The Beginner's Guide to IDAPython

Version 6.0

By Alexander Hanel

Table of Contents

Introduction 2
Updates 2
Intended Audience & Disclaimer 3
Conventions 3
IDAPython Background 4
Old vs New 4
Python-x86_64 Issues 5
The Basics 6
Segments 7
Functions 8
Extracting Function Arguments 14
Instructions 15
Operands 18
Basic Blocks 21
Structures 24
Enumerated Types 28
Xrefs 30
Searching 35
Selecting Data 41
Comments & Renaming 42
Coloring 49
Accessing Raw Data 50
Patching 51
Input and Output 53
PyQt 56
Batch File Generation` 58
Executing Scripts 60
Yara 61
Unicorn Engine 66
Debugging 77
What's Next? 77
Closing 78
Appendix 78

Unchanged IDC API Names 78
PeFile 79

Introduction

Hello!

This is a book about *IDAPython*.

I originally wrote it as a reference for myself - I wanted a place to go to where I could find examples of functions that I commonly use (and forget) in IDAPython. Since I started this book, I have used it many times as a quick reference to understand syntax or see an example of some code - if you have read my blog[1] you may notice a few familiar faces - lots of scripts that I cover here are result of sophomoric experiments that I documented online.

Over the years I have received numerous emails asking what the best guide for learning IDAPython is. Usually I point them to Ero Carrera's `Introduction to IDAPython` or the example scripts in the IDAPython's public repo[2]. They are excellent sources for learning, but they don't cover some common issues that I have come across. I wanted to create a book that addresses these issues. I feel this book is of value for anyone learning IDAPython or wanting a quick reference for examples and snippets. Being an e-book, it will not be a static document and I plan on updating it in the future on regular basis.

If you come across any issues, typos or have questions please send me an email **alexander< dot >hanel< at >gmail< dot > com** or ping me on Twitter **@nullandnull**.

Updates

- Version 1.0
    - Published
- Version 2.0
    - Table of Contents and closing added
- Version 3.0
    - Grammar fixes provided by Russell V. and added an example of renaming operands.
- Version 4.0
    - Support for IDAPython 7.0
- Version 4.1
    - Bug fixes provided by Minh-Triet Pham Tran @MinhTrietPT
- Version 5.0

[1] hooked-on-mnemonics.blogspot.com/
[2] https://github.com/idapython/src

    - Converted format from Markdown to Microsoft Word.
    - Yara chapter added
    - Coloring chapter added
    - Structure chapter added
    - Enumerated Types chapter added
    - What's next chapter added
    - Fixed bug found by @qmemcpy
    - Added MakeFunction as requested by Minh-Triet Pham Tran
- Version 6.0
    - Support for IDAPython 7.4 and Python 3.
    - Extracting Function Arguments chapter added
    - Basic Blocks chapter added
    - PyQt chapter added
    - Unicorn Engine chapter added
    - Debugging chapter added

Intended Audience & Disclaimer

This book is not intended for beginner reverse engineers. It is also not to serve as an introduction to IDA. If you are new to IDA, I would recommend purchasing Chris Eagles `The IDA PRO Book`. For something more hands on, try taking a training taught by Chris Eagle or Hex-Rays.

There are a couple of prerequisites for readers of this book. You should be comfortable with reading assembly, have a background in reverse engineering and know your way around IDA. If you have hit a point where you have asked yourself "How can I automate this task using IDAPython?" then this book might be for you. If you already have a handful of programming in IDAPython under your belt, then you're probably familiar with the material. That said, it will serve as a handy reference to find examples of commonly used functions or it might solve a problem you come across in the future. It should be stated that my background is in reverse engineering of x86 malware on Windows. Many of the examples provided in this book are derived from common tasks that I come across when reverse engineering malware. After reading this book the reader will feel comfortable with digging into the IDAPython documentation and source code on their own.

Conventions

IDA's Output Windows (command line interface) was used for most of the examples and output. For the sake of brevity some examples do not contain the assignment of the current address to a variable. Usually represented as `ea = here()`. All the code can be cut and paste into the command line or IDA's script command option `Shift-F2`. Reading from beginning to end is the recommended approach for this book. There are several examples that are not explained line by line because it assumed the reader understands the code from previous examples. Different authors call IDAPython's APIs in different ways. Sometimes the code is called as `idc.get_segm_name(ea)` or `get_segm_name(ea)`. This book uses the first style. I have found this convention to be easier to read and debug.

Sometimes when using this convention an error can be thrown.

```
Python>DataRefsTo(here()) # no issues
<generator object refs at 0x05247828>
Python>idautils.DataRefsTo(here()) # causes an exception
Traceback (most recent call last):
  File "<string>", line 1, in <module>
NameError: name 'idautils' is not defined
Python>import idautils   # manual importing of module
Python>idautils.DataRefsTo(here())
<generator object refs at 0x06A398C8>
```

If this happens the module needs to be manually imported as shown above.

IDAPython Background

IDAPython was created in 2004. It was a joint effort by Gergely Erdelyi and Ero Carrera. Their goal was to combine the power of Python with the analysis automation of IDA's IDC C-like scripting language. In the past IDAPython primarily consisted of three separate modules. The first is `idc`. It is a compatibility module for wrapping IDA's IDC functions. The second module is `idautils`. It is a high-level utility functions for IDA. The third module is `idaapi`. It allows access to more low-level data. With the release of 6.95, IDA started to include more modules that cover functionality that historically have been covered by `idaapi`. These newer modules have a naming convention of `ida_*`. A couple of the modules are referenced in this book. One such module is `ida_kernwin.py`. Once the reader has finished this book, I would recommend exploring these modules on your own. They are in `IDADIR\python\ida_*.py`.

Old vs New

In September of 2017 IDA 7.0 was released. This release was a substantial update for HexRays because IDA was ported from x86 to x86_64 binaries. A side effect of this release is that older plugins will need to be recompiled. Even though some major changes happened under the hood for IDAPython (See Hex-Rays' *IDA 7.0: IDAPython backward-compatibility with 6.95 APIs*[3] for more details); older scripts would execute in 7.0. The backwards compatibility from 6.95 to 7.0 is due to a compatibility layer that exists in `IDADIR\python\idc_bc695.py`. The following code is an example of the compatibility layer code.

```
def MakeName(ea, name): return set_name(ea, name, SN_CHECK)
```

[3] https://www.hex-rays.com/products/ida/7.0/docs/idapython_backward_compat_695.shtml

The old IDAPython function `MakeName` has been renamed to `set_name`. If we wanted to quickly print the new API name from `idc_bc695.py` using the command line, we can use the module `inspect`.

```
Python>import inspect
Python>inspect.getsource(MakeName)
def MakeName(ea, name): return set_name(ea, name, SN_CHECK)
```

For user of IDAPython who are familiar with the older naming convention, not all the API names were changed. Some API names cannot be redefined and therefore remain the same. A list of API names that have remained static can be found in the Appendix under *Unchanged IDC API Names*. In version 7.4 of IDA the compatibility layer was turned off by default. It is not recommended, but users of IDA can re-enable it by modifying `IDADIR\cfg\python.cfg` and making sure the `AUTOIMPORT_COMPAT_IDA695` equals `Yes`. Due to the backwards compatibility not being supported in future version of IDA, this book has been written using the "new" API names. As of publication date, the compatibility layer only targets APIs within in `idc.py`. In October of 2019, IDA 7.4 was released. This version provided support for Python 3. Upon release IDA 7.4 supports Python 2 and Python 3 but with the end of life for Python 2.x, it will not be supported in future releases. Since a host can have multiple versions of Python installed, Hex-Rays has provided a tool named `idapyswitch` that exists at `IDADIR\idapyswitch.exe`. Upon execution the tool enumerates all available versions of Python and allows the user to select which version of Python they would like to use.

Python-x86_64 Issues

Some common issues when upgrading from IDA 6.9 to newer versions is when executing older scripts that rely on non-standard modules. Previously installed modules (such as *pefile*[4]) need to be upgraded from x86 to x86_64 to be used in IDA. The easiest way to update them is by executing the following command `C:\>python%version%\python.exe -m pip install <package>`. Executing `import sys; print(sys.path)` from the IDA output window can be used to locate the folder path for the version of Python that IDA is using. As of April 2020, there are installation issues when installing IDAPython with Python 3.8 and 3.81. To resolve this issue please see Hex-Rays blog post *IDA 7.4 and Python 3.8*[5].

For many users, it is common practice to use the function `hex` to print an address. With the upgrade to IDA 7+, users who print addresses using `hex` will no longer have clickable addresses. The address types are now `long` rather than `int`. If you need the printed addresses to be clickable, please use string formatting. The first print address below is a long and is not clickable. The addresses printed using string formatting is printable.

```
Python>ea = idc.get_screen_ea() # get address of cursor
Python>print(hex(ea)) # print unclickable address
0x407e3bL
```

[4] https://github.com/erocarrera/pefile
[5] https://www.hex-rays.com/blog/ida-7-4-and-python-3-8/

```
Python>print("0x%x" % ea) # print clickable address
0x407e3b
```

The Basics

Before we dig too deep, we should define some keywords and go over the structure of IDA's disassembly output. This is commonly seen in GUI using the IDA-View window. We can use the following line of code as an example.

```
.text:00401570                 lea      eax, [ebp+arg_0]
```

The `.text` is the section name and the address is `00401570`. The displayed address is in a hexadecimal format without the `0x` prefix. The instruction `lea` is referred to as a mnemonic. After the mnemonic is the first operand of `eax` and the second operand is `[ebp+arg_0]`. When working with IDAPython APIs, the most common passed variable is an address. In the IDAPython documentation the address is referenced as `ea`. An address can be accessed manually using multiple functions. The most commonly used functions are `idc.get_screen_ea()` or `here()`. These functions return an integer value that contain the address at which the cursor is placed at. If we want to get the minimum address that is present in an IDB we can use `idc.get_inf_attr(INF_MIN_EA)` or to get the maximum address, we can use `idc.get_inf_attr(INF_MAX_EA)`.

```
Python>ea = idc.get_screen_ea()
Python>print("0x%x %s" % (ea, ea))
0x401570 4199792
Python>ea = here()
Python>print("0x%x %s" % (ea, ea))
0x401570 419972
Python>print("0x%x" % idc.get_inf_attr(INF_MIN_EA))
0x401000
Python>print("0x%x" % idc.get_inf_attr(INF_MAX_EA))
0x41d000
```

Each described element in the disassembly output can be accessed by a function in IDAPython. Below is an example of how to access each element. Please recall that we previously stored the address in `ea`.

```
Python>idc.get_segm_name(ea) # get text
.text
Python>idc.generate_disasm_line(ea, 0) # get disassembly
lea eax, [ebp+arg_0]
Python>idc.print_insn_mnem(ea) # get mnemonic
```

```
lea
Python>idc.print_operand(ea,0) # get first operand
eax
Python>idc.print_operand(ea,1) # get second operand
[ebp+arg_0]
```

To get a string representation of the segment's name we use `idc.get_segm_name(ea)` with `ea` being an address within the segment. Printing a string of the disassembly can be done using `idc.generate_disasm_line(ea, 0)`. The arguments are the address stored in `ea` and a flag of `0`. The flag `0` returns the displayed disassembly that IDA discovered during its analysis. `ea` can be any address within the instruction offset range when the `0` flag is passed. To disassemble an exact offset and ignore IDA's analysis a flag of `1` is used. To get the mnemonic or the instruction name we would call `idc.print_insn_mnem(ea)`. To get the operands of the mnemonic we would call `idc.print_operand(ea, long n)`. The first argument is the address and the second-long `n` is the operand index. The first operand is `0`, the second is `1` and each following operand is incremented by one for `n`.

