-9c77-54d7a935ff70”)
IWICImagingFactory : public IUnknown
{
public:
    virtual HRESULT STDMETHODCALLTYPE CreateDecoderFromFilename(
        /* [in] */ __RPC__in LPCWSTR wzFilename,
        /* [unique][in] */ __RPC__in_opt const GUID *pguidVendor,
        /* [in] */ DWORD dwDesiredAccess,
        /* [in] */ WICDecodeOptions metadataOptions,
        /* [retval][out] */ __RPC__deref_out_opt IWICBitmapDecoder **ppIDecoder) = 0;
[....]
```

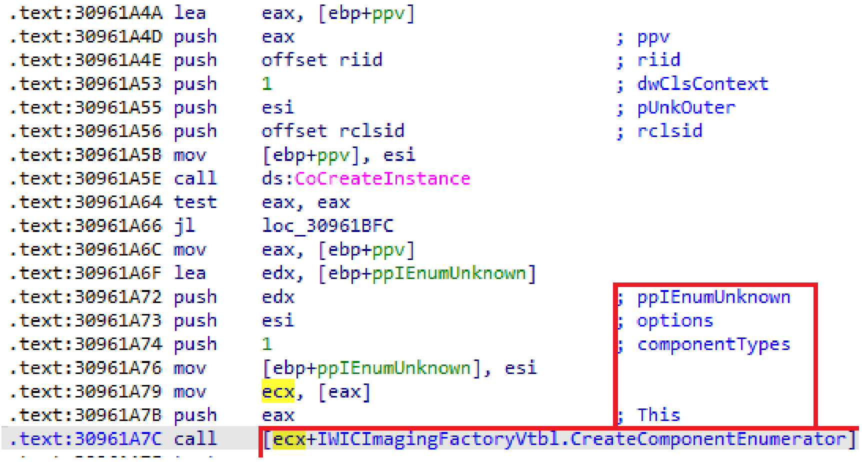
Now that we know we’re dealing with `IWICImagingFactory`, we can import `IWICImagingFactoryVtbl` and use it to label the calls made later by dereferencing the `ppv` variable:



```
.text:30961A4A lea     eax, [ebp+ppv]
.text:30961A4D push    eax                          ; ppv
.text:30961A4E push    offset riid                  ; riid
.text:30961A53 push    1                            ; dwClsContext
.text:30961A55 push    esi                          ; pUnkOuter
.text:30961A56 push    offset rclsid                ; rclsid
.text:30961A5B mov     [ebp+ppv], esi
.text:30961A5E call    ds:CoCreateInstance
.text:30961A64 test    eax, eax
.text:30961A66 jl      loc_30961BFC
.text:30961A6C mov     eax, [ebp+ppv]
.text:30961A6F lea     edx, [ebp+var_8]
.text:30961A72 push    edx
.text:30961A73 push    esi
.text:30961A74 push    1
.text:30961A76 mov     [ebp+var_8], esi
.text:30961A79 mov     ecx, [eax]
.text:30961A7B push    eax
.text:30961A7C call    dword ptr [ecx+5Ch]
.text:30961A7F test    eax, eax
.text:30961A81 jl      loc_30961BEF
.text:30961A87 mov     eax, [ebp+var_8]
.text:30961A8A mov     [ebp+var_4], esi
.text:30961A8D mov     ecx, [eax]
.text:30961A8F push    eax
.text:30961A90 call    dword ptr [ecx+14h
.text:30961A93
.text:30961A93 loc_30961A93:
.text:30961A93
.text:30961A93 mov     eax, [ebp+var_8]
.text:30961A96 mov     ecx, [eax]
.text:30961A98 push    esi
.text:30961A99 lea     edx, [ebp+var_4]
.text:30961A9C push    edx
.text:30961A9D push    1
```

Rename N
Pascal string
Structure offset T
Use standard symbolic constant
dword ptr [ecx+92] H
dword ptr [ecx+134o]
dword ptr [ecx+1011100b] B
dword ptr [ecx+'\'] R
dword ptr [ecx-0FFFFFFA4h] _
dword ptr [ecx+not 0FFFFFFA3h] ~
Manual Alt+F1
[ecx+(IFolderViewVtbl.Item+44h)]
[ecx+(tagPAINTSTRUCT.fIncUpdate+40h)]
[ecx+(tagWNDCLASSEXW.hIconSm+30h)]
[ecx+(WNDCLASSEXW.hIconSm+30h)]
[ecx+(IMAGELISTDRAWPARAMS._cx+44h)]
[ecx+(SHELLEXECUTEINFOW.hInstApp+3Ch)]
[ecx+IWICImagingFactoryVtbl.CreateComponentEnumer
[ecx+(MENUITEMINFOW.hbmpItem+30h)]
[ecx+(tagMENUITEMINFOW.hbmpItem+30h)]

IDA uses type information of the structure’s function pointer to label and propagate argument information[3]:

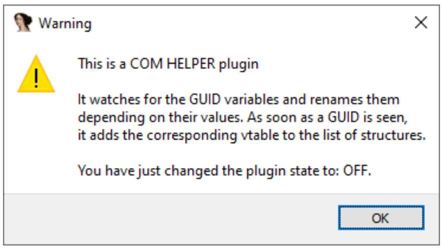


```
.text:30961A4A lea     eax, [ebp+ppv]
.text:30961A4D push    eax                          ; ppv
.text:30961A4E push    offset riid                  ; riid
.text:30961A53 push    1                            ; dwClsContext
.text:30961A55 push    esi                          ; pUnkOuter
.text:30961A56 push    offset rclsid                ; rclsid
.text:30961A5B mov     [ebp+ppv], esi
.text:30961A5E call    ds:CoCreateInstance
.text:30961A64 test    eax, eax
.text:30961A66 jl      loc_30961BFC
.text:30961A6C mov     eax, [ebp+ppv]
.text:30961A6F lea     edx, [ebp+ppIEnumUnknown]
.text:30961A72 push    edx                          ; ppIEnumUnknown
.text:30961A73 push    esi                          ; options
.text:30961A74 push    1                            ; componentTypes
.text:30961A76 mov     [ebp+ppIEnumUnknown], esi
.text:30961A79 mov     ecx, [eax]
.text:30961A7B push    eax                          ; This
.text:30961A7C call    [ecx+IWICImagingFactoryVtbl.CreateComponentEnumerator]
```

While this process works, it is somewhat tedious and error prone. Is there something better?

COM helper

IDA ships with a standard plugin which can automate some parts of the process. If you invoke Edit > Plugins > COM Helper, it shows a little help about what it does:



This is a COM HELPER plugin

It watches for the GUID variables and renames them depending on their values. As soon as a GUID is seen, it adds the corresponding vtable to the list of structures.

You have just changed the plugin state to: OFF.

Invoke the menu again to re-enable it. The default state is on for new databases, so normally you do not need to do that. With plugin enabled, we can do the following:

1. Undefine/delete the IID instance at `riid`.
2. Redefine it as a GUID (Alt-Q, choose “GUID”).

If the GUID is known, the instance is renamed to `CLSID_<name>`, and the corresponding `<name>Vtbl` is imported into the database automatically (if available in loaded type libraries). You can then use it to resolve the indirect calls from the interface pointer.

Extending the known interface list

To detect known GUIDs, on Windows the COM Helper uses the registry (`HKLM\Software\Classes\Interface` subtree). If the GUID is not found in registry (or not running on Windows), the file `cfg/clsid.cfg` in IDA’s install directory is consulted. It is a simple text file with the list of GUIDs and corresponding names. If you are dealing with lesser-known interfaces, you can add their GUIDs to this file so that they can be labeled nicely.

[1] https://en.wikipedia.org/wiki/Universally_unique_identifier
[2] https://docs.microsoft.com/en-us/windows/win32/api/combaseapi/nf-combaseapi-cocreateinstance
[3] https://hex-rays.com/blog/igors-tip-of-the-week-74-parameter-identification-and-tracking-pit/

#94: Variable-sized structures

17 Jun 2022

https://hex-rays.com/blog/igors-tip-of-the-week-94-variable-sized-structures/

Variable-sized structures is a construct used to handle binary structures of variable size with the advantage of compile-time type checking.

In source code

Usually such structures use a layout similar to following:

```
struct varsize_t
{
 // some fixed fields at the start
 int id;
 size_t datalen;
 //[more fields]
 unsigned char data[];// variable part
};
```

In other words, a fixed-layout part at the start and an array of unspecified size at the end.

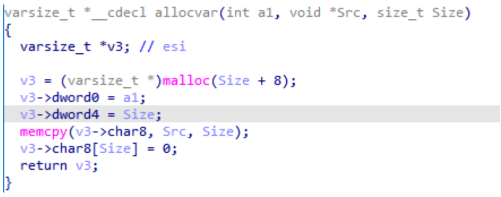
Some compilers do not like `[]` syntax so `[0]` or even `[1]` may be used too. At runtime, the space for the structure is allocated using the full size, and the array can be accessed as if it had expected size. For example:

```
struct varsize_t* allocvar(int id, void *data, size_t datalen);
{
 size_t fullsize = sizeof(varsize_t)+datalen+1;
 struct varsize_t *var = (struct varsize_t*) malloc(fullsize);
 var->id = id;
 var->datalen = datalen;
 memcpy(var->data, data, datalen);
 var->data[datalen]=0;
 return var;
}
```

Can such structs be handled by IDA? Yes, but there are some peculiarities you may need to be aware of.

In the decompiler

In the decompiler everything is pretty simple: just add the struct using C syntax to Local Types[1] and use it for types of local variables and function arguments. The decompiler automatically detects accesses to the variable part and represents them accordingly.



```
varsize_t *__cdecl allocvar(int a1, void *Src, size_t Size)
{
  varsize_t *v3; // esi

  v3 = (varsize_t *)malloc(Size + 8);
  v3->dword0 = a1;
  v3->dword4 = Size;
  memcpy(v3->char8, Src, Size);
  v3->char8[Size] = 0;
  return v3;
}
```

In disassembly

However, disassembly view is trickier. You can import the struct from Local Types to the IDB Structures, or create one manually by explicitly adding an array of 0 elements at the end:

```
00000000 varsize_t struc ; (sizeof=0x8, align=0x4, copyof_1, variable size)
00000000 id dd ?
00000004 datalen dd ?
00000008 data db 0 dup(?)
00000008 varsize_t ends
```

But when you have instances of such structs in data area, using this definition only covers the fixed part. To extend the struct, use * (Create/resize array action) and specify the full size of the struct.

Example

Recent Microsoft compilers add so-called “COFF group” info to the PE executables. It is currently not fully parsed by IDA but is labeled in the disassembly listing with the comment `IMAGE_DEBUG_TYPE_POGO`:

#94: Variable-sized structures

17 Jun 2022

https://hex-rays.com/blog/igors-tip-of-the-week-94-variable-sized-structures/

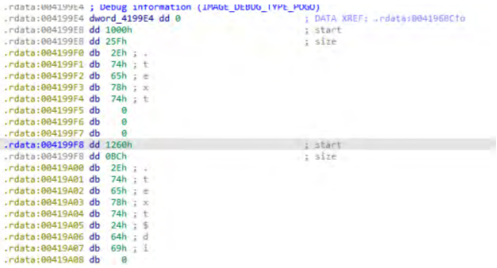
```
.rdata:004199E4 ; Debug information (IMAGE_DEBUG_TYPE_POGO)
.rdata:004199E4 dword_4199E4 dd 0 ; DATA XREF: .rdata:004196BC↑o
.rdata:004199E8 dd 1000h, 25Fh, 7865742Eh, 74h, 1260h, 0BCh, 7865742Eh, 69642474h, 0
.rdata:00419A0C dd 1320h, 11BE2h, 7865742Eh, 6E6D2474h, 0
.rdata:00419A20 dd 12F10h, 12Ch, 7865742Eh, 782474h, 13040h, 164h, 7865742Eh, 64792474h
.rdata:00419A20 dd 0
.rdata:00419A44 dd 14000h, 11Ch, 6164692Eh, 35246174h, 0
.rdata:00419A58 dd 1411Ch, 4, 6330302Eh, 6766h, 14120h, 4, 5452432Eh, 41435824h, 0
.rdata:00419A7C dd 14124h, 4, 5452432Eh, 41435824h, 41h, 14128h, 1Ch, 5452432Eh, 55435824h
```

On expanding the array or looking at the hex view, it becomes apparent that it stores info about the original section names of the executable, before they are merged by the linker. So it can be useful to format this info. It seems to consist of a list of following structures:

```
struct section_info
{
  int start; // RVA
  int size;
  char name[]; // zero-terminated
};
```

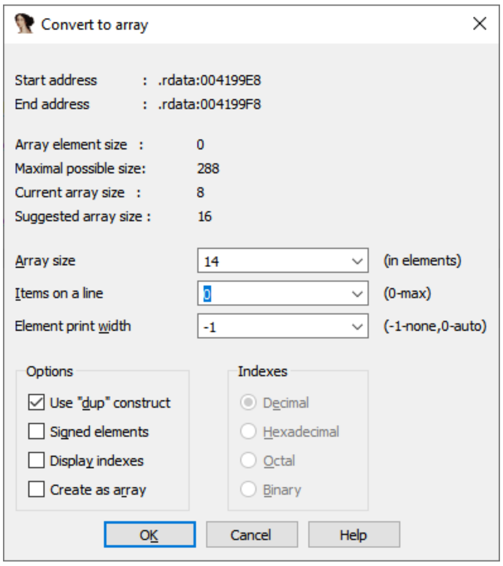
The string is padded with zeroes if necessary to align each struct on a 4-byte boundary.

After creating a local type and importing the struct to IDB, we can undefine the array created by IDA and start creating struct instances in the area using Edit > Struct var… (`Alt–Q`). However, only the fixed part is covered by default:



```
.rdata:004199E4 ; Debug information (IMAGE_DEBUG_TYPE_POGO)
.rdata:004199E4 dword_4199E4 dd 0                              ; DATA XREF: .rdata:004196BC↑o
.rdata:004199E8 dd 1000h                                       ; start
.rdata:004199E8 dd 25Fh                                        ; size
.rdata:004199F0 db  2Eh ; .
.rdata:004199F1 db  74h ; t
.rdata:004199F2 db  65h ; e
.rdata:004199F3 db  78h ; x
.rdata:004199F4 db  74h ; t
.rdata:004199F5 db    0
.rdata:004199F6 db    0
.rdata:004199F7 db    0
.rdata:004199F8 dd 1260h                                       ; start
.rdata:004199F8 dd 0BCh                                        ; size
.rdata:00419A00 db  2Eh ; .
.rdata:00419A01 db  74h ; t
.rdata:00419A02 db  65h ; e
.rdata:00419A03 db  78h ; x
.rdata:00419A04 db  74h ; t
.rdata:00419A05 db  24h ; $
.rdata:00419A06 db  64h ; d
.rdata:00419A07 db  69h ; i
.rdata:00419A08 db    0
```

To extend the struct, press * and enter full size. For example, the first one should be 14 (8 for the fixed part and 6 for ".text" and terminating zero), although you can also use the suggested 16:



Start address : .rdata:004199E8

End address : .rdata:004199F8

Array element size : 0

Maximal possible size: 288

Current array size : 8

Suggested array size : 16

Array size: 14 (in elements)

Items on a line: 0 (0-max)

Element print width: -1 (-1-none,0-auto)

☑ Use "dup" construct

☐ Signed elements

☐ Display indexes

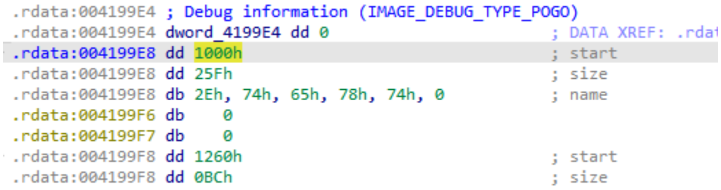
☐ Create as array

● Decimal

○ Hexadecimal

○ Octal

○ Binary



```
.rdata:004199E4 ; Debug information (IMAGE_DEBUG_TYPE_POGO)
.rdata:004199E4 dword_4199E4 dd 0                        ; DATA XREF: .rda
.rdata:004199E8 dd 1000h                                 ; start
.rdata:004199E8 dd 25Fh                                  ; size
.rdata:004199E8 db 2Eh, 74h, 65h, 78h, 74h, 0            ; name
.rdata:004199F6 db    0
.rdata:004199F7 db    0
.rdata:004199F8 dd 1260h                                 ; start
.rdata:004199F8 dd 0BCh                                  ; size
```

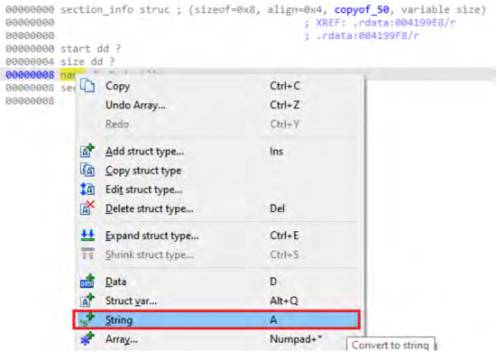
#94: Variable-sized structures

17 Jun 2022

https://hex-rays.com/blog/igors-tip-of-the-week-94-variable-sized-structures/

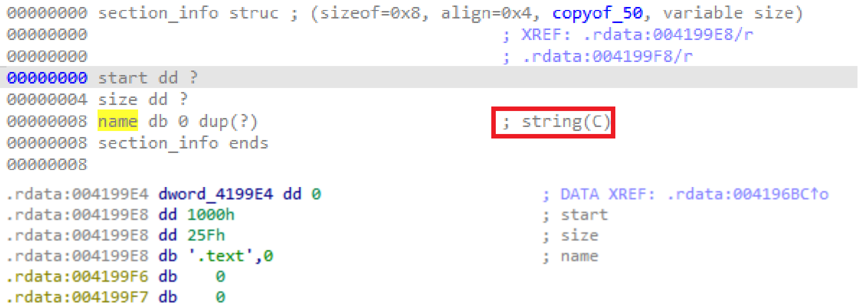
Now the struct has correct size and covers the string but it is printed as hex bytes and not text. Why and how to fix it?

When IDA converts C type to assembly-level (IDB) struct, it only relies on the sizes of C types, because on the assembly level there is no difference between a a byte and character[2]. Thus a char array is the same as a byte array. However, you can still apply additional representation flags to influence formatting of the structure. For example, you can go to the imported definition in Structures list and mark the `name` field as a string literal, either from context menu or by pressing `A`:



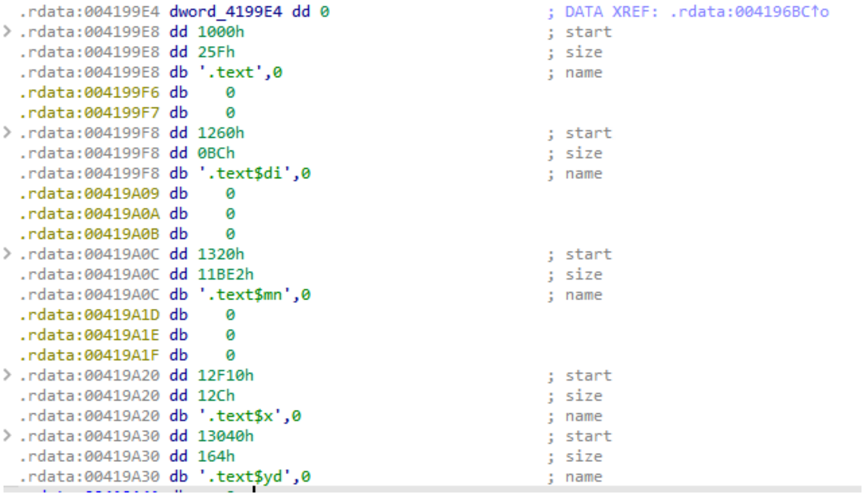
```
00000000 section_info struc ; (sizeof=0x8, align=0x4, copyof_50, variable size)
00000000                                         ; XREF: .rdata:004199E8/r
00000000                                         ; .rdata:004199F8/r
00000000 start dd ?
00000004 size dd ?
00000008 nar
00000008 se
00000008
```

The field is now commented correspondingly and the data instances show the string as text:



```
00000000 section_info struc ; (sizeof=0x8, align=0x4, copyof_50, variable size)
00000000                                         ; XREF: .rdata:004199E8/r
00000000                                         ; .rdata:004199F8/r
00000000 start dd ?
00000004 size dd ?
00000008 name db 0 dup(?)                        ; string(C)
00000008 section_info ends
00000008

.rdata:004199E4 dword_4199E4 dd 0                ; DATA XREF: .rdata:004196BC↑o
.rdata:004199E8 dd 1000h                         ; start
.rdata:004199E8 dd 25Fh                          ; size
.rdata:004199E8 db '.text',0                     ; name
.rdata:004199F6 db    0
.rdata:004199F7 db    0
```

In fact, once you mark the field as string, newly declared instances will be automatically sized by IDA using the zero terminator.



```
  .rdata:004199E4 dword_4199E4 dd 0              ; DATA XREF: .rdata:004196BC↑o
> .rdata:004199E8 dd 1000h                       ; start
  .rdata:004199E8 dd 25Fh                        ; size
  .rdata:004199E8 db '.text',0                   ; name
  .rdata:004199F6 db    0
  .rdata:004199F7 db    0
> .rdata:004199F8 dd 1260h                       ; start
  .rdata:004199F8 dd 0BCh                        ; size
  .rdata:004199F8 db '.text$di',0                ; name
  .rdata:00419A09 db    0
  .rdata:00419A0A db    0
  .rdata:00419A0B db    0
> .rdata:00419A0C dd 1320h                       ; start
  .rdata:00419A0C dd 11BE2h                      ; size
  .rdata:00419A0C db '.text$mn',0                ; name
  .rdata:00419A1D db    0
  .rdata:00419A1E db    0
  .rdata:00419A1F db    0
> .rdata:00419A20 dd 12F10h                      ; start
  .rdata:00419A20 dd 12Ch                        ; size
  .rdata:00419A20 db '.text$x',0                 ; name
> .rdata:00419A30 dd 13040h                      ; start
  .rdata:00419A30 dd 164h                        ; size
  .rdata:00419A30 db '.text$yd',0                ; name
```

See also:
Variable Length Structures Tutorial[3]
IDA Help: Convert to array[4]
IDA Help: Assembler level and C level types[5]
IDA Help: Structures window[6]

[1] https://hex-rays.com/blog/igor-tip-of-the-week-11-quickly-creating-structures/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-46-disassembly-operand-representation/
[3] https://hex-rays.com/products/ida/support/tutorials/varstr/
[4] https://www.hex-rays.com/products/ida/support/idadoc/455.shtml
[5] https://www.hex-rays.com/products/ida/support/idadoc/1042.shtml
[6] https://www.hex-rays.com/products/ida/support/idadoc/593.shtml

#95: Offsets

24 Jun 2022

https://hex-rays.com/blog/igors-tip-of-the-week-95-offsets/

As we've mentioned before[1], the same numerical value can be used represented in different ways even if it's the same bit pattern on the binary level. One of the representations used in IDA is offset.

Offsets

In IDA, an offset is a numerical value which is used as an address (either directly or as part of an expression) to refer to another location in the program.

The term comes from the keyword used in MASM (Microsoft Assembler) to distinguish an address expression from a variable.

For example:

```
mov eax, g_var1
```

Loads the value from the location `g_var1` into register `eax`. In C, this would be equivalent to using the variable's value.

While

```
mov eax, offset g_var1
```

Loads the address of the location `g_var1` into `eax`. In C, this would be equivalent to taking the variable's address.

On the binary level, the second instruction is equivalent to moving of a simple integer, e.g.:

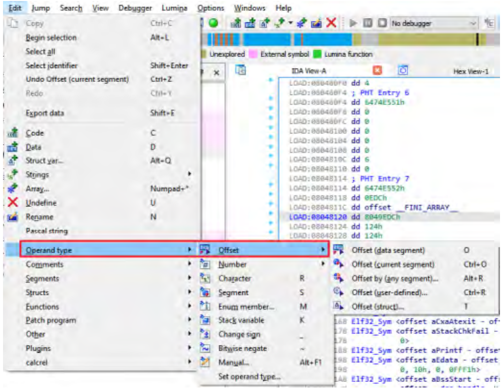
```
mov eax, 0x40002000
```

However, during analysis the offset form is obviously preferred, both for readability and because it allows you to see cross-references to variables and be able to quickly identify other places where the variable is used.

In general, distinguishing integer values used in instructions from addresses is impossible without whole program analysis or runtime tracing, but the majority of cases can be handled by relatively simple heuristics so usually IDA is able to recover offset expressions and add cross-references. However, in some cases they may fail or produce false positives so you may need to do it manually.

Converting values to offsets

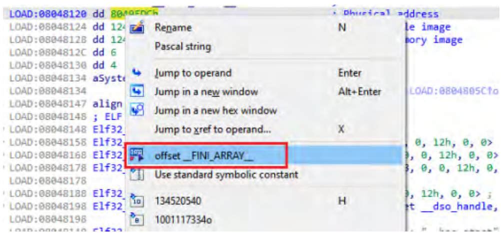
All options for converting to offsets are available under Edit > Operand type > Offset:



```
LOAD:080480F0 dd 4
LOAD:080480F4 ; PHT Entry 6
LOAD:080480F4 dd 6474E551h
LOAD:080480F8 dd 0
LOAD:080480FC dd 0
LOAD:08048100 dd 0
LOAD:08048104 dd 0
LOAD:08048108 dd 0
LOAD:0804810C dd 6
LOAD:08048110 dd 0
LOAD:08048114 ; PHT Entry 7
LOAD:08048114 dd 6474E552h
LOAD:08048118 dd 0EDCh
LOAD:0804811C dd offset __FINI_ARRAY__
LOAD:08048120 dd 8049EDCh
LOAD:08048124 dd 124h
LOAD:08048128 dd 124h
```

In most modern, flat-memory model binaries such as ELF, PE, Mach-O, the first two commands are equivalent, so you can usually use shortcut `O` or `Ctrl– O`.

The most common/applicable options are also shown in the context (right-click) menu:



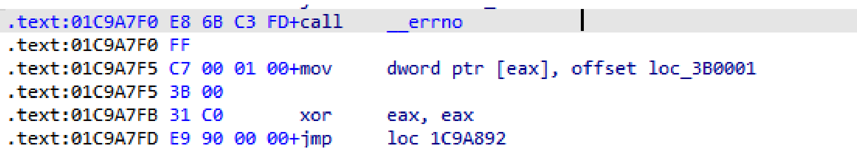
```
LOAD:08048120 dd 8049EDCh
LOAD:08048124 dd 12
LOAD:08048128 dd 12
LOAD:0804812C dd 6
LOAD:08048130 dd 4
LOAD:08048134 aSyst
LOAD:08048134
LOAD:08048147 align
LOAD:08048148 ; ELF
LOAD:08048148 Elf32_
LOAD:08048158 Elf32_
LOAD:08048168 Elf32_
LOAD:08048178 Elf32_
LOAD:08048178
LOAD:08048188 Elf32_
LOAD:08048198 Elf32_
LOAD:08048198
```

#95: Offsets

24 Jun 2022
https://hex-rays.com/blog/igors-tip-of-the-week-95-offsets/

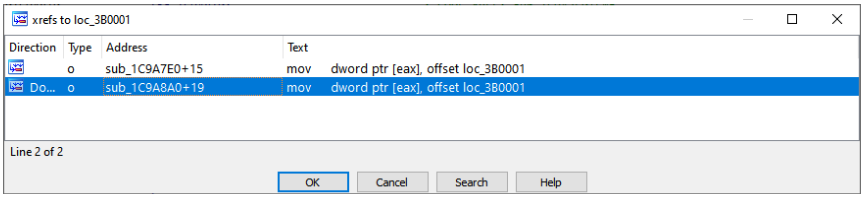
Fixing false positives

There may be cases when IDA's heuristics convert a value to an offset when it's not actually being used as one. One common example is bitwise operations done with values which happen to be in the range of the program's address space, but it can also happen for data values or simple data movement, like on the below screenshot.