In some situations, it is important to verify an address exists. `idaapi.BADADDR`, `idc.BADADDR` or `BADADDR` can be used to check for valid addresses.

```
Python>idaapi.BADADDR
4294967295
Python>print("0x%x" % idaapi.BADADDR)
0xffffffff
Python>if BADADDR != here(): print("valid address")
valid address
```

Example of `BADADDR` on a 64-bit binary.

```
Python>idc.BADADDR
18446744073709551615
Python>print("0x%x" % idaapi.BADADDR)
0xffffffffffffffff
```

Segments

Printing a single line is not particularly useful. The power of IDAPython comes from iterating through all instructions, cross-referencing addresses and searching for code or data. The last two will be described in more details in further sections. That said, iterating through all segments is a good place to start.

```
Python>for seg in idautils.Segments():\
```

```
print("%s, 0x%x, 0x%x" % (idc.get_segm_name(seg), idc.get_segm_start(seg),
idc.get_segm_end(seg)))
Python>
.textbss, 0x401000, 0x411000
.text, 0x411000, 0x418000
.rdata, 0x418000, 0x41b000
.data, 0x41b000, 0x41c000
.idata, 0x41c000, 0x41c228
.00cfg, 0x41d000, 0x41e000
```

`idautils.Segments()` returns an iterator type object. We can loop through the object by using a `for` loop. Each item in the list is a segment's start address. The address can be used to get the segment name if we pass it as an argument to `idc.get_segm_name(ea)`. The start and end of the segments can be found by calling `idc.get_segm_start(ea)` or `idc.get_segm_end(ea)`. The address or `ea` needs to be within the range of the start or end of the segment. If we didn't want to iterate through all segments but wanted to find the next segment from an offset, we could use `idc.get_next_seg(ea)`. The address passed can be any address within the segment range for which we would want to find the next segment for. If by chance we wanted to get a segment's start address by name, we could use `idc.get_segm_by_sel(idc.selector_by_name(str_SectionName))`. The function `idc.selector_by_name(segname)` returns the segment selector and is passed a single string argument of the segment name. The segment selector is an integer value that starts at 1 and increments for each segment (aka section) in the executable. `idc.get_segm_by_sel(int)` is passed the segment selector and returns the start address of segment.

Functions

Now that we know how to iterate through all segments, we should go over how to iterate through all known functions.

```
Python>for func in idautils.Functions():
    print("0x%x, %s" % (func, idc.get_func_name(func)))
Python>
0x401000, sub_401000
0x401006, w_vfprintf
0x401034, _main
…removed…
0x401c4d, terminate
0x401c53, IsProcessorFeaturePresent
```

`idautils.Functions()` returns a list of known functions. The list contains the start address of each function. `idautils.Functions()` can be passed arguments to search within a range. If we wanted to do this, we would pass the start and end address `idautils.Functions(start_addr, end_addr)`. To get a function's name we use `idc.get_func_name(ea)`. `ea` can be any address within the function boundaries. IDAPython contains a large set of APIs for working with functions. Let us start with a simple function. The semantics of this function is not important, but we should create a mental note of the addresses.

```
.text:0045C7C3 sub_45C7C3      proc near
.text:0045C7C3                 mov     eax, [ebp-60h]
.text:0045C7C6                 push    eax             ; void *
.text:0045C7C7                 call    w_delete
.text:0045C7CC                 retn
.text:0045C7CC sub_45C7C3      endp
```

To get the boundaries we can use `idaapi.get_func(ea)`.

```
Python>func = idaapi.get_func(ea)
Python>type(func)
<class 'ida_funcs.func_t'>
Python>print("Start: 0x%x, End: 0x%x" % (func.start_ea, func.end_ea))
Start: 0x45c7c3, End: 0x45c7cd
```

`idaapi.get_func(ea)` returns a class of `ida_funcs.func_t`. Sometimes it is not always obvious how to use a class returned by a function call. A useful command to explore classes in Python is the `dir(class)` function.

```
Python>dir(func)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__get_points__', '__get_regvars__', '__get_tails__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__swig_destroy__', '__weakref__', '_print', 'analyzed_sp',
'argsize', 'clear', 'color', 'compare', 'contains', 'does_return', 'empty',
'endEA', 'end_ea', 'extend', 'flags', 'fpd', 'frame', 'frregs', 'frsize',
'intersect', 'is_far', 'llabelqty', 'llabels', 'need_prolog_analysis', 'overlaps',
'owner', 'pntqty', 'points', 'referers', 'refqty', 'regargqty', 'regargs',
'regvarqty', 'regvars', 'size', 'startEA', 'start_ea', 'tailqty', 'tails', 'this',
'thisown']
```

From the output we can see the function `start_ea` and `end_ea`. These are used to access the start and end of the function. The end address is not the last address within the last instruction but a byte after the cinstruction. These attributes are only applicable towards the current function. If we wanted to access surrounding functions, we could use `idc.get_next_func(ea)` and `idc.get_prev_func(ea)`. The value of `ea` only needs to be an address within the boundaries of the analyzed function. A caveat with enumerating functions, is that it only works if IDA has identified the block of code as a function. Until the block of code is marked as a function, it is skipped during the function enumeration process. Code that is not marked as a function is labeled red in the legend (colored bar at the top in IDA's GUI). These can be manually fixed or automated using the function `idc.create_insn(ea)`.

IDAPython has a lot of different ways to access the same data. A common approach for accessing the boundaries within a function is using `idc.get_func_attr(ea, FUNCATTR_START)` and `idc.get_func_attr(ea, FUNCATTR_END)`.

```
Python>ea = here()
Python>start = idc.get_func_attr(ea, FUNCATTR_START)
Python>end = idc.get_func_attr(ea, FUNCATTR_END)
Python>cur_addr = start
Python>while cur_addr <= end:
    print("0x%x %s" % (cur_addr, idc.generate_disasm_line(cur_addr, 0)))
    cur_addr = idc.next_head(cur_addr, end)
Python>
0x45c7c3 mov     eax, [ebp-60h]
0x45c7c6 push    eax             ; void *
0x45c7c7 call    w_delete
0x45c7cc retn
```

`idc.get_func_attr(ea, attr)` is used to get the start and end of the function. We then print the current address and the disassembly by using `idc.generate_disasm_line(ea, 0)`. We use `idc.next_head(eax)` to get the start of the next instruction and continue until we reach the end of this function. A flaw to this approach is it relies on the instructions to be contained within the boundaries of the start and end of the function. If there was a jump to an address higher than the end of the function the loop would prematurely exit. These types of jumps are quite common in obfuscation techniques such as code transformation. Since boundaries can be unreliable it is best practice to call `idautils.FuncItems(ea)` to loop through addresses in a function. We will go into more details about this approach in the following section.

Similar to `idc.get_func_attr(ea, attr)` another useful arugument for gathering information about a function is `idc.get_func_attr(ea, FUNCATTR_FLAGS)`. The `FUNCATTR_FLAGS` can be used to retrieve information about a function such as if it is library code or if the function doesn't return a value. There are nine possible flags for a function. If we wanted to enumerate all the flags for all the functions, we could use the following code.

```
Python>import idautils
Python>for func in idautils.Functions():
    flags = idc.get_func_attr(func,FUNCATTR_FLAGS)
```

```
    if flags & FUNC_NORETURN:
        print("0x%x FUNC_NORET" % func)
    if flags & FUNC_FAR:
        print("0x%x FUNC_FAR" % func)
    if flags & FUNC_LIB:
        print("0x%x FUNC_LIB" % func)
    if flags & FUNC_STATIC:
        print("0x%x FUNC_STATIC" % func)
    if flags & FUNC_FRAME:
        print("0x%x FUNC_FRAME" % func)
    if flags & FUNC_USERFAR:
        print("0x%x FUNC_USERFAR" % func)
    if flags & FUNC_HIDDEN:
        print("0x%x FUNC_HIDDEN" % func)
    if flags & FUNC_THUNK:
        print("0x%x FUNC_THUNK" % func)
    if flags & FUNC_LIB:
        print("0x%x FUNC_BOTTOMBP" % func)
Python>
0x401006 FUNC_FRAME
0x40107c FUNC_LIB
0x40107c FUNC_STATIC
…
```

We use `idautils.Functions()` to get a list of all known functions addresses and then we use `idc.get_func_attr(ea, FUNCATTR_FLAGS)` to get the flags. We check the value by using a logical AND (`&`) operation on the returned value. For example, to check if the function does not have a return value, we would use the following comparison `if flags & FUNC_NORET`. Now let us go over all the function flags. Some of these flags are quite common while the other are rare.

FUNC_NORET

This flag is used to identify a function that does not execute a return instruction. It is internally represented as equal to 1. An example of a function that does not return a value can be seen below.

```
CODE:004028F8 sub_4028F8      proc near
CODE:004028F8
CODE:004028F8                 and     eax, 7Fh
CODE:004028FB                 mov     edx, [esp+0]
CODE:004028FE                 jmp     sub_4028AC
CODE:004028FE sub_4028F8      endp
```

Notice how `ret` or `leave` is not the last instruction.

FUNC_FAR

This flag is rarely seen unless reversing software that uses segmented memory. It is internally represented as an integer of 2.

FUNC_USERFAR

This flag is rarely seen and has little documentation. Hex-Rays describes the flag as "user has specified far-ness of the function". It has an internal value of 32.

FUNC_LIB

This flag is used to find library code. Identifying library code is very useful because it is code that typically can be ignored when doing analysis. Its internally represented as an integer value of 4. Below is an example of its usage and functions it has identified.

```
Python>for func in idautils.Functions():
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    if flags & FUNC_LIB:
        print("0x%x FUNC_LIB %s" % (func,idc.get_func_name(func)))
Python>
0x40107c FUNC_LIB ?pre_c_initialization@@YAHXZ
0x40113a FUNC_LIB ?__scrt_common_main_seh@@YAHXZ
0x4012b2 FUNC_LIB start
0x4012bc FUNC_LIB ?find_pe_section@@YAPAU_IMAGE_SECTION_HEADER@@QAEI@Z
0x401300 FUNC_LIB ___scrt_acquire_startup_lock
0x401332 FUNC_LIB ___scrt_initialize_crt
```

FUNC_STATIC

This flag is used to identify library functions with a static `ebp` based frame.

FUNC_FRAME

This flag indicates the function uses a frame pointer `ebp`. Functions that use frame pointers typically start with the standard function prologue for setting up the stack frame.

```
.text:1A716697       push    ebp
.text:1A716698       mov     ebp, esp
```

```
.text:1A71669A      sub     esp, 5Ch
```

FUNC_BOTTOMBP

Like `FUNC_FRAM` this flag is used to track the frame pointer. It identifies functions that base pointer points to the stack pointer.

FUNC_HIDDEN

Functions with the `FUNC_HIDDEN` flag means they are hidden and needs to be expanded to view. If we were to go to an address of a function that is marked as hidden it would automatically be expanded.

FUNC_THUNK

This flag identifies functions that are thunk functions. They are simple functions that jump to another function.

```
.text:1A710606 Process32Next proc near
.text:1A710606      jmp     ds:__imp_Process32Next
.text:1A710606 Process32Next endp
```

It should be noted that a function can consist of multiple flags. The following is an example of a function with multiple flags.

```
0x1a716697 FUNC_LIB
0x1a716697 FUNC_FRAME
0x1a716697 FUNC_HIDDEN
0x1a716697 FUNC_BOTTOMBP
```

Sometimes a section of code or data needs to be defined as a function. For example, the following code hasn't been defined as a function during the analysis phase or has no cross-references.

```
.text:00407DC1
.text:00407DC1         mov     ebp, esp
.text:00407DC3         sub     esp, 48h
.text:00407DC6         push    ebx
```

To define a function, we can use `idc.add_func(start, end)`.

```
Python>idc.add_func(0x00407DC1, 0x00407E90
```

The first argument to `idc.add_func(start, end)` is the start address of the function and the second is the end address of the function. In many instances the end address is not needed, and IDA automatically recognizes the end of the function. The below assembly is the output of executing the above code.

```
.text:00407DC1 sub_407DC1 proc near
.text:00407DC1
.text:00407DC1 SystemInfo= _SYSTEM_INFO ptr -48h
.text:00407DC1 Buffer  = _MEMORY_BASIC_INFORMATION ptr -24h
.text:00407DC1 flOldProtect= dword ptr -8
.text:00407DC1 dwSize  = dword ptr -4
.text:00407DC1
.text:00407DC1         mov     ebp, esp
.text:00407DC3         sub     esp, 48h
.text:00407DC6         push    ebx
```

Extracting Function Arguments

Extracting function arguments is not always a straightforward task in IDAPython. In many instances the calling conventions need to be identified for a function and the arguments must be manually parsed using back-tracing or a similar technique. Due to the vast array of calling conventions[6], this is not always feasible to implement generically. IDAPython does contain a function named `idaapi.get_arg_addrs(ea)` that can be used to get the addresses of arguments if IDA was able to identify the prototype for the called function. This identification is not always present, but it is commonly observed in calls to APIs or within 64bit code. For example, in the following assembly we can see that the API `SendMessage` has four arguments passed to it.

```
.text:000000014001B5FF                js      loc_14001B72B
.text:000000014001B605                mov     rcx, cs:qword_14002D368 ; hWnd
.text:000000014001B60C                xor     r9d, r9d        ; lParam
.text:000000014001B60F                xor     r8d, r8d        ; wParam
.text:000000014001B612                mov     edx, 0BDh ; '½' ; Msg
.text:000000014001B617                call    cs:SendMessageW
.text:000000014001B61D                xor     esi, esi
```

By using `idaapi.get_arg_addrs(ea)` with `ea` being the address of the API, we can retrieve a list of addresses which the arguments were passed.

```
Python>ea = 0x00014001B617
Python>idaapi.get_arg_addrs(ea)
[0x14001b605, 0x14001b612, 0x14001b60f, 0x14001b60c]
```

[6] https://www.agner.org/optimize/calling_conventions.pdf

Instructions

Since we know how to work with functions, it is now time to go over how to access instructions within a function. If we have the address of a function, we can use `idautils.FuncItems(ea)` to get a list of all the addresses.

```
Python>dism_addr = list(idautils.FuncItems(here()))
Python>type(dism_addr)
<type 'list'>
Python>print(dism_addr)
[4573123, 4573126, 4573127, 4573132]
Python>for line in dism_addr: print("0x%x %s" % (line,
idc.generate_disasm_line(line, 0)))
0x45c7c3 mov     eax, [ebp-60h]
0x45c7c6 push    eax             ; void *
0x45c7c7 call    w_delete
0x45c7cc retn
```

`idautils.FuncItems(ea)` returns an iterator type but is cast to a `list`. The list contains the start address of each instruction in consecutive order. Now that we have a good knowledge base for looping through segments, functions, and instructions; let show a useful example. Sometimes when reversing packed code, it is useful to only know where dynamic calls happens. A dynamic call would be a call or jump to an operand that is a register such as `call eax` or `jmp edi`.

```
Python>
for func in idautils.Functions():
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
    dism_addr = list(idautils.FuncItems(func))
    for line in dism_addr:
        m = idc.print_insn_mnem(line)
        if m == 'call' or m == 'jmp':
            op = idc.get_operand_type(line, 0)
            if op == o_reg:
                print("0x%x %s" % (line, idc.generate_disasm_line(line, 0)))
Python>
0x43ebde call    eax             ; VirtualProtect
```

We call `idautils.Functions()` to get a list of all known functions. For each function we retrieve the functions flags by calling `idc.get_func_attr(ea, FUNCATTR_FLAGS)`. If the function is library code or a thunk function the function is passed. Next, we call `idautils.FuncItems(ea)` to get all the addresses within the function. We loop through the list using a for loop. Since we are only interested in `call` and `jmp` instructions we need to get the mnemonic by calling `idc.print_insn_mnem(ea)`. We then use a simple string comparison to check the mnemonic. If the mnemonic is a jump or call, we get the operand type by calling `idc.get_operand_type(ea, n)`. This function returns an integer that is internally called `op_t.type`. This value can be used to determine if the operand is a register, memory reference, etc. We then check if the `op_t.type` is a register. If so, we print the line. Casting the return of `idautils.FuncItems(ea)` into a list is useful because iterators do not have objects such as `len()`. By casting it as a list we could easily get the number of lines or instructions in a function.

```
Python>ea = here()
Python>len(idautils.FuncItems(ea))
Traceback (most recent call last):
  File "<string>", line 1, in <module>
TypeError: object of type 'generator' has no len()
Python>len(list(idautils.FuncItems(ea)))
39
```

In the previous example we used a list that contained all addresses within a function. We looped through each entity to access the next instruction. What if we only had an address and wanted to get the next instruction? To move to the next instruction address we can use `idc.next_head(ea)` and to get the previous instruction address we use `idc.prev_head(ea)`. These functions get the start of the next instruction but not the next address. To get the next address we use `idc.next_addr(ea)` and to get the previous address we use `idc.prev_head(ea)`.

```
Python>ea = here()
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x10004f24 call    sub_10004F32
Python>next_instr = idc.next_head(ea)
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(next_instr, 0)))
0x10004f29 mov     [esi], eax
Python>prev_instr = idc.prev_head(ea)
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(prev_instr, 0)))
0x10004f1e mov     [esi+98h], eax
Python>print("0x%x" % idc.next_addr(ea))
0x10004f25
Python>print("0x%x" % idc.prev_head(ea))
0x10004f23
```

In the dynamic call example, the IDAPython code relies on using a string comparison of `jmp` and `call`. Rather than using a string comparison, we can also decode the instructions using `idaapi.decode_insn(insn_t, ea)`. The first argument is an `insn_t` class from `ida_ua` that is created by calling `ida_ua.insn_t()`. This class is populated with attributes once `idaapi.decode_insn` is called. The second argument is the addresses to be analyzed. Decoding an instruction can be advantageous because working with the integer representation of the instruction can be faster and less error prone. Unfortunately, the integer representation is specific to IDA and cannot be easily ported to other disassembly tools. Below is the same example but using `idaapi.decode_insn(insn_t, ea)` and comparing the integer representation.

```
Python>JMPS = [idaapi.NN_jmp, idaapi.NN_jmpfi, idaapi.NN_jmpni]
Python>CALLS = [idaapi.NN_call, idaapi.NN_callfi, idaapi.NN_callni]
Python>for func in idautils.Functions():
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
    dism_addr = list(idautils.FuncItems(func))
    for line in dism_addr:
        ins = ida_ua.insn_t()
        idaapi.decode_insn(ins, line)
        if ins.itype in CALLS or ins.itype in JMPS:
            if ins.Op1.type == o_reg:
                print("0x%x %s" % (line, idc.generate_disasm_line(line, 0))
Python>
0x43ebde call    eax             ; VirtualProtect
```