```
.text:01C9A7F0 E8 6B C3 FD+call     __errno
.text:01C9A7F0 FF
.text:01C9A7F5 C7 00 01 00+mov      dword ptr [eax], offset loc_3B0001
.text:01C9A7F5 3B 00
.text:01C9A7FB 31 C0        xor     eax, eax
.text:01C9A7FD E9 90 00 00+jmp      loc 1C9A892
```

In this example, IDA has converted the second operand of the mov instruction to an offset because it turned out to match a program address. However, we can see that it is being moved into a location returned by the call to `__errno` function. This is a common way compilers implement setting of the errno pseudo-variable (which can be thread-specific instead of a global), so obviously that operand should be a number and not an offset. Besides being a wrong representation, this also lead to bogus cross-references:



xrefs to loc_3B0001

Direction	Type	Address	Text
	o	sub_1C9A7E0+15	mov dword ptr [eax], offset loc_3B0001
Do...	o	sub_1C9A8A0+19	mov dword ptr [eax], offset loc_3B0001

Line 2 of 2

You have the following options to fix the false positive:

1. Press `O` or `Ctrl–O` to reset the "offset" attribute of the operand and let IDA show the default representation (hex). Note that the number will be printed in orange to hint that its value falls into the address space of the program, i.e. it is suspicious[2];
2. Use `Q` / `*` (for hex), `H` (for decimal), or select the corresponding option from the context menu to explicitly mark the operand as a number and also avoid flagging it as suspicious;
3. If you have created an enumeration to represent such numbers as symbolic constants, you can use the `M` shortcut or the context menu to convert it to a symbolic constant.

See also:
IDA Help: Edit|Operand types|Offset submenu[3]
IDA Help: Edit|Operand types|Number submenu[4]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-46-disassembly-operand-representation/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-90-suspicious-operand-limits/
[3] https://www.hex-rays.com/products/ida/support/idadoc/1381.shtml
[4] https://www.hex-rays.com/products/ida/support/idadoc/1382.shtml

#96: Loading additional files

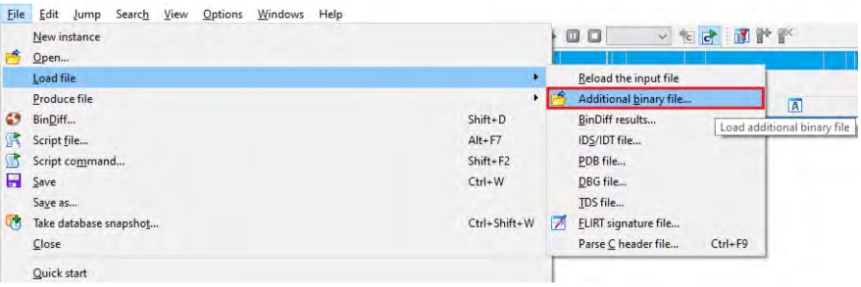
04 Jul 2022

https://hex-rays.com/blog/igors-tip-of-the-week-96-loading-additional-files/

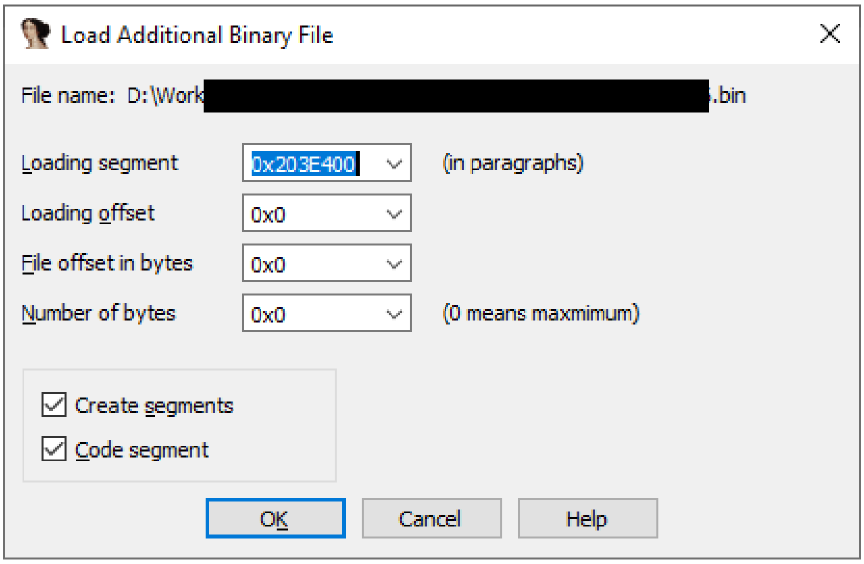
Although most of the time IDA is used to work on single, self-contained file (e.g. an executable, library, or a firmware image), this is not always the case. Sometimes the program may refer to or load additional files or data, and it may be useful to have that data in the database and analyze it together with the original file.

Load Additional Binary File

For simple cases where you have a raw binary file with the contents you want to add to the database, you can use File > Load file > Additional binary file…



Please note that any file you select will be treated as raw binary, even for formats otherwise supported by IDA (e.g. PE/ELF/Mach-O). Once you select a file, IDA will show you the dialog to specify where exactly you want to load it:



Load Additional Binary File

File name: D:\Work[redacted].bin

Loading segment: 0x203E400 (in paragraphs)

Loading offset: 0x0

File offset in bytes: 0x0

Number of bytes: 0x0 (0 means maxmimum)

☑ Create segments

☑ Code segment

OK Cancel Help

Loading segment and *Loading offset* together specify the location where you want to load the file's data. By default, IDA tries to pick the values which are located just after the end of the last segment of the database in such a way that the newly loaded data starts at offset 0 in the new segment. However, if you are working with **flat memory** layout binary such as the case with most of modern OSes, you should instead set the **segment** value to **0** and **offset** to the linear address where you need to have the data loaded.

File offset in bytes and *Number of bytes* specify what part of the file you need loaded. With the default values IDA will load the whole file from the beginning, but you can also change them to load only a part of it.

Create segments is checked by default because in most cases the file is being loaded into a new address range which does not exist in the database. If you've already created a segment for the file's data or plan to do it after loading the bytes, you can uncheck this checkbox.

See also:
IDA Help: Load Additional Binary File[1]
Igor's tip of the week #41: Binary file loader[2]
Several files in one IDB, part 3[3]

[1] https://www.hex-rays.com/products/ida/support/idadoc/1372.shtml
[2] https://hex-rays.com/blog/igors-tip-of-the-week-41-binary-file-loader/
[3] https://hex-rays.com/blog/several-files-in-one-idb-part-3/

#97: Cross reference depth

08 Jul 2022

https://hex-rays.com/blog/igors-tip-of-the-week-97-cross-reference-depth/

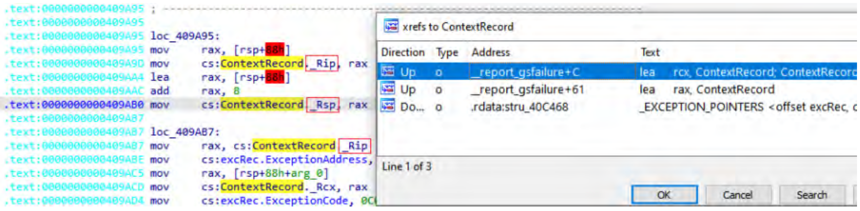
We have covered basic usage of cross-references before[1], but there are situations where they may not behave as you may expect.

Accessing large data items

If there is a large structure or an array and the code reads or writes data deep inside it, you may not see cross-references from that code listed at the structure definition.

Example

For example, in the Microsoft CRT function `__report_gsfailure`, there are writes to the fields `_Rip` and `_Rsp` of the `ContextRecord` variable (an instance of a structure `_CONTEXT`), but if we check the cross-references to `ContextRecord`, we will not see those writes listed.



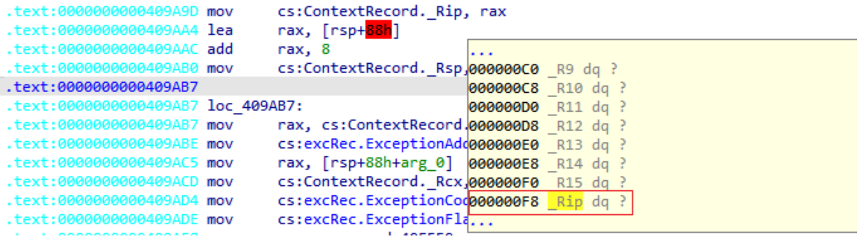
```
.text:0000000000409A95 ; ---------------------------------------------------
.text:0000000000409A95
.text:0000000000409A95 loc_409A95:
.text:0000000000409A95 mov     rax, [rsp+88h]
.text:0000000000409A9D mov     cs:ContextRecord._Rip, rax
.text:0000000000409AA4 lea     rax, [rsp+88h]
.text:0000000000409AAC add     rax, 8
.text:0000000000409AB0 mov     cs:ContextRecord._Rsp, rax
.text:0000000000409AB7
.text:0000000000409AB7 loc_409AB7:
.text:0000000000409AB7 mov     rax, cs:ContextRecord._Rip
.text:0000000000409ABE mov     cs:excRec.ExceptionAddress,
.text:0000000000409AC5 mov     rax, [rsp+88h+arg_0]
.text:0000000000409ACD mov     cs:ContextRecord._Rcx, rax
.text:0000000000409AD4 mov     cs:excRec.ExceptionCode, 0C
```

xrefs to ContextRecord

Direction	Type	Address	Text
Up	o	_report_gsfailure+C	lea rcx, ContextRecord; ContextRecord
Up	o	_report_gsfailure+61	lea rax, ContextRecord
Do...	o	.rdata:stru_40C468	_EXCEPTION_POINTERS <offset excRec, c

Line 1 of 3

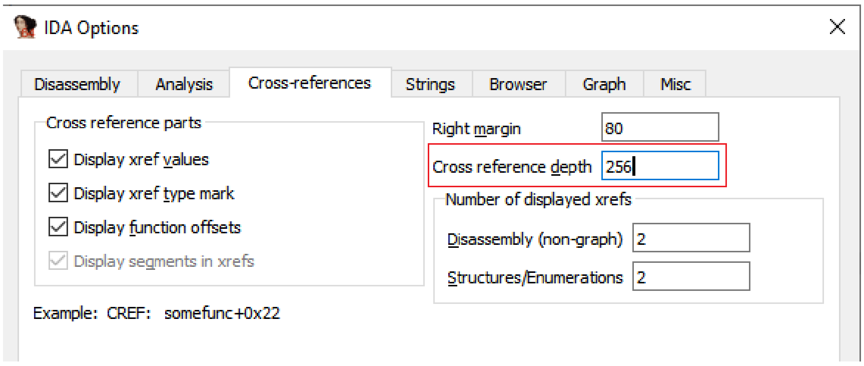
This happens because these fields are situated rather far from the start of the structure (offsets `0x98` and `0xF8`).



```
.text:0000000000409A9D mov     cs:ContextRecord._Rip, rax
.text:0000000000409AA4 lea     rax, [rsp+88h]
.text:0000000000409AAC add     rax, 8
.text:0000000000409AB0 mov     cs:ContextRecord._Rsp,
.text:0000000000409AB7
.text:0000000000409AB7 loc_409AB7:
.text:0000000000409AB7 mov     rax, cs:ContextRecord.
.text:0000000000409ABE mov     cs:excRec.ExceptionAdd
.text:0000000000409AC5 mov     rax, [rsp+88h+arg_0]
.text:0000000000409ACD mov     cs:ContextRecord._Rcx,
.text:0000000000409AD4 mov     cs:excRec.ExceptionCod
.text:0000000000409ADE mov     cs:excRec.ExceptionFla
```

```
...
000000C0 _R9 dq ?
000000C8 _R10 dq ?
000000D0 _R11 dq ?
000000D8 _R12 dq ?
000000E0 _R13 dq ?
000000E8 _R14 dq ?
000000F0 _R15 dq ?
000000F8 _Rip dq ?
...
```

As a speed optimization, IDA only checks for direct accesses into large data items up to a limited depth. The default value is 16(0x10), so any accesses beyond that offset will not be shown. The value for current database can be changed via Options > General... Cross-references tab.



Cross reference parts

☑ Display xref values

☑ Display xref type mark

☑ Display function offsets

☑ Display segments in xrefs

Right margin: 80

Cross reference depth: 256

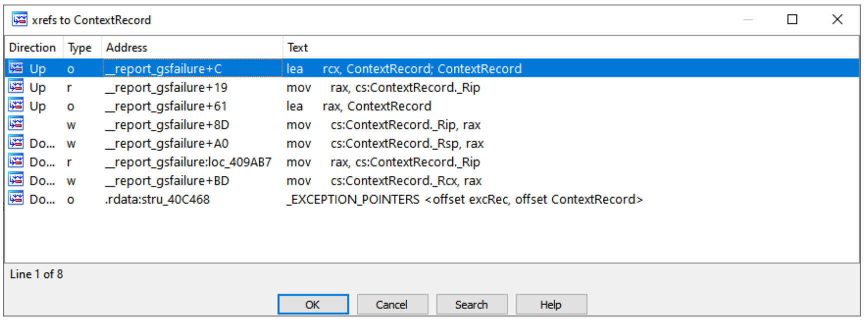
Number of displayed xrefs

Disassembly (non-graph): 2

Structures/Enumerations: 2

Example: CREF: somefunc+0x22

For example, after setting it to 256, the accesses to _Rip and _Rsp are shown in the cross-references to `ContextRecord` :



xrefs to ContextRecord

Direction	Type	Address	Text
Up	o	_report_gsfailure+C	lea rcx, ContextRecord; ContextRecord
Up	r	_report_gsfailure+19	mov rax, cs:ContextRecord._Rip
Up	o	_report_gsfailure+61	lea rax, ContextRecord
	w	_report_gsfailure+8D	mov cs:ContextRecord._Rip, rax
Do...	w	_report_gsfailure+A0	mov cs:ContextRecord._Rsp, rax
Do...	r	_report_gsfailure:loc_409AB7	mov rax, cs:ContextRecord._Rip
Do...	w	_report_gsfailure+BD	mov cs:ContextRecord._Rcx, rax
Do...	o	.rdata:stru_40C468	_EXCEPTION_POINTERS <offset excRec, offset ContextRecord>

Line 1 of 8

To change the limit for all new databases, change the parameter `MAX_TAIL` in `ida.cfg`.

See also:
IDA Help: Cross References Dialog[2]

[1] https://hex-rays.com/blog/igor-tip-of-the-week-16-cross-references/
[2] https://www.hex-rays.com/products/ida/support/idadoc/607.shtml

#98: Analysis options

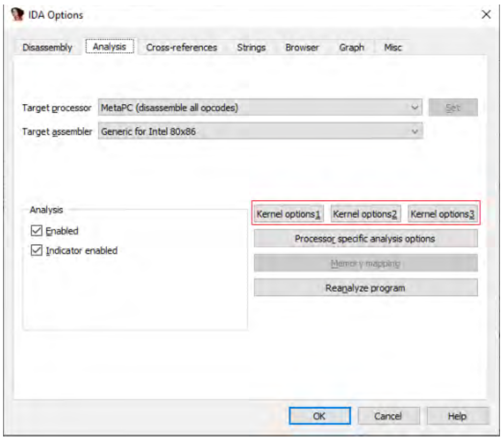
18 Jul 2022

https://hex-rays.com/blog/igors-tip-of-the-week-98-analysis-options/

The autoanalysis engine is the heart of IDA's disassembly functionality. In most cases it "just works" but in rare situations tweaking it may be necessary.

Analysis options

The generic analysis options are available in Options > General, Analysis tab, Kernel Options 1..3.



IDA Options

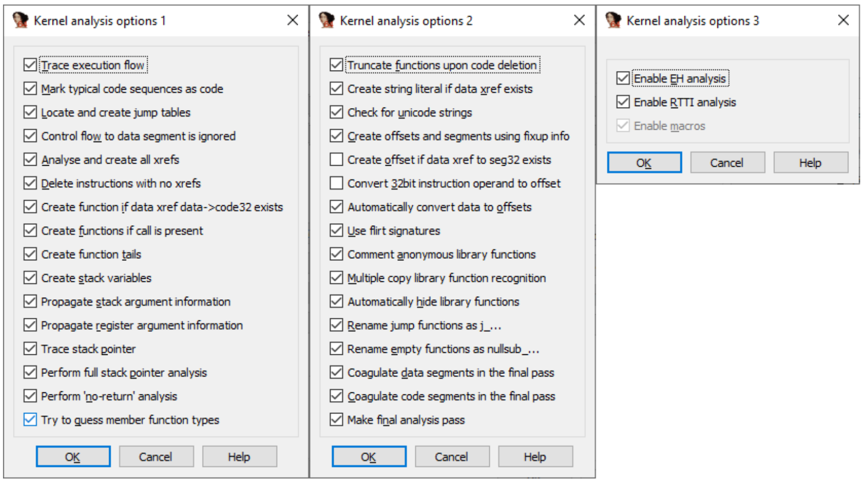
Target processor: MetaPC (disassemble all opcodes)

Target assembler: Generic for Intel 80x86

Analysis

☑ Enabled

☑ Indicator enabled



Kernel analysis options 1

☑ Trace execution flow
☑ Mark typical code sequences as code
☑ Locate and create jump tables
☑ Control flow to data segment is ignored
☑ Analyse and create all xrefs
☑ Delete instructions with no xrefs
☑ Create function if data xref data->code32 exists
☑ Create functions if call is present
☑ Create function tails
☑ Create stack variables
☑ Propagate stack argument information
☑ Propagate register argument information
☑ Trace stack pointer
☑ Perform full stack pointer analysis
☑ Perform 'no-return' analysis
☑ Try to guess member function types

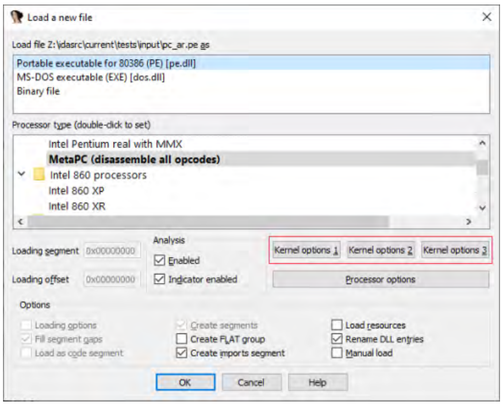
Kernel analysis options 2

☑ Truncate functions upon code deletion
☑ Create string literal if data xref exists
☑ Check for unicode strings
☑ Create offsets and segments using fixup info
□ Create offset if data xref to seg32 exists
□ Convert 32bit instruction operand to offset
☑ Automatically convert data to offsets
☑ Use flirt signatures
☑ Comment anonymous library functions
☑ Multiple copy library function recognition
☑ Automatically hide library functions
☑ Rename jump functions as j_...
☑ Rename empty functions as nullsub_...
☑ Coagulate data segments in the final pass
☑ Coagulate code segments in the final pass
☑ Make final analysis pass

Kernel analysis options 3

☑ Enable EH analysis
☑ Enable RTTI analysis
☑ Enable macros

The same settings are also available at the initial load time.



Load a new file

Load file Z:\idasrc\current\tests\input\pc_ar.pe as

Portable executable for 80386 (PE) [pe.dll]
MS-DOS executable (EXE) [dos.dll]
Binary file

Processor type (double-click to set)

Intel Pentium real with MMX
MetaPC (disassemble all opcodes)
Intel 860 processors
Intel 860 XP
Intel 860 XR

Loading segment 0x00000000

Loading offset 0x00000000

Analysis

☑ Enabled

☑ Indicator enabled

Options

□ Loading options
☑ Fill segment gaps
□ Load as code segment
☑ Create segments
□ Create FLAT group
☑ Create imports segment
□ Load resources
☑ Rename DLL entries
□ Manual load

You can even turn off the autoanalysis completely by unchecking the "Enabled" checkbox. This can be useful, for example, if you have some custom analysis scripts or plugins specific to the target and want to run them before IDA's own analysis. After running the scripts, analysis can be re-enabled to handle the remaining parts of the binary.

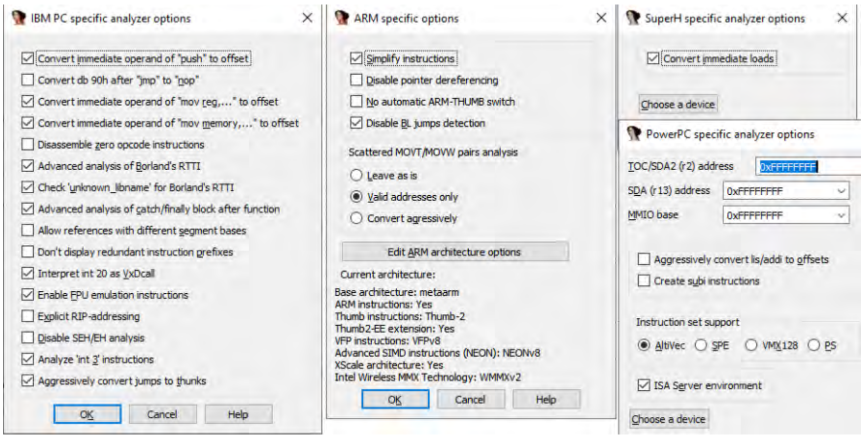
#98: Analysis options

18 Jul 2022

https://hex-rays.com/blog/igors-tip-of-the-week-98-analysis-options/

Processor-specific options

Some processor modules have additional options which are accessible via the “Processor options” button.



IBM PC specific analyzer options

☑ Convert immediate operand of "push" to offset
□ Convert db 90h after "jmp" to "nop"
☑ Convert immediate operand of "mov reg,..." to offset
☑ Convert immediate operand of "mov memory,..." to offset
□ Disassemble zero opcode instructions
☑ Advanced analysis of Borland's RTTI
☑ Check 'unknown_libname' for Borland's RTTI
☑ Advanced analysis of catch/finally block after function
□ Allow references with different segment bases
□ Don't display redundant instruction prefixes
☑ Interpret int 20 as VxDcall
☑ Enable FPU emulation instructions
□ Explicit RIP-addressing
□ Disable SEH/EH analysis
☑ Analyze 'int 3' instructions
☑ Aggressively convert jumps to thunks

OK Cancel Help

ARM specific options

☑ Simplify instructions
□ Disable pointer dereferencing
□ No automatic ARM-THUMB switch
☑ Disable BL jumps detection

Scattered MOVT/MOVW pairs analysis
○ Leave as is
● Valid addresses only
○ Convert agressively

Edit ARM architecture options

Current architecture:
Base architecture: metaarm
ARM instructions: Yes
Thumb instructions: Thumb-2
Thumb2-EE extension: Yes
VFP instructions: VFPv8
Advanced SIMD instructions (NEON): NEONv8
XScale architecture: Yes
Intel Wireless MMX Technology: WMMXv2

OK Cancel Help

SuperH specific analyzer options

☑ Convert immediate loads

Choose a device

PowerPC specific analyzer options

TOC/SDA2 (r2) address: 0xFFFFFFFF
SDA (r13) address: 0xFFFFFFFF
MMIO base: 0xFFFFFFFF

□ Aggressively convert lis/addi to offsets
□ Create subi instructions

Instruction set support
● AltiVec ○ SPE ○ VMX128 ○ PS

☑ ISA Server environment

Choose a device

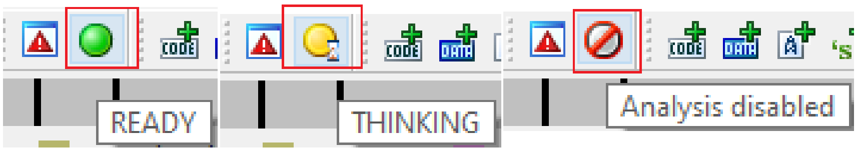
Toggling autoanalysis

In some situations where IDA does not behave correctly or even getting in your way, instead of looking for a specific setting to disable, it may suffice to quickly disable autoanalysis, perform some action, then enable it again for default behavior. For example, if you try to use the technique described in the tip #87[1] without deleting the function first, you may find yourself fighting the autoanalysis:

1. try to truncate the function at the start of the shared tail using “Set Function End” (shortcut `E`);
2. IDA truncates the main function and creates a separate tail chunk;
3. because the function (head chunk) and the tail are adjacent, the autoanalysis immediately merges them back into one big function.

To prevent the undesired merging, you can disable autoanalysis, perform the necessary manipulations, then re-enable it. This can be done by unchecking the “Enabled” checkbox in the Options dialog but there is a faster way: autoanalysis indicator button on the toolbar.

It is usually either a yellow circle with hourglass (autoanalysis in progress) or green circle (autoanalysis idle, waiting for user action). If you click the button, it will turn into a crossed red circle to indicate that autoanalysis has been disabled.



Click on the crossed circle again to re-enable autoanalysis.

See also:
IDA Help: Analysis options (hex-rays.com)[2]
Igor’s tip of the week #09: Reanalysis – Hex Rays (hex-rays.com)[3]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-87-function-chunks-and-the-decompiler/
[2] https://www.hex-rays.com/products/ida/support/idadoc/620.shtml
[3] https://hex-rays.com/blog/igor-tip-of-the-week-09-reanalysis/

#99: Enums

22 Jul 2022

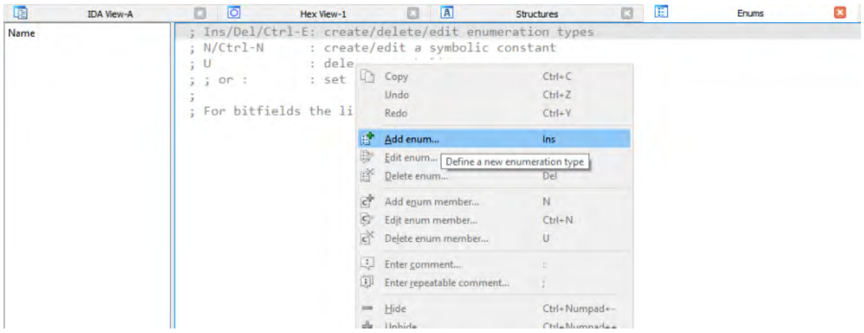
https://hex-rays.com/blog/igors-tip-of-the-week-99-enums/

In IDA, an enum (from “enumeration”) is a set of symbolic constants with numerical values. They can be thought of as a superset of C/C++ enum types and preprocessor defines.

These constants can be used in disassembly or pseudocode to replace specific numbers or their combinations with symbolic names, making the listing more readable and understandable.

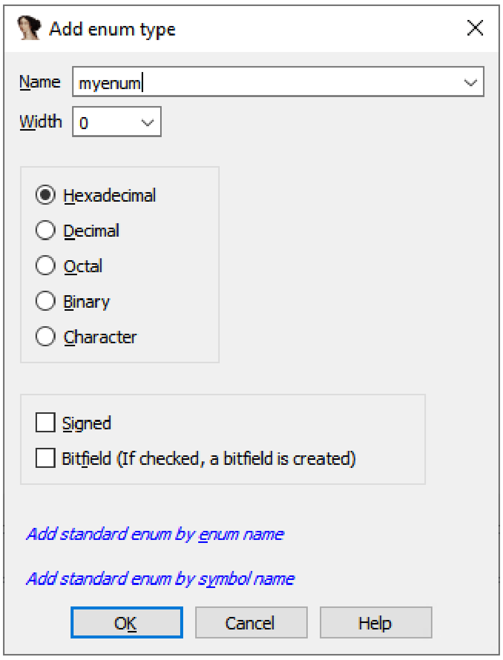
Creating enums manually

The Enums view is a part of the default IDA desktop layout, but it can also be opened via View > Open subviews > Enumerations, or the shortcut `Shift-F10`.