The output is the same as the previous example. The first two lines put the constants for `jmp` and `call` into two lists. Since we are not working with the string representation of the mnemonic, we need to be cognizant that a mnemonic (such as `call` or `jmp`) could have multiple values. For example, `jmp` could be represented by `idaapi.NN_jmp` for a jump, `idaapi.NN_jmpfi` for an indirect far jump or `idaapi.NN_jmpni` for an indirect near jump. X86 and X64 instruction types all start with NN. To explore all 1,700+ instruction types we can execute `[name for name in dir(idaapi) if "NN" in name]` in the command line or review them in IDA's SDK file allins.hpp. Once we have the instructions in lists, we use a combination of `idautils.Functions()` and `get_func_attr(ea, FUNCATTR_FLAGS)` to get all applicable functions while ignoring libraries and thunks. We get each instruction in a function by calling `idautils.FuncItems(ea)`. This is where are newly introduced function `idaapi.decode_insn(ins, ea)` is called. This function takes the address of instruction we want decoded. Once it is decoded, we can access different properties of the instruction by accessing the `insn_t` class within the variable `ins`.

```
Python>dir(ins)
```

```
['Op1', 'Op2', 'Op3', 'Op4', 'Op5', 'Op6', 'Operands', '__class__', '__del__',
'__delattr__', '__dict__', '__doc__', '__format__', '__get_auxpref__',
'__get_operand__', '__get_ops__', '__getattribute__', '__getitem__', '__hash__',
'__init__', '__iter__', '__module__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__set_auxpref__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__swig_destroy__', '__weakref__', 'add_cref', 'add_dref',
'add_off_drefs', 'assign', 'auxpref', 'create_op_data', 'create_stkvar', 'cs',
'ea', 'flags', 'get_canon_feature', 'get_canon_mnem', 'get_next_byte',
'get_next_dword', 'get_next_qword', 'get_next_word', 'insnpref', 'ip', 'is_64bit',
'is_canon_insn', 'is_macro', 'itype', 'ops', 'segpref', 'size', 'this', 'thisown']
```

As we can see from the `dir()` command `ins` have a good number of attributes. The operand type is accessed by using `ins.Op1.type`. Please note that the operand index starts at 1 rather than 0 which is different than `idc.get_operand_type(ea,n)`.

Operands

Operand types are commonly used so it is beneficial to go over all the types. As previous stated we can use `idc.get_operand_type(ea,n)` to get the operand type. `ea` is the address and `n` is the index. There are eight different type of operand types.

o_void

If an instruction does not have any operands it returns 0.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0xa09166 retn
Python>print(idc.get_operand_type(ea,0))
0
```

o_reg

If an operand is a general register it returns this type. This value is internally represented as 1.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0xa09163 pop     edi
Python>print(idc.get_operand_type(ea,0))
1
```

o_mem

If an operand is direct memory reference it returns this type. This value is internally represented as 2. This type is useful for finding references to DATA.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0xa05d86 cmp     ds:dword_A152B8, 0
```

```
Python>print(idc.get_operand_type(ea,0))
2
```

o_phrase

This operand is returned if the operand consists of a base register and/or an index register. This value is internally represented as 3.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x1000b8c2 mov     [edi+ecx], eax
Python>print(idc.get_operand_type(ea,0))
3
```

o_displ

This operand is returned if the operand consists of registers and a displacement value. The displacement is an integer value such 0x18. It is commonly seen when an instruction accesses values in a structure. Internally it is represented as a value of 4.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0xa05dc1 mov     eax, [edi+18h]
Python>print(idc.get_operand_type(ea,1))
4
```

o_imm

Operands that are a value such as an integer of 0xC are of this type. Internally it is represented as 5.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0xa05da1 add     esp, 0Ch
Python>print(idc.get_operand_type(ea,1))
5
```

o_far

This operand is not common when reversing x86 or x86_64. It is used to find operands that are accessing immediate far addresses. It is represented internally as 6

o_near

This operand is not common when reversing x86 or x86_64. It is used to find operands that are accessing immediate near addresses. It is represented internally as 7.

Example

Sometimes when reversing a memory dump of an executable the operands are not recognized as an offset.

```
seg000:00BC1388             push    0Ch
seg000:00BC138A             push    0BC10B8h
seg000:00BC138F             push    [esp+10h+arg_0]
seg000:00BC1393             call    ds:_strnicmp
```

The second value being pushed is a memory offset. If we were to right click on it and change it to a data type; we would see the offset to a string. This is okay to do once or twice but after that we might as well automate the process.

```
min = idc.get_inf_attr(INF_MIN_EA)
max = idc.get_inf_attr(INF_MAX_EA)
# for each known function
for func in idautils.Functions():
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    # skip library & thunk functions
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
    dism_addr = list(idautils.FuncItems(func))
    for curr_addr in dism_addr:
        if idc.get_operand_type(curr_addr, 0) == 5 and \
                (min < idc.get_operand_value(curr_addr, 0) < max):
            idc.OpOff(curr_addr, 0, 0)
        if idc.get_operand_type(curr_addr, 1) == 5 and \
                (min < idc.get_operand_value(curr_addr, 1) < max):
            idc.op_plain_offset(curr_addr, 1, 0)
```

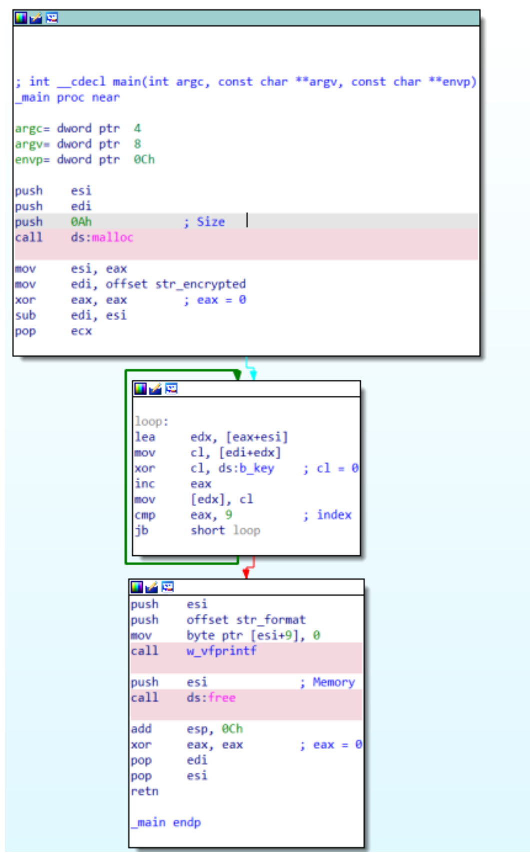
After running the above code, we would now see the string.

```
seg000:00BC1388             push    0Ch
seg000:00BC138A             push    offset aNtoskrnl_exe  ; "ntoskrnl.exe"
seg000:00BC138F             push    [esp+10h+arg_0]
seg000:00BC1393             call    ds:_strnicmp
```

At the start we get the minimum and maximum address by calling `idc.get_inf_attr(INF_MIN_EA)` and `idc.get_inf_attr(INF_MAX_EA)` We loop through all functions and instructions. For each instruction we check if the operand type is of `o_imm` and is represented internally as the number 5. `o_imm` types are values such as an integer or an offset. Once a value is found we read the value by calling `idc.get_operand_value(ea,n)`. The value is then checked to see if it is in the range of the minimum and maximum addresses. If so, we use `idc.op_plain_offset(ea, n, base)` to convert the operand to an offset. The first argument `ea` is the address, `n` is the operand index and base is the base address. Our example only needs to have a base of zero.

Basic Blocks

A basic block is a straight-line code sequence which has no branches, consists of a single-entry point and a single-exit point. Basic blocks are useful when doing analysis of a program's control flow. IDA's representation of basic blocks is commonly observed when using the graph disassembly view of a function. Some notable examples of using basic blocks for analysis is for identifying loops or control flow obfuscation. When a basic block transfers control to another block, the next block is called the successor and the previous block is called the predecessors. The following flow graph is a function that decrypts a string with single byte XOR.



Since the image is difficult to see the code and addresses, the assembly output can be found below. The function contains three blocks with the basic block leader at offset `0x0401034`, `0x040104A` and `0x0040105E`. The XOR loop starts at `0x040104A`, the index is evaluated at offset `0x0401059` and continues to `0x040105E` once the XOR loop is done.

```
.text:00401034             push    esi
.text:00401035             push    edi
.text:00401036             push    0Ah             ; Size
.text:00401038             call    ds:malloc
.text:0040103E             mov     esi, eax
.text:00401040             mov     edi, offset str_encrypted
.text:00401045             xor     eax, eax        ; eax = 0
.text:00401047             sub     edi, esi
.text:00401049             pop     ecx
.text:0040104A
.text:0040104A loop:                                     ; CODE XREF: _main+28↓j
.text:0040104A             lea     edx, [eax+esi]
.text:0040104D             mov     cl, [edi+edx]
.text:00401050             xor     cl, ds:b_key    ; cl = 0
.text:00401056             inc     eax
.text:00401057             mov     [edx], cl
.text:00401059             cmp     eax, 9          ; index
.text:0040105C             jb      short loop
.text:0040105E             push    esi
.text:0040105F             push    offset str_format
.text:00401064             mov     byte ptr [esi+9], 0
.text:00401068             call    w_vfprintf
.text:0040106D             push    esi             ; Memory
.text:0040106E             call    ds:free
.text:00401074             add     esp, 0Ch
.text:00401077             xor     eax, eax        ; eax = 0
.text:00401079             pop     edi
.text:0040107A             pop     esi
.text:0040107B             retn
.text:0040107B _main       endp
```

If we extracted the offset of the single byte XOR encryption at `0x0401050`, we use the following code to get the start and end of the basic block in which the XOR occurs and get the successor and predecessor basic blocks.

```
ea = 0x0401050
f = idaapi.get_func(ea)
fc = idaapi.FlowChart(f, flags=idaapi.FC_PREDS)
for block in fc:
```

```
    print("ID: %i Start: 0x%x End: 0x%x" % (block.id, block.start_ea,
block.end_ea))
    if block.start_ea <= ea < block.end_ea:
        print("  Basic Block selected")
    successor = block.succs()
    for addr in successor:
        print("  Successor: 0x%x" % addr.start_ea)
    pre = block.preds()
    for addr in pre:
        print("  Predecessor: 0x%x" % addr.end_ea)
    if ida_gdl.is_ret_block(block.type):
        print("  Return Block")
```

The first instruction assigns the single byte XOR offset to the variable `ea`. The function `idaapi.FlowChart(f=None, bounds=None, flags=0)` requires a class of `func_t` to be passed as the first argument. In order to get the class, we call `idaapi.get_func(ea)`. The argument `bounds` can be passed a tuple with the first item being the start address and the second being the end address `bounds=(start, end)`. In IDA 7.4, The third argument `flags` must be set to `idaapi.FC_PREDS` if the predecessor is to be calculated. The variable `fc` contains an `ida_gdl.FlowChart` object that can be looped through to iterate over all the blocks. Each block contains the following attributes.