```
; Ins/Del/Ctrl-E: create/delete/edit enumeration types
; N/Ctrl-N       : create/edit a symbolic constant
; U              : dele
; ; or :         : set
;
; For bitfields the li
```

To add a new enum, use “Add enum...” from the context menu, or the shortcut `Ins` (`I` on Macs).



Add enum type

Name: myenum

Width: 0

● Hexadecimal
○ Decimal
○ Octal
○ Binary
○ Character

□ Signed
□ Bitfield (If checked, a bitfield is created)

Add standard enum by enum name
Add standard enum by symbol name
OK
Cancel
Help

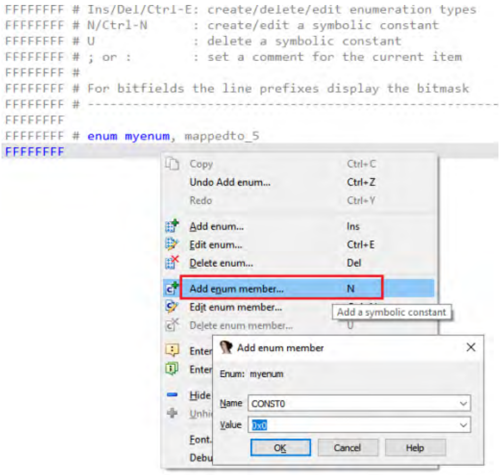
In the dialog you can specify the name, width (size in bytes), and numerical radix for the symbolic constants.

Once the enum has been created, you can start adding constants to it. For this, use “Add enum member...” from the context menu, or the shortcut `N`.

#99: Enums

22 Jul 2022

https://hex-rays.com/blog/igors-tip-of-the-week-99-enums/



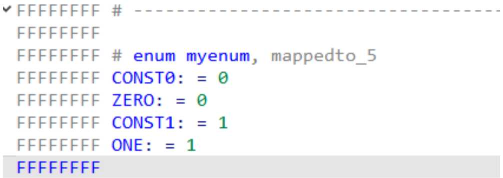
```
FFFFFFFF # Ins/Del/Ctrl-E: create/delete/edit enumeration types
FFFFFFFF # N/Ctrl-N      : create/edit a symbolic constant
FFFFFFFF # U             : delete a symbolic constant
FFFFFFFF # ; or :        : set a comment for the current item
FFFFFFFF #
FFFFFFFF # For bitfields the line prefixes display the bitmask
FFFFFFFF # ------------------------------------------------------
FFFFFFFF
FFFFFFFF # enum myenum, mappedto_5
FFFFFFFF
```

Enum: myenum

Name: CONST0

Value: 0x0

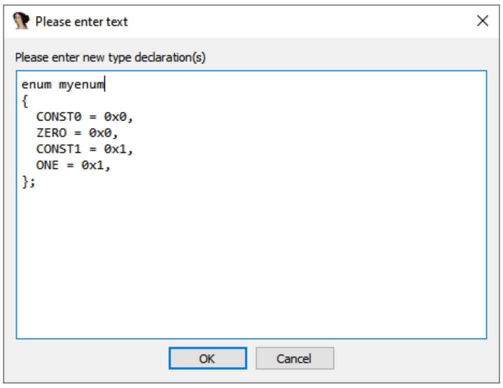
An enum may have multiple constants with the same value but the names of all constants must be unique.



```
FFFFFFFF # -----------------------------------
FFFFFFFF
FFFFFFFF # enum myenum, mappedto_5
FFFFFFFF CONST0: = 0
FFFFFFFF ZERO: = 0
FFFFFFFF CONST1: = 1
FFFFFFFF ONE: = 1
FFFFFFFF
```

Creating enums via Local Types

Local Types view can also be used for creating enums. Simply press `Ins` , write a C syntax definition in the text box and click OK.



Please enter new type declaration(s)

```
enum myenum
{
  CONST0 = 0x0,
  ZERO = 0x0,
  CONST1 = 0x1,
  ONE = 0x1,
};
```

To make the enum available in the Enums view, so that it can be used in the disassembly, use “Synchronize to idb” from the context menu, or simply double-click the newly added enum type.

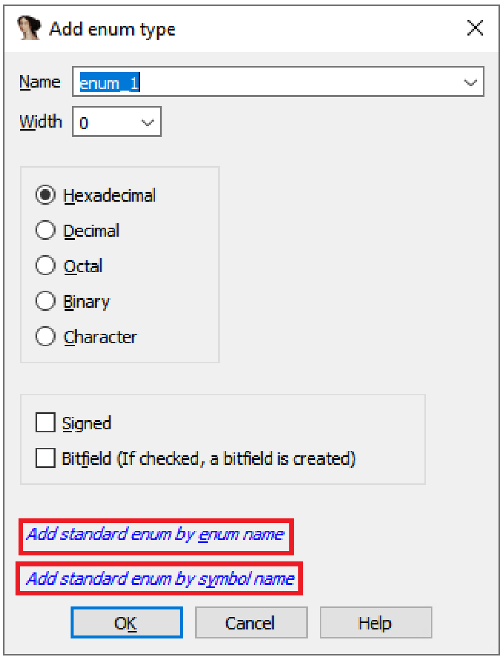
Importing enums from type libraries

Instead of creating an enum from scratch, you can also make use of type libraries shipped with IDA, which include enums from system headers and SDKs. If you know a name of the enum or one of its members, you can check if they’re present in the loaded type libraries. For this, use one of the two link buttons available in the “Add enum” dialog:

#99: Enums

22 Jul 2022

https://hex-rays.com/blog/igors-tip-of-the-week-99-enums/



Name enum_1

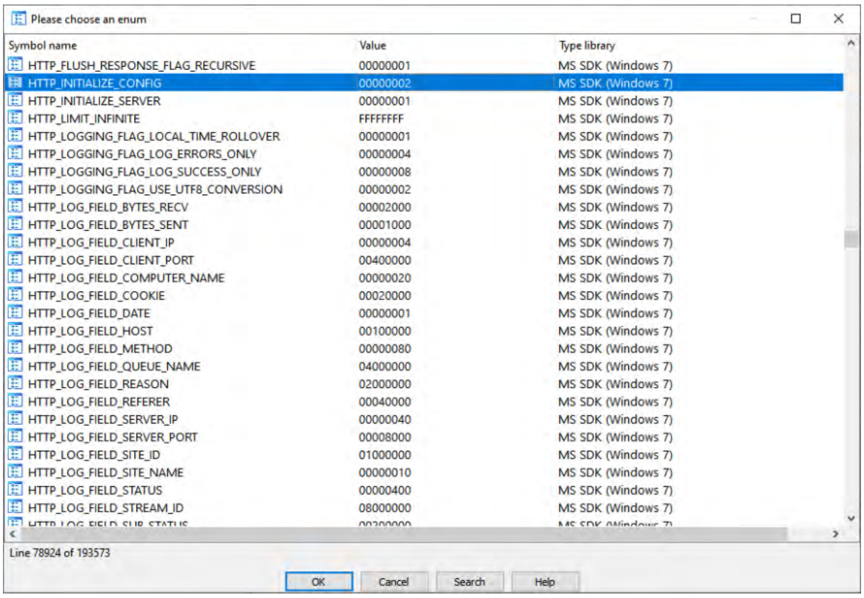
Width 0

● Hexadecimal
○ Decimal
○ Octal
○ Binary
○ Character

□ Signed
□ Bitfield (If checked, a bitfield is created)

Add standard enum by enum name
Add standard enum by symbol name
OK
Cancel
Help

If you click one or the other, IDA will show you the list of all enums or members(symbols) available in the currently loaded type libraries[1].



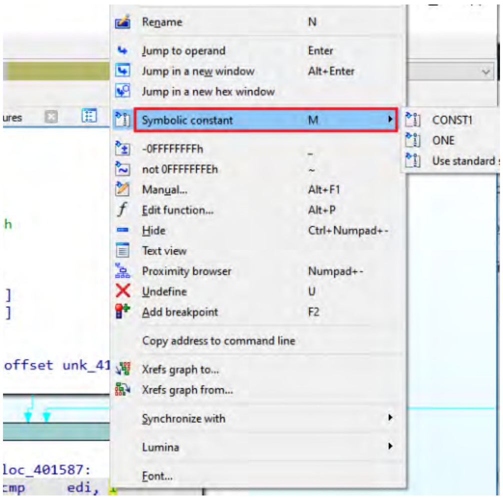
Symbol name	Value	Type library
HTTP_FLUSH_RESPONSE_FLAG_RECURSIVE	00000001	MS SDK (Windows 7)
HTTP_INITIALIZE_CONFIG	00000002	MS SDK (Windows 7)
HTTP_INITIALIZE_SERVER	00000001	MS SDK (Windows 7)
HTTP_LIMIT_INFINITE	FFFFFFFF	MS SDK (Windows 7)
HTTP_LOGGING_FLAG_LOCAL_TIME_ROLLOVER	00000001	MS SDK (Windows 7)
HTTP_LOGGING_FLAG_LOG_ERRORS_ONLY	00000004	MS SDK (Windows 7)
HTTP_LOGGING_FLAG_LOG_SUCCESS_ONLY	00000008	MS SDK (Windows 7)
HTTP_LOGGING_FLAG_USE_UTF8_CONVERSION	00000002	MS SDK (Windows 7)
HTTP_LOG_FIELD_BYTES_RECV	00002000	MS SDK (Windows 7)
HTTP_LOG_FIELD_BYTES_SENT	00001000	MS SDK (Windows 7)
HTTP_LOG_FIELD_CLIENT_IP	00000004	MS SDK (Windows 7)
HTTP_LOG_FIELD_CLIENT_PORT	00400000	MS SDK (Windows 7)
HTTP_LOG_FIELD_COMPUTER_NAME	00000020	MS SDK (Windows 7)
HTTP_LOG_FIELD_COOKIE	00020000	MS SDK (Windows 7)
HTTP_LOG_FIELD_DATE	00000001	MS SDK (Windows 7)
HTTP_LOG_FIELD_HOST	00100000	MS SDK (Windows 7)
HTTP_LOG_FIELD_METHOD	00000080	MS SDK (Windows 7)
HTTP_LOG_FIELD_QUEUE_NAME	04000000	MS SDK (Windows 7)
HTTP_LOG_FIELD_REASON	02000000	MS SDK (Windows 7)
HTTP_LOG_FIELD_REFERER	00040000	MS SDK (Windows 7)
HTTP_LOG_FIELD_SERVER_IP	00000040	MS SDK (Windows 7)
HTTP_LOG_FIELD_SERVER_PORT	00008000	MS SDK (Windows 7)
HTTP_LOG_FIELD_SITE_ID	01000000	MS SDK (Windows 7)
HTTP_LOG_FIELD_SITE_NAME	00000010	MS SDK (Windows 7)
HTTP_LOG_FIELD_STATUS	00000400	MS SDK (Windows 7)
HTTP_LOG_FIELD_STREAM_ID	08000000	MS SDK (Windows 7)
HTTP_LOG_FIELD_SUB_STATUS	00200000	MS SDK (Windows 7)

Line 78924 of 193573

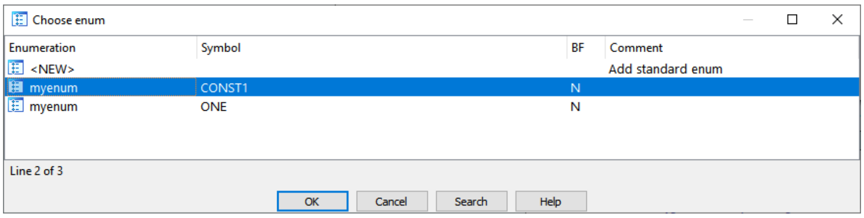
If you know the standard enum name beforehand, simply enter it in the “Add enum” dialog and IDA will automatically import it if a match is found in a loaded type library.

Using enums

Enums can be used to replace (almost) any numerical value in the disassembly or pseudocode by a symbolic constant. This can be done from the context menu on a number:



Or by pressing the shortcut M, which shows a chooser:

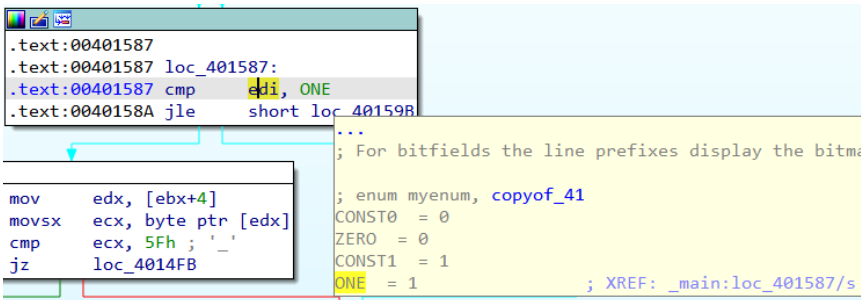

Choose enum

Enumeration	Symbol	BF	Comment
<NEW>			Add standard enum
myenum	CONST1	N	
myenum	ONE	N	

Line 2 of 3

The list of enum members is automatically narrowed down to those matching the number in the disassembly/pseudo-code.

To see the value of the symbolic constant after conversion, hover the mouse[2] over it:


```
.text:00401587
.text:00401587 loc_401587:
.text:00401587 cmp     edi, ONE
.text:0040158A jle     short loc_40159B
```

```
mov     edx, [ebx+4]
movsx   ecx, byte ptr [edx]
cmp     ecx, 5Fh ; '_'
jz      loc_4014FB
```

```
...
; For bitfields the line prefixes display the bitma

; enum myenum, copyof_41
CONST0  = 0
ZERO   = 0
CONST1  = 1
ONE   = 1                    ; XREF: _main:loc_401587/s
```

See also:
IDA Help: Enums window[3]
IDA Help: Convert operand to symbolic constant (enum)[4]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-60-type-libraries/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-47-hints-in-ida/
[3] https://www.hex-rays.com/products/ida/support/idadoc/594.shtml
[4] https://www.hex-rays.com/products/ida/support/idadoc/473.shtml

#100: Collapsing pseudocode parts

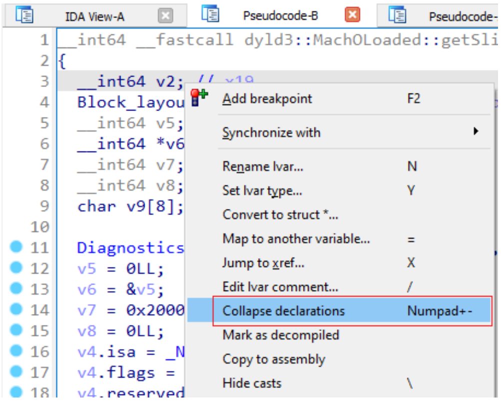
29 Jul 2022

https://hex-rays.com/blog/igors-tip-of-the-week-100-collapsing-pseudocode-parts/

When working with big functions in the decompiler, it may be useful to temporarily hide some parts of the pseudocode to analyze the rest. While currently it's not possible to hide arbitrary lines like in disassembly[1], you can hide specific sections of it.

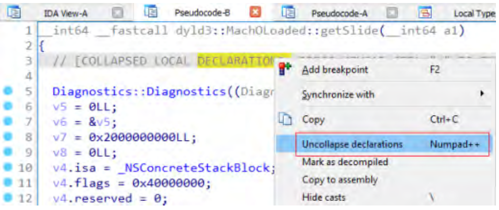
Collapsing local variable declarations

While the local variable declarations are useful to see the overall layout of the stack frame and other interesting info[2], in big functions they may take up a lot of valuable screen estate. To get them out of the way, you can use "Collapse declarations..." from the context menu, or the - key on the numpad.



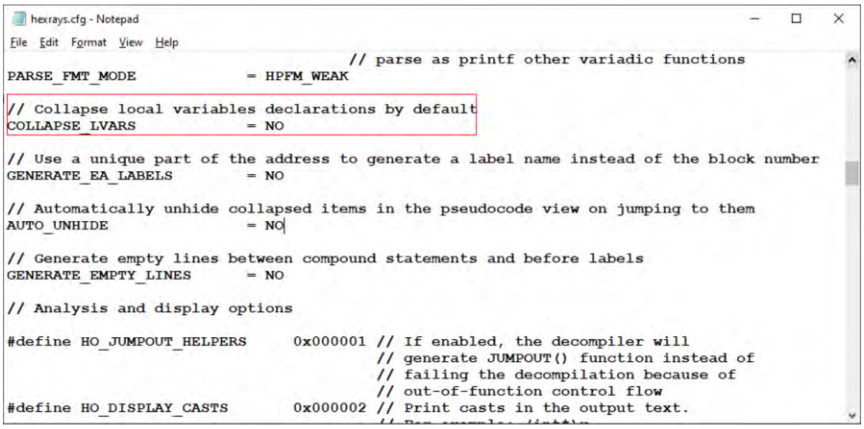
```
__int64 __fastcall dyld3::MachOLoaded::getSli
{
  __int64 v2; // x19
  Block_layou
  __int64 v5;
  __int64 *v6
  __int64 v7;
  __int64 v8;
  char v9[8];

  Diagnostics
  v5 = 0LL;
  v6 = &v5;
  v7 = 0x2000
  v8 = 0LL;
  v4.isa = _N
  v4.flags =
  v4.reserved
```

This replaces the declarations with a single comment line. To show them again, use "Uncollapse declarations.." or the numpad + key.



```
__int64 __fastcall dyld3::MachOLoaded::getSlide(__int64 a1)
{
  // [COLLAPSED LOCAL DECLARATIONS

  Diagnostics::Diagnostics((Diagr
  v5 = 0LL;
  v6 = &v5;
  v7 = 0x2000000000LL;
  v8 = 0LL;
  v4.isa = _NSConcreteStackBlock;
  v4.flags = 0x40000000;
  v4.reserved = 0;
```

To always collapse the declarations by default, set COLLAPSE_LVARS option in `cfg/hexrays.cfg`.



```
                                        // parse as printf other variadic functions
PARSE_FMT_MODE                = HPFM_WEAK

// Collapse local variables declarations by default
COLLAPSE_LVARS                = NO

// Use a unique part of the address to generate a label name instead of the block number
GENERATE_EA_LABELS            = NO

// Automatically unhide collapsed items in the pseudocode view on jumping to them
AUTO_UNHIDE                   = NO

// Generate empty lines between compound statements and before labels
GENERATE_EMPTY_LINES          = NO

// Analysis and display options

#define HO_JUMPOUT_HELPERS      0x000001 // If enabled, the decompiler will
                                         // generate JUMPOUT() function instead of
                                         // failing the decompilation because of
                                         // out-of-function control flow
#define HO_DISPLAY_CASTS        0x000002 // Print casts in the output text.
```

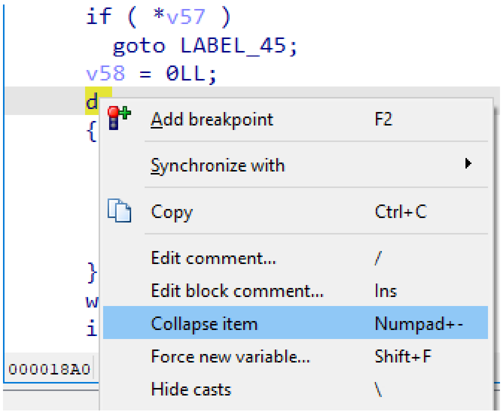
Collapsing statements

Compound statements can be collapsed too: `if` and `switch` statements, as well as `for`, `while`, and do loops. This can be done using the "Collapse item" context menu command, or the same numpad - shortcut.

#100: Collapsing pseudocode parts

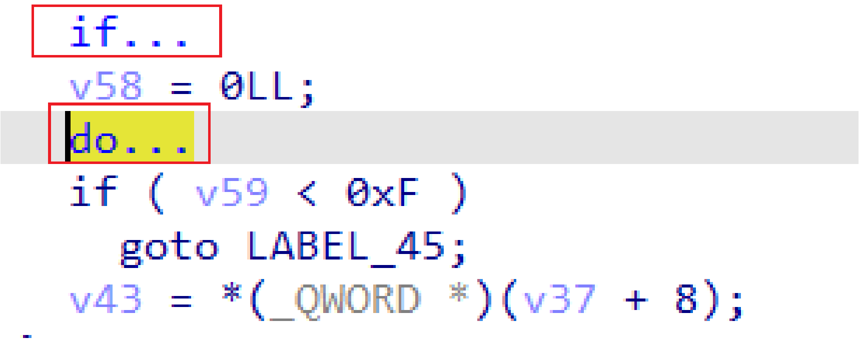
29 Jul 2022

https://hex-rays.com/blog/igors-tip-of-the-week-100-collapsing-pseudocode-parts/



```
if ( *v57 )
  goto LABEL_45;
v58 = 0LL;
d
{


}
w
i
```

After collapsing, the whole statement is replaced by one line with the keyword and ellipsis:



```
if...
v58 = 0LL;
do...
if ( v59 < 0xF )
  goto LABEL_45;
v43 = *(_QWORD *)(v37 + 8);
```

And can be uncollapsed again from context menu or the numpad + key.

You can use this approach to progressively hide analyzed code and tackle long functions piece by piece.

See also

Hex-Rays interactive operation: Hide/unhide C statements (hex-rays.com)[3]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-31-hiding-and-collapsing/

[2] https://hex-rays.com/blog/igors-tip-of-the-week-66-decompiler-annotations/

[3] https://www.hex-rays.com/products/decompiler/manual/cmd_hide.shtml

#101: Decompiling variadic function calls

05 Aug 2022

https://hex-rays.com/blog/igors-tip-of-the-week-101-decompiling-variadic-function-calls/

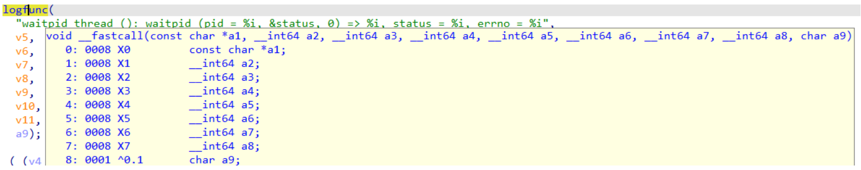
Variadic functions[1] are functions which accept different number of arguments depending on the needs of the caller. Typical examples include **printf** and **scanf** in C and C++ but there are other functions, or even some custom ones (specific to the binary being analyzed). Because each call of a variadic function may have a different set of arguments, they need special handling in the decompiler. In many cases the decompiler detects such functions and their arguments automatically but there may be situations where user intervention is required.

Changing the function prototype

For standard variadic functions IDA usually applies the prototype from a type library[2] but if there's a non-standard function or IDA did not detect that a function is variadic, you can do it manually. For example, a decompiled prototype of unrecognized variadic function on ARM64 may look like this:

```
void __fastcall logfunc(
        const char *a1,
        __int64 a2,
        __int64 a3,
        __int64 a4,
        __int64 a5,
        __int64 a6,
        __int64 a7,
        __int64 a8,
        char a9)
```

But when you inspect the call sites, you see that most of the passed arguments are marked as possibly uninitialized (orange color):

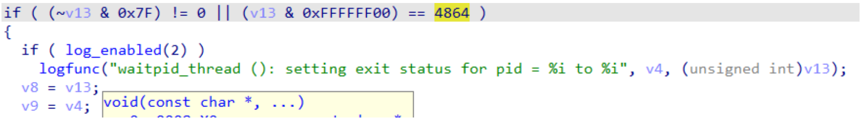


```
logfunc(
  "waitpid thread (): waitpid (pid = %i, &status, 0) => %i, status = %i, errno = %i",
  v5,  void __fastcall(const char *a1, __int64 a2, __int64 a3, __int64 a4, __int64 a5, __int64 a6, __int64 a7, __int64 a8, char a9)
  v6,     0: 0008 X0          const char *a1;
  v7,     1: 0008 X1          __int64 a2;
  v8,     2: 0008 X2          __int64 a3;
  v9,     3: 0008 X3          __int64 a4;
  v10,    4: 0008 X4          __int64 a5;
  v11,    5: 0008 X5          __int64 a6;
  a9);    6: 0008 X6          __int64 a7;
          7: 0008 X7          __int64 a8;
 ( (v4    8: 0001 ^0.1        char a9;
```

The first argument looks like a format string so the rest are likely variadic. So we can try to change the prototype[3] to:

```
void logfunc(const char *, ...);
```

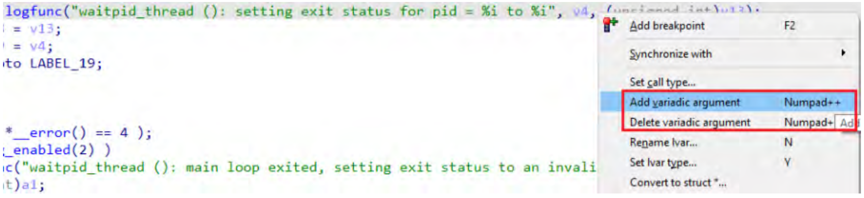
which results in clean decompilation:



```
if ( (~v13 & 0x7F) != 0 || (v13 & 0xFFFFFF00) == 4864 )
{
  if ( log_enabled(2) )
    logfunc("waitpid_thread (): setting exit status for pid = %i to %i", v4, (unsigned int)v13);
  v8 = v13;
  v9 = v4; void(const char *, ...)
```

Adjusting variadic arguments

With correct prototypes, decompiler usually can guess the actual arguments passed to each invocation of the function. However, in some cases the autodetection can misfire, especially if the function uses non-standard format specifiers or does not use a format string at all. In such case, you can adjust the actual number of arguments being passed to the call. This can be done via the context menu commands "Add variadic argument" and "Delete variadic argument", or the corresponding shortcuts Numpad + and Numpad -.