- `id` each basic block within a function has a unique index. The first block starts with an id of `0`.
- `type` the type describes the basic block with the following types
  - `fcb_normal` represents a normal block and has an internal value of `0`
  - `fcb_indjump` is a block that ends with an indirect jump and has an internal value of `1`
  - `fcb_ret` is a return block and has an internal value of `2`. `ida_gdl.is_ret_block(block.type)` can also be used to determine if the block is of `fcb_ret` type
  - `fcb_cndret` is a conditional return bloc and has a value of `3`
  - `fcb_noret` is a block with no return and has an internal value of 4
  - `fcb_enoret` is a block with no return that does not belong to a function and has an internal value of `5`
  - `fcb_extern` is an external normal block and has an internal value of `6`
  - `fcb_error` is a block that passes execution past the function end and has an internal value of `7`
- `start_ea` is the start address of the basic block.
- `end_ea` is the end address of the basic block. The end address of the basic block is not the last instruction address but the offset following it.
- `preds` is a function that returns a generator which contains all the predecessor addresses.
- `succs` is a function that returns a generator which contains all the successor address.

After `idaapi.FlowChart` is called, each basic block is iterated through. The id, start address and end address is printed. To locate the block that ea is within, ea is compared to be greater than or equal to the start of the basic block by comparing `block.start_ea` and less then the end of the basic block by comparing `block.end_ea`. The variable name `block` was arbitrarily chosen. To get a generator of all the offset(s) that are successor, we call `block.succs()`. Each item in the `succs` generator is looped through and printed. To get a generator of all the offset(s) that are predecessor, we can call `block.preds()`. Each item in the `preds` generator is looped through and printed. The last if statement calls `ida_gdl.is_ret_block(btype)` to determine if the block is a return type. The output of the script can be seen below.

```
ID: 0 Start: 0x401034 End: 0x40104a
  Successor: 0x40104a
ID: 1 Start: 0x40104a End: 0x40105e
  Basic Block selected
  Successor: 0x40105e
  Successor: 0x40104a
  Predecessor: 0x40104a
  Predecessor: 0x40105e
ID: 2 Start: 0x40105e End: 0x40107c
  Predecessor: 0x40105e
  Return Block
```

The basic block with an ID of 1 is a loop is why it happens multiple successors and predecessors.

Structures

Structure layout, structure names and types are removed from the code during the compilation process. Reconstructing structures and properly labeling the member names can aid tremendously in the reversing process. The following is a snippet[7] of assembly commonly observed in x86 shellcode. The complete code traversers structures within the thread environment block (TEB) and the process environmental block (PEB) to find the base address of `kernel32.dll`.

```
seg000:00000000         xor     ecx, ecx
seg000:00000002         mov     eax, fs:[ecx+30h]
seg000:00000006         mov     eax, [eax+0Ch]
seg000:00000009         mov     eax, [eax+14h]
```

The next step typically observed is traversing the Portable Executable file format to lookup Window APIs. *This technique was first documented* by The Last Stage of Delirium in their paper *Win32 Assembly Components*[8] back in 2002. With all the different structures being parsed it is easy to get lost unless the structure offsets are labeled. As can be seen in the following code, even a couple structures labeled can be helpful.

[7] https://gist.github.com/tophertimzen/5d32f255292a0201853cb7009fc55fba
[88] http://www.lsd-pl.net/winasm.pdf

```
seg000:00000000        xor     ecx, ecx
seg000:00000002        mov     eax, fs:[ecx+_TEB.ProcessEnvironmentBlock]
seg000:00000006        mov     eax, [eax+PEB.Ldr]
seg000:00000009        mov     eax, [eax+PEB_LDR_DATA.InMemoryOrderModuleList.Flink]
seg000:0000000C        mov     eax, [eax+ecx]
```

We can use the following code to label the offsets to their corresponding structure names.

```
status = idc.add_default_til("ntapi")
if status:
    idc.import_type(-1, "_TEB")
    idc.import_type(-1, "PEB")
    idc.import_type(-1, "PEB_LDR_DATA")
    ea = 2
    teb_id = idc.get_struc_id("_TEB")
    idc.op_stroff(ea, 1, teb_id, 0)
    ea = idc.next_head(ea)
    peb_ldr_id = idc.get_struc_id("PEB_LDR_DATA")
    idc.op_stroff(ea, 1, peb_ldr_id, 0)
    ea = idc.next_head(ea)
    idc.op_stroff(ea, 1, peb_ldr_id, 0)
```

The first line is to load the type library (TIL) by calling `idc.add_default_til(name)`. For individuals not familiar with TIL, they are IDA's own format of C/C++ header files. They contain definitions for structures, enums, unions and other data types. The different TILs can be explored manually by opening the Type Library Window (`SHIFT+F11`). `idc.add_default_til(name)` returns the status of if the library could be loaded or not. If the TIL could be loaded, it returns `1` (True) or `0` (False) if the library was loaded or not. It is a good habit to add this check to your code. IDA does not always identify the compiler to import the TIL or forgetting that we manually loaded the TIL. After the TIL is loaded, the individual definitions from the TIL need to be imported into the IDB. To import the individual definitions, we call `idc.import_type(idx, type_name)`. The first argument is `idx`, which is the index of the type. Each type has an index and id. An `idx` of `-1` signals that the type should be added to the end of IDA's imported types list. The index of a type can be change so relying on the index is not always reliable. An `idx` of `-1` is the most used argument. The three types that are added to the IDB in the above code are `_TEB`, `PEB` and `PEB_LDR_DATA`.

The variable `ea` is assigned the value `2`. After the assignment, we get the id of imported type by calling `idc.get_struc_id(string_name)`. The string `"_TEB"` is passed to `idc.get_struct_id` which returns the struct ID as an integer. The struct id is assigned to `teb_id`. To apply the member name `“ProcessEnvironmentBlock”` to the structure offset (0x30) we can use `idc.op_stroff(ea, n, strid, delta)`. `op_stroff` takes 4 arguments. The first argument is the address (`ea`) of the instructions that contain the offset that is going to be labeled. The second argument `n` is the operand number. In our example, since we are wanting to change the label the 0x30 in `mov eax, fs:[ecx+30h]` we need to pass a value of 1 for the second operand. The third argument is the type id that needs to be used for converting the offset to a structure. The last argument is the delta between the structures base and the pointer into the structure. This delta typically has a value of 0. The function `idc.op_stroff` is used to add the structure names to the offsets. The code then calls `idc.next_head(ea)` to get the next instruction address and then use the same previously described process to label another two structures.

Along with using IDA's built in TIL to access structures, we can create our own structure. For this example, we are going to pretend that IDA did not have a type definition for `PEB_LDR_DATA`. Instead of using IDA, we had to dump the type definition using *Windbg* using the command `dt nt!_PEB_LDR_DATA`. The output of this command can be seen below.

```
0:000> dt nt!_PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
   +0x000 Length           : Uint4B
   +0x004 Initialized      : UChar
   +0x008 SsHandle         : Ptr64 Void
   +0x010 InLoadOrderModuleList : _LIST_ENTRY
   +0x020 InMemoryOrderModuleList : _LIST_ENTRY
   +0x030 InInitializationOrderModuleList : _LIST_ENTRY
   +0x040 EntryInProgress  : Ptr64 Void
   +0x048 ShutdownInProgress : UChar
   +0x050 ShutdownThreadId : Ptr64 Void
```

Note: These fields should be static on your machine but do not worry if they differ. *This can change over time with Microsoft adding new fields*. Viewing the output, we can see the offset, name, and type. This is enough information to create our own type. The following code checks if a struct named `my_peb_ldr_data` is present. If the struct is present, the code deletes the struct, creates a new one and then adds the struct member fields from `nt!_PEB_LDR_DATA`.

```
sid = idc.get_struc_id("my_peb_ldr_data")
if sid != idc.BADADDR:
      idc.del_struc(sid)
sid = idc.add_struc(-1, "my_peb_ldr_data", 0)
idc.add_struc_member(sid, "length", 0, idc.FF_DWORD, -1, 4)
idc.add_struc_member(sid, "initialized", 4, idc.FF_DWORD, -1, 4)
idc.add_struc_member(sid, "ss_handle", -1, idc.FF_WORD, -1, 2)
idc.add_struc_member(sid, "in_load_order_module_list", -1, idc.FF_DATA, -1, 10)
idc.add_struc_member(sid, "in_memory_order_module_list", -1, idc.FF_QWORD +
idc.FF_WORD, -1, 10)
idc.add_struc_member(sid, "in_initialization_order_module_list", -1, idc.FF_QWORD +
idc.FF_WORD, -1, 10)
idc.add_struc_member(sid, "entry_in_progress", -1, idc.FF_QWORD, -1, 8)
idc.add_struc_member(sid, "shutdown_in_progress", -1, idc.FF_WORD, -1, 2)
```

```
idc.add_struc_member(sid, "shutdown_thread_id", -1, idc.FF_QWORD, -1, 8)
```