```
logfunc("waitpid_thread (): setting exit status for pid = %i to %i", v4,
 = v13;
 = v4;
to LABEL_19;


*__error() == 4 );
_enabled(2) )
c("waitpid_thread (): main loop exited, setting exit status to an invali
t)a1;
```

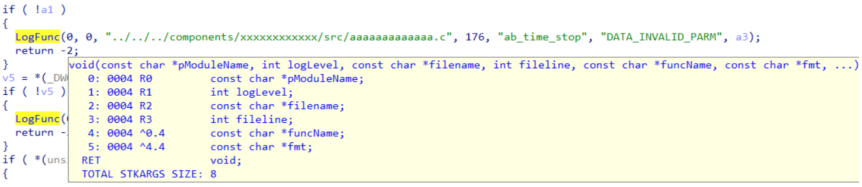
Variadic calls and tail branch optimization

In some rare situations you may run into the following issue: when trying to add or remove variadic arguments, the decompiler seems to ignore the action. This may occur in functions subjected to a specific optimization. For example, here's pseudocode of a function which seems to have two calls to a logging function:

#101: Decompiling variadic function calls

05 Aug 2022

https://hex-rays.com/blog/igors-tip-of-the-week-101-decompiling-variadic-function-calls/

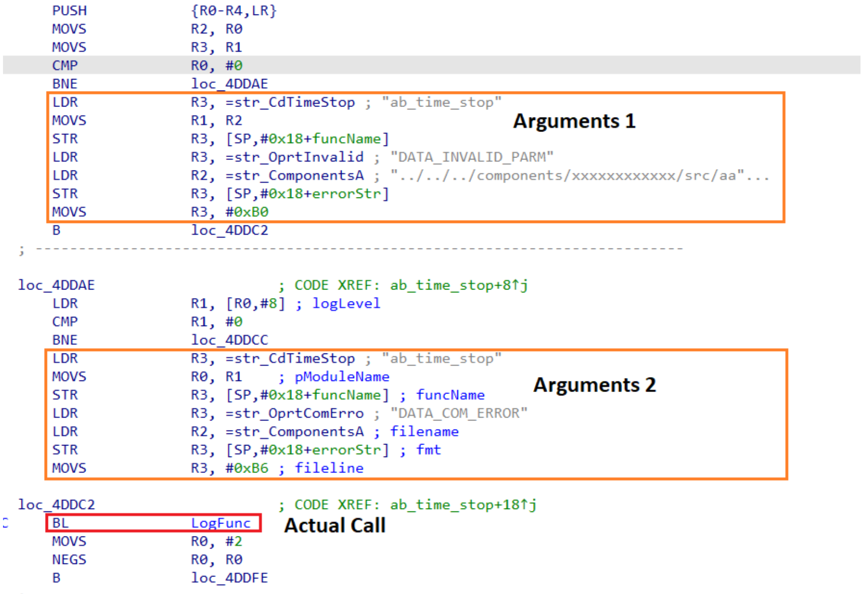


```
if ( !a1 )
{
  LogFunc(0, 0, "../../../components/xxxxxxxxxxxx/src/aaaaaaaaaaaaa.c", 176, "ab_time_stop", "DATA_INVALID_PARM", a3);
  return -2;
}
v5 = *(_DW
if ( !v5 )
{
  LogFunc(
  return -
}
if ( *(uns
{
```

```
void(const char *pModuleName, int logLevel, const char *filename, int fileline, const char *funcName, const char *fmt, ...)
  0: 0004 R0         const char *pModuleName;
  1: 0004 R1         int logLevel;
  2: 0004 R2         const char *filename;
  3: 0004 R3         int fileline;
  4: 0004 ^0.4       const char *funcName;
  5: 0004 ^4.4       const char *fmt;
 RET                 void;
 TOTAL STKARGS SIZE: 8
```

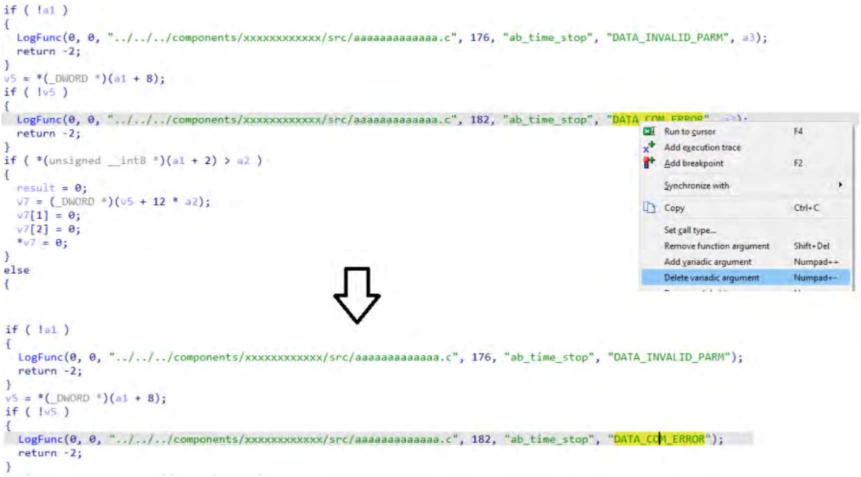
The decompiler has decided that a3 is also passed to the calls, however we can see that the format strings do not have any format specifiers so a3 is a false positive and should be removed. However, using “Delete variadic argument” on the first call seems to have no effect. What’s happening?

This is one of the rare cases where switching to disassembly can clear things up. By pressing **Tab**, we can see a curious picture in the disassembly: there is only one call!



```
    PUSH            {R0-R4,LR}
    MOVS            R2, R0
    MOVS            R3, R1
    CMP             R0, #0
    BNE             loc_4DDAE
    LDR             R3, =str_CdTimeStop ; "ab_time_stop"
    MOVS            R1, R2
    STR             R3, [SP,#0x18+funcName]
    LDR             R3, =str_OprtInvalid ; "DATA_INVALID_PARM"
    LDR             R2, =str_ComponentsA ; "../../../components/xxxxxxxxxxxx/src/aa"...
    STR             R3, [SP,#0x18+errorStr]
    MOVS            R3, #0xB0
    B               loc_4DDC2
; ---------------------------------------------------------------------------

loc_4DDAE                              ; CODE XREF: ab_time_stop+8↑j
    LDR             R1, [R0,#8] ; logLevel
    CMP             R1, #0
    BNE             loc_4DDCC
    LDR             R3, =str_CdTimeStop ; "ab_time_stop"
    MOVS            R0, R1    ; pModuleName
    STR             R3, [SP,#0x18+funcName] ; funcName
    LDR             R3, =str_OprtComErro ; "DATA_COM_ERROR"
    LDR             R2, =str_ComponentsA ; filename
    STR             R3, [SP,#0x18+errorStr] ; fmt
    MOVS            R3, #0xB6 ; fileline

loc_4DDC2                              ; CODE XREF: ab_time_stop+18↑j
    BL              LogFunc
    MOVS            R0, #2
    NEGS            R0, R0
    B               loc_4DDFE
```

This is an example of so-called tail branch merging optimization, where the same function call is reused with different arguments. For better code readability, the decompiler detects this situation and creates a duplicate call statement with the second set of arguments. Because the information about the number of variadic arguments is attached to the actual call instruction, it can’t be changed for the “fake” call inserted by the decompiler. You can change it for the “canonical” one which can be discovered by pressing **Tab** on the call (BL instruction). Removing the argument there affects both calls in the pseudocode.



```
if ( !a1 )
{
  LogFunc(0, 0, "../../../components/xxxxxxxxxxxx/src/aaaaaaaaaaaaa.c", 176, "ab_time_stop", "DATA_INVALID_PARM", a3);
  return -2;
}
v5 = *(_DWORD *)(a1 + 8);
if ( !v5 )
{
  LogFunc(0, 0, "../../../components/xxxxxxxxxxxx/src/aaaaaaaaaaaaa.c", 182, "ab_time_stop", "DATA_COM_ERROR", a3);
  return -2;
}
if ( *(unsigned __int8 *)(a1 + 2) > a2 )
{
  result = 0;
  v7 = (_DWORD *)(v5 + 12 * a2);
  v7[1] = 0;
  v7[2] = 0;
  *v7 = 0;
}
else
{
```

Run to cursor F4
Add execution trace
Add breakpoint F2
Synchronize with
Copy Ctrl+C
Set call type...
Remove function argument Shift+Del
Add variadic argument Numpad++
Delete variadic argument Numpad+-

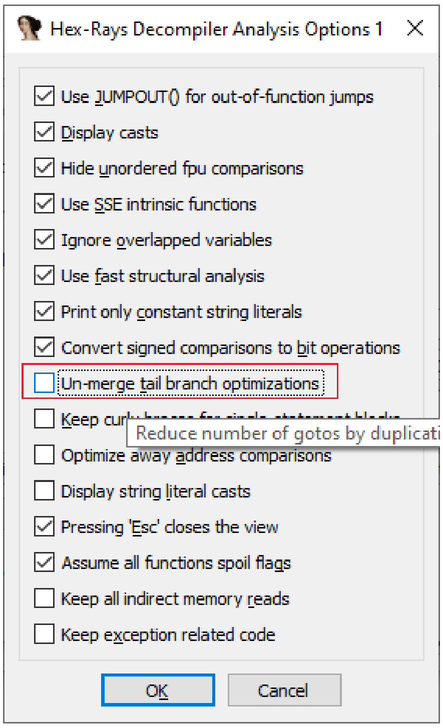
```
if ( !a1 )
{
  LogFunc(0, 0, "../../../components/xxxxxxxxxxxx/src/aaaaaaaaaaaaa.c", 176, "ab_time_stop", "DATA_INVALID_PARM");
  return -2;
}
v5 = *(_DWORD *)(a1 + 8);
if ( !v5 )
{
  LogFunc(0, 0, "../../../components/xxxxxxxxxxxx/src/aaaaaaaaaaaaa.c", 182, "ab_time_stop", "DATA_COM_ERROR");
  return -2;
}
```

If you’re curious to see the “original” code, it can be done by turning off “Un-merge tail branch optimization” in the decompiler’s Analysis Options [14].

#101: Decompiling variadic function calls

05 Aug 2022

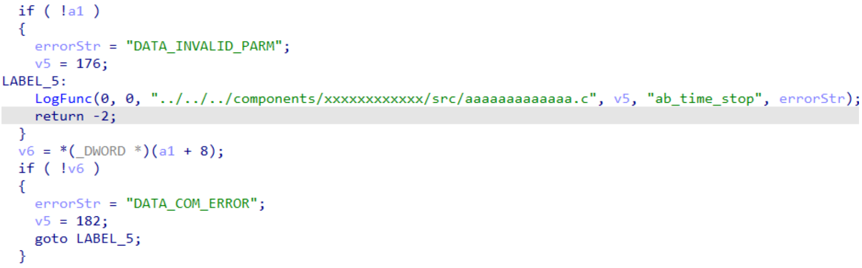
https://hex-rays.com/blog/igors-tip-of-the-week-101-decompiling-variadic-function-calls/



Hex-Rays Decompiler Analysis Options 1

☑ Use JUMPOUT() for out-of-function jumps
☑ Display casts
☑ Hide unordered fpu comparisons
☑ Use SSE intrinsic functions
☑ Ignore overlapped variables
☑ Use fast structural analysis
☑ Print only constant string literals
☑ Convert signed comparisons to bit operations
☐ Un-merge tail branch optimizations
☐ Keep curly braces for single-statement blocks
Reduce number of gotos by duplicati
☐ Optimize away address comparisons
☐ Display string literal casts
☑ Pressing 'Esc' closes the view
☑ Assume all functions spoil flags
☐ Keep all indirect memory reads
☐ Keep exception related code

With it off, there is only one call just like in the disassembly, at the cost of an extra goto and some local variables:



```
  if ( !a1 )
  {
    errorStr = "DATA_INVALID_PARM";
    v5 = 176;
LABEL_5:
    LogFunc(0, 0, "../../../components/xxxxxxxxxxxx/src/aaaaaaaaaaaaa.c", v5, "ab_time_stop", errorStr);
    return -2;
  }
  v6 = *(_DWORD *)(a1 + 8);
  if ( !v6 )
  {
    errorStr = "DATA_COM_ERROR";
    v5 = 182;
    goto LABEL_5;
  }
```

See also:
Hex-Rays interactive operation: Add/del variadic arguments (hex-rays.com)[5]

[1] https://en.wikipedia.org/wiki/Variadic_function
[2] https://hex-rays.com/blog/igors-tip-of-the-week-60-type-libraries/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-42-renaming-and-retyping-in-the-decompiler/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-56-string-literals-in-pseudocode/
[5] https://www.hex-rays.com/products/decompiler/manual/cmd_variadic.shtml

#102: Resetting decompiler information

12 Aug 2022

https://hex-rays.com/blog/igors-tip-of-the-week-102-resetting-decompiler-information/

While working with pseudocode, you may make various changes to it, for example:

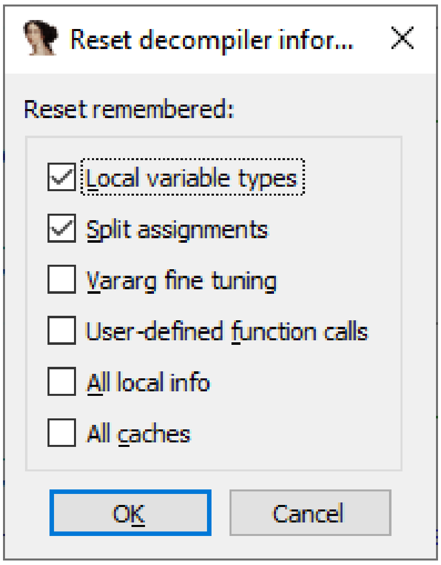
- add comments[1]
- rename local variables and change their types[2]
- collapse code blocks[3]
- map variables[4]
- mark skippable instructions[5]
- split expressions[6]
- adjust variadic arguments[7]
- select union members[8]
- and so on

If the results of some actions do not look better, you can always undo, but what to do if you discover a problem long after the action which caused it?

In fact, there is a way to reset specific or all user customizations at once.

Reset decompiler information

By Invoking Edit > Other > Reset decompiler information… you get the following dialog:



Reset decompiler infor...

Reset remembered:

☑ Local variable types

☑ Split assignments

☐ Vararg fine tuning

☐ User-defined function calls

☐ All local info

☐ All caches

Here, you can pick what kinds of information to reset. The fist several options reset information specific to the current function while the last one also resets caches, such as the microcode and pseudocode caches, for all functions, as well as the global cross references[9] cache.

See also:
Decompiler Manual: Interactive operation[10]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-43-annotating-the-decompiler-output/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-42-renaming-and-retyping-in-the-decompiler/
[3] https://hex-rays.com/blog/igors-tip-of-the-week-100-collapsing-pseudocode-parts/
[4] https://hex-rays.com/blog/igors-tip-of-the-week-77-mapped-variables/
[5] https://hex-rays.com/blog/igors-tip-of-the-week-68-skippable-instructions/
[6] https://hex-rays.com/blog/igors-tip-of-the-week-69-split-expression/
[7] https://hex-rays.com/blog/igors-tip-of-the-week-101-decompiling-variadic-function-calls/
[8] https://hex-rays.com/blog/igors-tip-of-the-week-75-working-with-unions/
[9] https://hex-rays.com/blog/igors-tip-of-the-week-18-decompiler-and-global-cross-references/
[10] https://www.hex-rays.com/products/decompiler/manual/interactive.shtml#07

#103: Sharing plugins between IDA installs

19 Aug 2022

https://hex-rays.com/blog/igors-tip-of-the-week-103-sharing-plugins-between-ida-installs/

As of the time of writing, IDA does not have a built-in plugin manager, so third-party plugins have to be installed manually.

Installing into IDA directory

The standard location for IDA plugins is the `plugins` directory in IDA's installation (for example, `C:\Program Files\IDA Pro 8.0\plugins` on Windows). So this is the most common way of installing them – just copy the plugin file(s) there and they'll be loaded on next start of IDA. However, this only makes them available for this specific IDA install. If you install a new version of IDA (which by default uses a version-specific directory name), you'll need to re-copy plugins to the new location.

Installing into user directory

In addition to IDA's own directory, IDA also checks for plugins in the user directory[1]. So you can put them in:

- `%APPDATA%\Hex-Rays\IDA Pro\plugins` on Windows
- `$HOME/.idapro/plugins` on Linux/Mac

You can find out the exact path for your system by executing `idaapi.get_ida_subdirs("plugins")` in IDA.

Such plugins will be loaded by any IDA, so there may be issues if they use functionality which is not available or changed between versions, but the advantage is that there's no need to reinstall them when upgrading IDA (or using multiple versions).

See also:
Igor's tip of the week #33: IDA's user directory (IDAUSR)[2]
IDA Help: Environment variables[3]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-33-idas-user-directory-idausr/
[2] https://hex-rays.com/blog/igors-tip-of-the-week-33-idas-user-directory-idausr/
[3] https://www.hex-rays.com/products/ida/support/idadoc/1375.shtml

#104: Immediate search

26 Aug 2022

https://hex-rays.com/blog/igors-tip-of-the-week-104-immediate-search/

Immediate search is one of three main search types[1] available in IDA. While not that known, it can be very useful in some situations. Here are some examples.

Unique (magic) constants

If you know some unique constants used by the program, looking for them can let you narrow down the range of code you have to analyze. For example, if a program reports a numerical error code, you could look for it to find the possible locations which may be returning this error.

Undiscovered cross-references in RISC processors

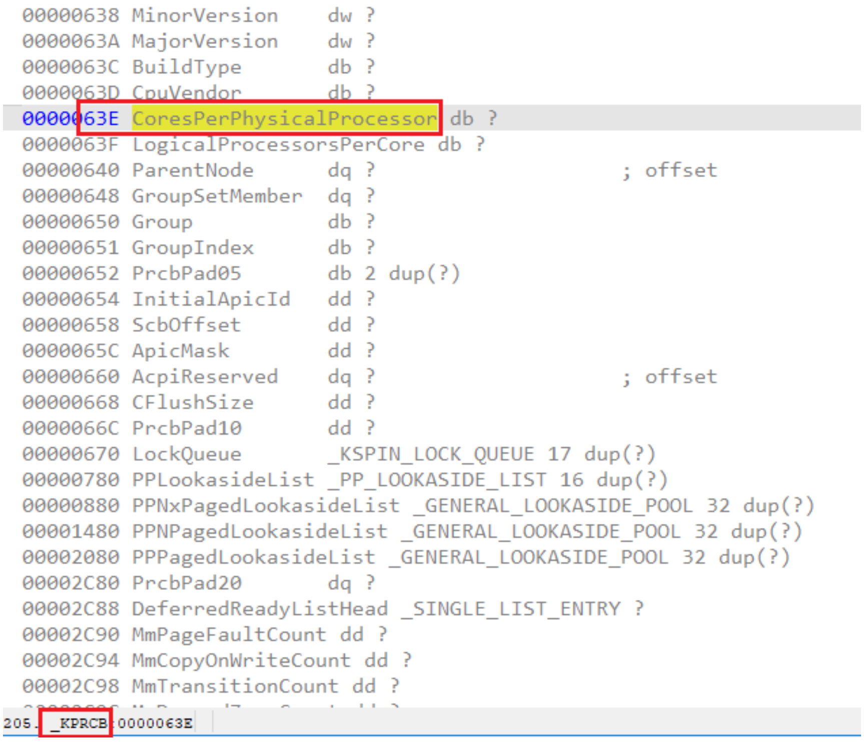
Many RISC processors use fixed-width instructions which does not leave enough space for encoding full address values in the instruction. Thus they have to resort to building address values out of small pieces. For, example, in SPARC, loading of a 32-bit value has to be done as a pair of instructions[2]:

```
sethi %hi(Prompt),%o1
or    %o1,%lo(Prompt),%o1
```

Where `%hi` returns top 22 bits of the value and `%lo` returns the low 10 bits. Because such instructions may be not immediately next to each other, IDA may fail to "connect" them and recover the full 32-bit value, leading to missing cross references. So if you have, for example, a string constant at address `N`, which you think should be referenced from somewhere, doing an immediate search for `N&0x3FF` should produce a list of potential candidates for instructions referring to that address.

Structure field references

Sometimes you may have a structure with a field at a specific offset which is pretty unique (not a small or round value) and want to find where it is used in the program. For example, let's look at a recent Windows kernel and the structure `_KPRCB`. At offset 63Eh, it has a field `CoresPerPhysicalProcessor`:



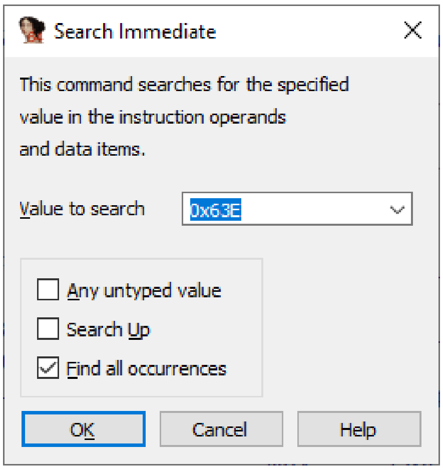
```
00000638 MinorVersion     dw ?
0000063A MajorVersion     dw ?
0000063C BuildType        db ?
0000063D CpuVendor        db ?
0000063E CoresPerPhysicalProcessor db ?
0000063F LogicalProcessorsPerCore db ?
00000640 ParentNode       dq ?                    ; offset
00000648 GroupSetMember   dq ?
00000650 Group            db ?
00000651 GroupIndex       db ?
00000652 PrcbPad05        db 2 dup(?)
00000654 InitialApicId    dd ?
00000658 ScbOffset        dd ?
0000065C ApicMask         dd ?
00000660 AcpiReserved     dq ?                    ; offset
00000668 CFlushSize       dd ?
0000066C PrcbPad10        dd ?
00000670 LockQueue        _KSPIN_LOCK_QUEUE 17 dup(?)
00000780 PPLookasideList _PP_LOOKASIDE_LIST 16 dup(?)
00000880 PPNxPagedLookasideList _GENERAL_LOOKASIDE_POOL 32 dup(?)
00001480 PPNPagedLookasideList _GENERAL_LOOKASIDE_POOL 32 dup(?)
00002080 PPPagedLookasideList _GENERAL_LOOKASIDE_POOL 32 dup(?)
00002C80 PrcbPad20        dq ?
00002C88 DeferredReadyListHead _SINGLE_LIST_ENTRY ?
00002C90 MmPageFaultCount dd ?
00002C94 MmCopyOnWriteCount dd ?
00002C98 MmTransitionCount dd ?
205. _KPRCB:0000063E
```

How to find where it is used? Searching for the value 0x63e gives a list of instructions using that value.

#104: Immediate search

26 Aug 2022

https://hex-rays.com/blog/igors-tip-of-the-week-104-immediate-search/

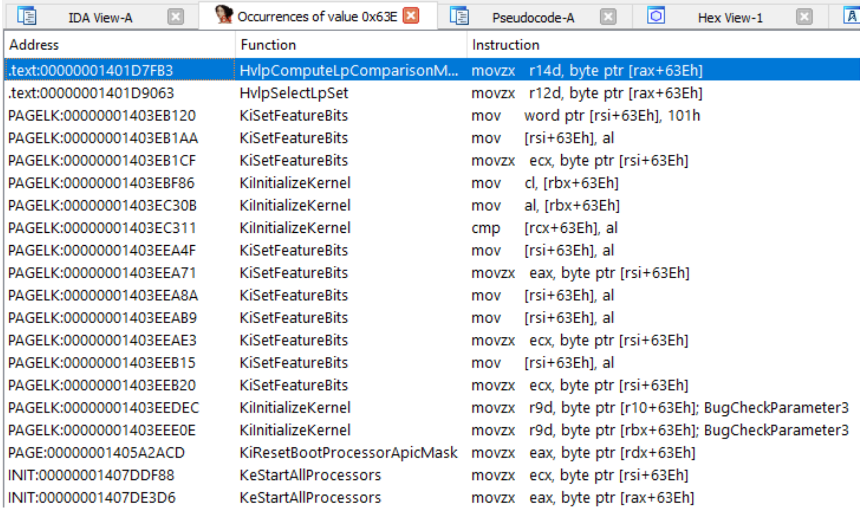


Search Immediate

This command searches for the specified value in the instruction operands and data items.

Value to search: 0x63E

□ Any untyped value
□ Search Up
☑ Find all occurrences



Address	Function	Instruction
.text:00000001401D7FB3	HvlpComputeLpComparisonM...	movzx r14d, byte ptr [rax+63Eh]
.text:00000001401D9063	HvlpSelectLpSet	movzx r12d, byte ptr [rax+63Eh]
PAGELK:00000001403EB120	KiSetFeatureBits	mov word ptr [rsi+63Eh], 101h
PAGELK:00000001403EB1AA	KiSetFeatureBits	mov [rsi+63Eh], al
PAGELK:00000001403EB1CF	KiSetFeatureBits	movzx ecx, byte ptr [rsi+63Eh]
PAGELK:00000001403EBF86	KiInitializeKernel	mov cl, [rbx+63Eh]
PAGELK:00000001403EC30B	KiInitializeKernel	mov al, [rbx+63Eh]
PAGELK:00000001403EC311	KiInitializeKernel	cmp [rcx+63Eh], al
PAGELK:00000001403EEA4F	KiSetFeatureBits	mov [rsi+63Eh], al
PAGELK:00000001403EEA71	KiSetFeatureBits	movzx eax, byte ptr [rsi+63Eh]
PAGELK:00000001403EEA8A	KiSetFeatureBits	mov [rsi+63Eh], al
PAGELK:00000001403EEAB9	KiSetFeatureBits	mov [rsi+63Eh], al
PAGELK:00000001403EEAE3	KiSetFeatureBits	movzx ecx, byte ptr [rsi+63Eh]
PAGELK:00000001403EEB15	KiSetFeatureBits	mov [rsi+63Eh], al
PAGELK:00000001403EEB20	KiSetFeatureBits	movzx ecx, byte ptr [rsi+63Eh]
PAGELK:00000001403EEDEC	KiInitializeKernel	movzx r9d, byte ptr [r10+63Eh]; BugCheckParameter3
PAGELK:00000001403EEE0E	KiInitializeKernel	movzx r9d, byte ptr [rbx+63Eh]; BugCheckParameter3
PAGE:00000001405A2ACD	KiResetBootProcessorApicMask	movzx eax, byte ptr [rdx+63Eh]
INIT:00000001407DDF88	KeStartAllProcessors	movzx ecx, byte ptr [rsi+63Eh]
INIT:00000001407DE3D6	KeStartAllProcessors	movzx eax, byte ptr [rax+63Eh]

You can then inspect these instructions and see if they indeed reference the `_KPRCB` field and not something else.

This is probably one of the best uses for immediate search but it does not replace manual analysis. For example:

1. it may miss references which do not use the value directly but calculate it one way or another;
2. false positives may happen, especially for common or small values
3. the field may be referenced indirectly via a bigger containing structure (e.g. _KPCR includes _KPRCB as a member, so references from _KPCR will have an additional offset).

See also:
IDA Help: Search for next instruction/data with the specified operand[3]

[1] https://hex-rays.com/blog/igors-tip-of-the-week-48-searching-in-ida/
[2] https://arcb.csc.ncsu.edu/~mueller/codeopt/codeopt00/notes/sparc.html
[3] https://www.hex-rays.com/products/ida/support/idadoc/574.shtml