The first step in our code, calls `idc.get_struc_id(struct_name)` to return the id of the struct by name. if there is struct without a name of `"my_peb_ldr_data"`, `idc.get_struct_id` returns `idc.BADADDR`. If the struct id is not `idc.BADADDR`, then we know a struct with a name of "my_peb_ldr_data" already exists. For this example, we delete the struct by calling `idc.del_struc(sid)`. It takes a single argument of the struct id. To create a struct the code calls `idc.add_struc(index, name, is_union)`. The first argument is the index of the new structure. As with `idc.import_type`, it is best practice to pass a value of -1. This specifies that IDA should use the next biggest index for an id. The second argument passed to `idc.add_struc` is the struct name. The third argument of `is_union` is a bool that defines if the newly created struct is a union. In the code above, we pass a value of 0 to specify it is not a union. Members of the struct can be labeled by calling `idc.add_struc_member(sid, name, offset, flag, typeid, nbytes)`. *Note: `idc.add_struc_member` has more arguments but since they are used for more complex definitions, we will not be covering them. If you are interested in how to create more complex definitions, I would recommend digging into the IDAPython source code later*. The first argument is the struct id previously assigned to the variable `sid`. The second argument is a string of the member name. The third argument is the `offset`. The offset can be `-1` to add to the end of the structure or an integer value to specify an offset. The fourth argument is the `flag`. A flag specifies the data type (word, float, etc). The flag available flag data types can be seen below.

```
FF_BYTE       0x00000000       // byte
FF_WORD       0x10000000       // word
FF_DWORD      0x20000000       // dword
FF_QWORD      0x30000000       // qword
FF_TBYTE      0x40000000       // tbyte
FF_STRLIT     0x50000000       // ASCII    ?
FF_STRUCT     0x60000000       // Struct   ?
FF_OWORD      0x70000000       // octaword (16 bytes/128 bits)
FF_FLOAT      0x80000000       // float
FF_DOUBLE     0x90000000       // double
FF_PACKREAL   0xA0000000       // packed decimal real
FF_ALIGN      0xB0000000       // alignment directive
FF_CUSTOM     0xD0000000       // custom data type
FF_YWORD      0xE0000000       // ymm word (32 bytes/256 bits)
FF_ZWORD      0xF0000000       // zmm word (64 bytes/512 bits)
FF_DATA       0x400            // data
```

The fifth argument is the `typeid` and is used for more complex definitions. For our examples, it has a value of `-1`. The last argument is the number of bytes (`nbyte`) to allocate. It is important that the flag and `nbytes` are equal in size. If a dword with a flag of `idc.FF_DWORD` is used, a size of 4 must be specified. If not, IDA does not create the member. This can be a tricky bug to catch because IDA does not throw any warnings. A combination of flags can be used. For example, `idc.FF_QWORD + idc.FF_WORD` is used to specify a size of `10` in the creation of the "in_memory_order_module_list" member. If a flag of `idc.FF_DATA` is passed than any size can be used without having to combining and adding other flags. We'd seen the following if we viewed the newly created structure in IDA Structure Window.

```
00000000 my_peb_ldr_data struc ; (sizeof=0x3A, mappedto_139)
00000000 length  dd ?
00000004 initialized dd ?
00000008 ss_handle dw ?
0000000A in_load_order_module_list db 10 dup(?)
00000014 in_memory_order_module_list dt ?
0000001E in_initialization_order_module_list dt ?
00000028 entry_in_progress dq ?
00000030 shutdown_in_progress dw ?
```

Enumerated Types

A simplified description of enumerated types; is it's a way of using symbolic constants to represent a meaningful name. Enumerated types (aka Enums) are commonplace when calling system APIs. When calling `CreateFileA` on Windows, the desired access of `GENERIC_READ` is represented as the constant `0x80000000`. Unfortunately, the names are stripped during the compilation process. Repopulating the constants with meaningful names aids in the reverse engineering process. When reversing engineering malware, it is not uncommon to see constants that represent hashes of API names. This technique is used to obfuscate API calls from static analysis. The following code is an example of the technique.

```
seg000:00000018         push    0CA2BD06Bh  ; ROR 13 hash of CreateThread
seg000:0000001D         push    dword ptr [ebp-4]
seg000:00000020         call    lookup_hash
seg000:00000025         push    0
seg000:00000027         push    0
seg000:00000029         push    0
seg000:0000002B         push    4C30D0h    ; StartAddress
seg000:00000030         push    0
seg000:00000032         push    0
seg000:00000034         call    eax        ; CreateThread
```

The value `0xCA2BD06B` is the hash of `"CreateThread"`. The hashing is created using a combination of looping through each character, shifting the bits of the byte by 13 using ROR and storing the results to create the hash. This technique is commonly referred to as z0mbie hashing or ROR-13. Since the hash is in a way a symbolic name of `"CreateThread"`, it is a practical example of when to use enums.

Since we already know that the hash `0xCA2BD06B` is the string `"CreateThread"` we could just create the enum. What if we didn't know what API name the hash represented? Then we would need some way to hash all exported symbol names in some Windows DLL. For brevity sake, we can cheat and say the DLL is `kernel32.dll`. To export the symbol names from, kernel32.dll we can use pefile. Please see the *Appendix* for a short example on the most common use case of using pefile. Then we need a way to replicate the hashing algorithm. For the below code, we will be using a modified version of Rolf Rolles (see section *What's Next*) implementation[9] of z0mbie hash and pefile. The code was designed so it can be easily modified by the reader to match any hash or to add all hashes.

```
import pefile

def ror32(val, amt):
    return ((val >> amt) & 0xffffffff) | ((val << (32 - amt)) & 0xffffffff)

def add32(val, amt):
    return (val + amt) & 0xffffffff

def z0mbie_hash(name):
    hash = 0
    for char in name:
        hash = add32(ror32(hash, 13), ord(char) & 0xff)
    return hash

def get_name_from_hash(file_name, hash):
    pe = pefile.PE(file_name)
    for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
        if z0mbie_hash(exp.name) == hash:
            return exp.name

api_name = get_name_from_hash("kernel32.dll", 0xCA2BD06B)
if api_name:
    id = idc.add_enum(-1, "z0mbie_hashes", idaapi.hexflag())
    idc.add_enum_member(id, api_name, 0xCA2BD06B, -1)
```

The first line imports `pefile` into IDA. The two functions `ror32` and `add32` are responsible for replicating the ROR instruction. The function `z0mbie_hash(name)` takes a single argument of the string that is to be hashed and returns the hash. The last function `get_name_from_hash(file_path, hash)` takes two arguments. The first argument is the file path of the DLL that symbols are to be hashed. The second argument is the hash value that we are

[9] http://www.openrce.org/blog/view/681/Shellcode_Analysis

searching for the name of. The function returns the string name. The first line in this function calls `pefile.PE(file_path)` to load and parse `kernel32.dll`. The pefile PE instance is saved into the variable `pe`. Each symbol within the DLL is iterated through by looping through each item in `pe.DIRECTORY_ENTRY_EXPORT.symbols`. This field contains the name, address, and other attributes for each exported symbol in the DLL. The symbol name is hashed by calling `z0mbie_hash(exp.name)` and then compared. If a match happens, the symbol name is returned and assigned to `api_name`. At this point in the code is when the creation and adding of the enum is done. The first step in adding an enum is creating the enum id. This is done by calling `idc.add_enum(idx, name, flag)`. The first argument is `idx` or serial number for the new enum. A value of -1 assigns the next available id. The second argument is the name of the enum. The last argument is the `flag` which is `idaapi.hexflag()`. After executing the code if we were to press the shortcut M while highlighting the value `0xCA2BD06B` in IDA, we would see the string `“CreateThread”` as a symbolic constant option. The following code is the code we saw previously with the hash now a symbolic constant.

```
seg000:00000015          mov     [ebp-4], ebx
seg000:00000018          push    CreateThread      ; ROR 13 hash of CreateThread
seg000:0000001D          push    dword ptr [ebp-4]
```

Xrefs

Being able to locate cross-references (aka xrefs) to data or code is a common analysis task. Locating Xrefs is important because they provide locations of where certain data is being used or where a function is being called from. For example, what if we wanted to locate all the address where `WriteFile` was called from. By using Xrefs, all we would need to do is locate the address of `WriteFile` by name and then find all xrefs to it.

```
Python>wf_addr = idc.get_name_ea_simple("WriteFile")
Python>print("0x%x %s" % (wf_addr, idc.generate_disasm_line(wf_addr, 0)))
0x1000e1b8 extrn WriteFile:dword
Python>for addr in idautils.CodeRefsTo(wf_addr, 0):\
    print("0x%x %s" % (addr, idc.generate_disasm_line(addr, 0)))
0x10004932 call    ds:WriteFile
0x10005c38 call    ds:WriteFile
0x10007458 call    ds:WriteFile
```

In the first line we get the address of the API `WriteFile` by using `idc.get_name_ea_simple(str)`. This function returns the address of the API. We print out the address of `WriteFile` and its string representation. Then loop through all code cross references by calling `idautils.CodeRefsTo(ea, flow)`. It returns an iterator that can be looped through. `ea` is the address that we would like to have cross-referenced to. The argument flow is a `bool`. It is used to specify to follow normal code flow or not. Each cross reference to the address is then displayed. A quick note about the use of `idc.get_name_ea_simple(str)`. All renamed functions and APIs in an IDB can be accessed by calling `idautils.Names()`. This function returns an iterator object which can be looped through to print or access the names. Each named item is a tuple of `(ea,` `str_name)`.

```
Python>[x for x in Names()]
[(268439552, 'SetEventCreateThread'), (268439615, 'StartAddress'), (268441102,
'SetSleepClose'),....]
```

If we wanted to get where code was referenced from, we would use `idautisl.CodeRefsFrom(ea,flow)`. For example, let us get the address of where `0x10004932` is referenced from.

```
Python>ea = 0x10004932
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x10004932 call    ds:WriteFile
Python>for addr in idautils.CodeRefsFrom(ea, 0):\
    print("0x%x %s" % (addr, idc.generate_disasm_line(addr, 0)))
Python>
0x1000e1b8 extrn WriteFile:dword
```

If we review the `idautils.CodeRefsTo(ea, flow)` example we see the address `0x10004932` is a `to` address to `WriteFile`. `idautils.CodeRefsTo(ea, flow)` and `idautils.CodeRefsFrom(ea, flow)` are used to search for cross references to and from code. A limitation of using `idautils.CodeRefsTo(ea, flow)` is that APIs that are imported dynamically and then manually renamed, do not show up as code cross-references. Say we manually rename a dword address to `"RtlCompareMemory"` using `idc.set_name(ea, name, SN_CHECK)`.

```
Python>print("0x%x" % (ea)
0xa26c78
Python>idc.set_name(ea, "RtlCompareMemory", SN_CHECK)
True
Python>for addr in idautils.CodeRefsTo(ea, 0):\
    print("0x%x %s" % (addr, idc.generate_disasm_line(addr, 0)))
```

IDA does not label these APIs as code cross references. A little later we will describe a generic technique to get all cross references. If we wanted to search for cross references to and from data, we could use `idautils.DataRefsTo(e)` or `idautils.DataRefsFrom(ea)`.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x1000e3ec db 'vnc32',0
Python>for addr in idautils.DataRefsTo(ea):\
      print("0x%x %s" % (addr, idc.generate_disasm_line(addr, 0)))
0x100038ac push    offset aVnc32         ; "vnc32"
```

`idautils.DataRefsTo(ea)` takes an argument of the address and returns an iterator of all the addresses that cross reference to the data.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x100038ac push    offset aVnc32         ; "vnc32"
Python>for addr in idautils.DataRefsFrom(ea):\
      print("0x%x %s" % (addr, idc.generate_disasm_line(addr, 0)))
0x1000e3ec db 'vnc32',0
```

To do the opposite and show the from address we call `idautils.DataRefsFrom(ea)`, pass the address as an argument. Which returns an iterator of all the addresses that cross reference back to the data. The different usage of code and data can be a little confusing. Let's describe a more generic technique. This approach can be used to get all cross references to an address by calling a single function. We can get all cross references to an address using `idautils.XrefsTo(ea, flags=0)` and get all cross references from an address by calling `idautils.XrefsFrom(ea, flags=0)`.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x1000eee0 unicode 0, <Path>,0
Python>for xref in idautils.XrefsTo(ea, 1):
         print("%i %s 0x%x 0x%x %i" % (xref.type, idautils.XrefTypeName(xref.type),
xref.frm, xref.to, xref.iscode))
Python>
1 Data_Offset 0x1000ac0d 0x1000eee0 0
Python>>print("0x%x %s" % (xref.frm, idc.generate_disasm_line(xref.frm, 0))
0x1000ac0d push    offset KeyName        ; "Path"
```

The first line displays our address and a string named `“Path”`. We use `idautils.XrefsTo(ea, 1)` to get all cross references to the string. We then use `xref.type` to print the `xrefs` type value. `idautils.XrefTypeName(xref.type)` is used to print the string representation of this type. There are twelve different documented reference type values. The value can be seen on the left and its corresponding name can be seen below.

```
0  = 'Data_Unknown'
1  = 'Data_Offset'
2  = 'Data_Write'
3  = 'Data_Read'
4  = 'Data_Text'
5  = 'Data_Informational'
16 = 'Code_Far_Call'
17 = 'Code_Near_Call'
```

```
18 = 'Code_Far_Jump'
19 = 'Code_Near_Jump'
20 = 'Code_User'
21 = 'Ordinary_Flow'
```

The `xref.frm` prints out the from address and `xref.to` prints out the two address. `xref.iscode` prints if the xref is in a code segment. In the previous example we had the flag of `idautils.XrefsTo(ea, 1)` set to the value 1. If the flag is zero than any cross reference is displayed. We can use the following block of assembly to illustrate this point.

```
.text:1000AAF6           jnb     short loc_1000AB02   ; XREF
.text:1000AAF8           mov     eax, [ebx+0Ch]
.text:1000AAFB           mov     ecx, [esi]
.text:1000AAFD           sub     eax, edi
.text:1000AAFF           mov     [edi+ecx], eax
.text:1000AB02
.text:1000AB02 loc_1000AB02:                           ; ea is here()
.text:1000AB02           mov     byte ptr [ebx], 1
```

We have the cursor at `0x1000AB02`. This address has a cross reference from `0x1000AAF6`, but it also has second cross reference to `0x1000AAF6`.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x1000ab02 mov     byte ptr [ebx], 1
Python>for xref in idautils.XrefsTo(ea, 1):
         print("%i %s 0x%x 0x%x %i" % (xref.type, idautils.XrefTypeName(xref.type),
xref.frm, xref.to, xref.iscode))
Python>
19 Code_Near_Jump 0x1000aaf6 0x1000ab02 1
Python>for xref in idautils.XrefsTo(ea, 0):
         print("%i %s 0x%x 0x%x %i" % (xref.type, idautils.XrefTypeName(xref.type),
xref.frm, xref.to, xref.iscode))
Python>
21 Ordinary_Flow 0x1000aaff 0x1000ab02 1
19 Code_Near_Jump 0x1000aaf6 0x1000ab02 1
```

The second cross reference is from `0x1000AAFF` to `0x1000AB02`. Cross references do not have to be caused by branch instructions. They can also be caused by normal ordinary code flow. If we set the flag to 1, `Ordinary_Flow` reference types will not be added. Now back to our `RtlCompareMemory` example from earlier. We can use `idautils.XrefsTo(ea, flow)` to get all cross references.

```
Python>print("0x%x" % ea)
0xa26c78
Python>idc.set_name(ea, "RtlCompareMemory", SN_CHECK)
True
Python>for xref in idautils.XrefsTo(ea, 1):
         print("%i %s 0x%x 0x%x %i" % (xref.type, idautils.XrefTypeName(xref.type),
xref.frm, xref.to, xref.iscode))
Python>
3 Data_Read 0xa142a3 0xa26c78 0
3 Data_Read 0xa143e8 0xa26c78 0
3 Data_Read 0xa162da 0xa26c78 0
```

Getting all cross references can be a little verbose sometimes.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0xa21138 extrn GetProcessHeap:dword
Python> for xref in idautils.XrefsTo(ea, 1):
         print("%i %s 0x%x 0x%x %i" % (xref.type, idautils.XrefTypeName(xref.type),
xref.frm, xref.to, xref.iscode))
Python>
17 Code_Near_Call 0xa143b0 0xa21138 1
17 Code_Near_Call 0xa1bb1b 0xa21138 1
3 Data_Read 0xa143b0 0xa21138 0
3 Data_Read 0xa1bb1b 0xa21138 0
Python>print(idc.generate_disasm_line(0xa143b0, 0))
call    ds:GetProcessHeap
```

The verboseness comes from the `Data_Read` and the `Code_Near` both added to the xrefs. Getting all the addresses and adding them to a set can be useful to slim down on all the addresses.

```
def get_to_xrefs(ea):
    xref_set = set([])
    for xref in idautils.XrefsTo(ea, 1):
        xref_set.add(xref.frm)
    return xref_set

def get_frm_xrefs(ea):
    xref_set = set([])
    for xref in idautils.XrefsFrom(ea, 1):
        xref_set.add(xref.to)
```

```
    return xref_set
```

Example of the slim down functions on out `GetProcessHeap` example.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0xa21138 extrn GetProcessHeap:dword
Python>get_to_xrefs(ea)
set([10568624, 10599195])
Python>[("0x%x" % x) for x in get_to_xrefs(ea)]
['0xa143b0', '0xa1bb1b']
```

Searching

We have already gone over some basic searches by iterating over all known functions or instructions. This is useful but sometimes we need to search for specific bytes such as `0x55 0x8B 0xEC`. This byte pattern is the classic function prologue `push ebp`, `mov ebp, esp`. To search for byte or binary patterns we can use `ida_search.find_binary(start, end, searchstr, radiux, sflag)`. `start` and `end` defines the range we would like to search. `searchstr` is the pattern we are searching for. The `radix` is used when writing processor modules. This topic is outside of the scope of this book. I would recommend reading Chapter 19 of Chris Eagle's The IDA Pro Book. For now, the `radix` field is populated with a value of 16. The `sflag` is the direction or condition. There are several different types of flags. The names and values can be seen below.

```
SEARCH_UP = 0
SEARCH_DOWN = 1
SEARCH_NEXT = 2
SEARCH_CASE = 4
SEARCH_REGEX = 8
SEARCH_NOBRK = 16
SEARCH_NOSHOW = 32
SEARCH_IDENT = 128
SEARCH_BRK = 256
```

Not all these flags are worth going over, but we can touch upon the most used flags.

- `SEARCH_UP` and `SEARCH_DOWN` is used to select the direction we would like our search to follow.
- `SEARCH_NEXT` is used to get the next found object.
- `SEARCH_CASE` is used to specify case sensitivity.
- `SEARCH_NOSHOW` does not show the search progress.

Previous versions of IDA contained a `sflag` of `SEARCH_UNICODE` to search for Unicode strings. This flag is no longer necessary when searching for characters because IDA searches for both ASCII and

Unicode by default. Let us go over a quick walk through on finding the function prologue byte pattern mentioned earlier.

```
Python>pattern = '55 8B EC'
addr = idc.get_inf_attr(INF_MIN_EA)
pattern = '55 8B EC'
addr = idc.get_inf_attr(INF_MIN_EA)
for x in range(0, 5):
    addr = ida_search.find_binary(addr, idc.BADADDR, pattern, 
16,ida_search.SEARCH_DOWN)
    if addr != idc.BADADDR:
        print("0x%x %s" % (addr, idc.generate_disasm_line(addr, 0)))
Python>
0x401000 push    ebp
0x401000 push    ebp
0x401000 push    ebp
0x401000 push    ebp
0x401000 push    ebp
```

In the first line we define our search pattern. The search pattern can be in the format of hexadecimal starting with `0x` as in `0x55 0x8B 0xEC` or as bytes appear in IDA's hex view `55 8B EC`. The format `\x55\x8B\xEC` cannot be used unless we were using `ida_search.find_text(ea, y, x, searchstr, sflag)`. `idc.get_inf_attr(INF_MIN_EA)` is used to get the first address in the executable. We then assign the return of use `ida_search.find_binary(start, end, searchstr, radiux, sflag)` to a variable called `addr`.

When searching it is important to verify that the search did find the pattern. This is tested by comparing `addr` with `idc.BADADDR`. We then print the address and disassembly. Notice how the address did not increment? This is because we did not pass the `SEARCH_NEXT` flag. If this flag is not passed the current address is used to search for the pattern. If the last address contained our byte pattern the search will never increment passed it. Below is the corrected version with the `SEARCH_NEXT` flag before `SEARCH_DOWN` .

```
Python> pattern = '55 8B EC'
addr = idc.get_inf_attr(INF_MIN_EA)
pattern = '55 8B EC'
addr = idc.get_inf_attr(INF_MIN_EA)
for x in range(0, 5):
    addr = ida_search.find_binary(addr, idc.BADADDR, pattern, 16, 
ida_search.SEARCH_NEXT|ida_search.SEARCH_DOWN)
    if addr != idc.BADADDR:
        print("0x%x %s" % (addr, idc.generate_disasm_line(addr, 0))
```

```
Python>
0x401000 push    ebp
0x401040 push    ebp
0x401070 push    ebp
0x4010e0 push    ebp
0x401150 push    ebp
```

Searching for byte patterns is useful but sometimes we might want to search for strings such as `"chrome.dll"`. We could convert the strings to a hex bytes using `[hex(y) for y in bytearray("chrome.dll")]` but this is a little ugly. Also, if the string is Unicode, we would have to account for that encoding. The simplest approach is using `ida_search.find_text(ea, y, x, searchstr, sflag)`. Most of these fields should look familiar because they are the same as `ida_search.find_binary`. `ea` is the start address. `y` is the number of lines at `ea` to search from and `x` is the coordinate in the line. The fields `y` and `x` are typically assigned as `0`. `searchstr` is the pattern to search for and `sflag` defines the direction and types to search for. As an example, we can search for all occurrences of the string `"Accept"`. Any string from the strings window `shift+F12` can be used for this example search .

```
Python>cur_addr = idc.get_inf_attr(INF_MIN_EA)
for x in range(0, 5):
    cur_addr = ida_search.find_text(cur_addr, 0, 0, "Accept",
ida_search.SEARCH_DOWN)
    if addr == idc.BADADDR:
        break
    print("0x%x %s" % (cur_addr, idc.generate_disasm_line(cur_addr, 0)))
    cur_addr = idc.next_head(cur_addr)
Python>
0x40da72 push    offset aAcceptEncoding; "Accept-Encoding:\n"
0x40face push    offset aHttp1_1Accept; " HTTP/1.1\r\nAccept: */* \r\n "
0x40fadf push    offset aAcceptLanguage; "Accept-Language: ru \r\n"
...
0x423c00 db 'Accept',0
0x423c14 db 'Accept-Language',0
0x423c24 db 'Accept-Encoding',0
0x423ca4 db 'Accept-Ranges',0
```

We use `idc.get_inf_attr(INF_MIN_EA)` to get the minimum address and assign that to a variable named `cur_addr`. This is similarly done again for the maximum address by calling `idc.get_inf_attr(INF_MAX_EA)` and assigning the return to a variable named the end. Since we do not know how many occurrences of the string are present, we need to check that the search continues down and is less than the maximum address. We then assign the return of `ida_search.find_text` to the current address. Since we are manually incrementing the address by calling `idc.next_head(ea)` we do not need the `SEARCH_NEXT` flag. The reason why we manually increment the current address to the following line is because a string can occur multiple times on a single line. This can make it tricky to get the address of the next string.

Along with pattern searching previously described there a couple of functions that can be used to find other types. The naming conventions of the find APIs makes it easy to infer its overall functionality. Before we discuss finding the different types, we firstly go over identifying types by their address. There is a subset of APIs that start with `"is"` that can be used to determine an address's type. The APIs return a Boolean value of `True` or `False`.

idc.is_code(f)

Returns `True` if IDA has marked the address as code.

idc.is_data(f)

Returns `True` if IDA has marked the address as data.

idc.is_tail(f)

Returns `True` if IDA has marked the address as tail.

idc.is_unknown(f)

Returns `True` if IDA has marked the address as unknown. This type is used when IDA has not identified if the address is code or data.

idc.is_head(f)

Returns `True` if IDA has marked the address as head.

The `f` is new to us. Rather than passing an address we first need to get the internal flags representation and then pass it to our `idc.is_*` set of functions. To get the internal flags we use `idc.get_full_flags(ea)`. Now that we have a basics on how the function can be used and the different types let's do a quick example.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x10001000 push    ebp
Python>idc.is_code(idc.get_full_flags(ea))
True
```

ida_search.find_code(ea, flag)

It is used to find the next address that is marked as code. This can be useful if we want to find the end of a block of data. If `ea` is an address that is already marked as code it returns the next address. The flag is used as previously described in `ida_search.find_text`.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x4140e8 dd offset dword_4140EC
Python>addr = ida_search.find_code(ea, SEARCH_DOWN|SEARCH_NEXT)
Python>print("0x%x %s" % (addr, idc.generate_disasm_line(addr, 0)))
0x41410c push    ebx
```

As we can see `ea` is the address `0x4140e8` of some data. We assign the return of `ida_search.find_code(ea, SEARCH_DOWN|SEARCH_NEXT)` to `addr`. Then we print `addr` and its disassembly. By calling this single function we skipped 36 bytes of data to get the start of a section marked as code.

ida_search.find_data(ea, flag)

It is used exactly as `ida_search.find_code` except it returns the start of the next address that is marked as a block of data. If we reverse the previous scenario and start from the address of code and search up to find the start of the data.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x41410c push    ebx
Python>addr = ida_search.find_data(ea, SEARCH_UP|SEARCH_NEXT)
Python>print("0x%x %s" % (addr, idc.generate_disasm_line(addr, 0)))
0x4140ec dd 49540E0Eh, 746E6564h, 4570614Dh, 7972746Eh, 8, 1, 4010BCh
```

The only thing that is slightly different than the previous example is the direction of `SEARCH_UP|SEARCH_NEXT` and searching for data.

ida_search.find_unknown(ea, flag)

This function is used to find the address of bytes that IDA did not identify as code or data. The `unknown` type requires further manual analysis either visually or through scripting.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x406a05 jge     short loc_406A3A
Python>addr = ida_search.find_unknown(ea, SEARCH_DOWN)
Python>print("0x%x %s" % (addr, idc.generate_disasm_line(addr, 0))))
0x41b004 db 0DFh ; ?
```

ida_search.find_defined(ea, flag)

It is used to find an address that IDA identified as code or data.

```
0x41b900 db    ? ;
Python>addr = ida_search.find_defined(ea, SEARCH_UP)
Python>print("0x%x %s" % (addr, idc.generate_disasm_line(addr, 0))))
0x41b5f4 dd ?
```

This might not seem of any real value but if we were to print the cross references of `addr` we would see it is being used.

```
Python>for xref in idautils.XrefsTo(addr, 1):
      print("0x%x %s" % (xref.frm, idc.generate_disasm_line(addr, 0))))
Python>
0x4069c3 mov     eax, dword_41B5F4[ecx*4]
```

ida_search.find_imm(ea, flag, value)

Rather than searching for a type we might want to search for a specific value. say for example that we have a feeling that the code calls `rand` to generate a random number, but we can't find the code. If we knew that `rand` uses the value `0x343FD` as a seed, we could search for that number via `ida_search.find_imm(get_inf_attr(INF_MIN_EA), SEARCH_DOWN, 0x343FD )`

```
Python>addr = ida_search.find_imm(get_inf_attr(INF_MIN_EA), SEARCH_DOWN, 0x343FD )
Python>addr
[268453092, 0]
Python>print("0x%x %s %x" % (addr[0], idc.generate_disasm_line(addr[0], 0),
addr[1]))
0x100044e4 imul    eax, 343FDh 0
```

In the first line we pass the minimum address via `get_inf_attr(INF_MIN_EA)`, search down and then search for the value `0x343FD`. Rather than returning an address as shown in the previous Find APIs `ida_search.find_imm` returns a tuple. The first item in the tuple is the address and second is the operand. Like the return of `idc.print_operand` the first operand starts at zero. When we print the address and disassembly, we can see the value is the second operand. If we wanted to search for all uses of an immediate value, we could do the following.

```
Python>addr = idc.get_inf_attr(INF_MIN_EA)
while True:
    addr, operand = ida_search.find_imm(addr, SEARCH_DOWN | SEARCH_NEXT, 4)
    if addr == BADADDR:
        break
    print("0x%x %s Operand %i" % (addr, idc.generate_disasm_line(addr, 0),
operand))
Python>
```

```
0x402434 dd 9, 0FF0Bh, 0Ch, 0FF0Dh, 0Dh, 0FF13h, 13h, 0FF1Bh, 1Bh Operand  0
0x40acee cmp     eax, 7Ah Operand  1
0x40b943 push    7Ah Operand  0
0x424a91 cmp     eax, 7Ah Operand  1
0x424b3d cmp     eax, 7Ah Operand  1
0x425507 cmp     eax, 7Ah Operand  1
```

Most of the code should look familiar but since we are searching for multiple values it uses a while loop and the `SEARCH_DOWN|SEARCH_NEXT` flag.

There are some situations when searching using `ida_search.find_*` can be a little slow. Yara can be used to speed up searches in IDA. Please see chapter *Yara*, for more details on using Yara within IDA to speed up searches.

Selecting Data

We will not always need to search for code or data. In some instances, we already know the location of the code or data, but we want to select it for analysis. In situations like this we might just want to highlight the code and start working with it in IDAPython. To get the boundaries of selected data we can use `idc.read_selection_start()` to get the start and `idc.read_selection_end()` to get the `end`. Let's say we have the below code selected.

```
.text:00408E46                  push    ebp
.text:00408E47                  mov     ebp, esp
.text:00408E49                  mov     al, byte ptr dword_42A508
.text:00408E4E                  sub     esp, 78h
.text:00408E51                  test    al, 10h
.text:00408E53                  jz      short loc_408E78
.text:00408E55                  lea     eax, [ebp+Data]
```

We can use the following code to print out the addresses.

```
Python>start = idc.read_selection_start()
Python>print("0x%x" % start)
0x408e46
Python>end = idc.read_selection_end()
Python>print("0x%x" % end)
0x408e58
```

We assign the return of `idc.read_selection_start()` to start. This is the address of the first selected address. We then use the return of `idc.read_selection_end()` and assign it to end.

One thing to note is that end is not the last selected address but the start of the next address. If we preferred to make only one API call, we could use `idaapi.read_selection()`.

Comments & Renaming

A personal belief of mine is "If I'm not writing, I'm not reversing". Adding comments, renaming functions, and interacting with the assembly is one of the best ways to understand what the code is doing. Over time some of the interaction becomes redundant. In situations like this it useful to automate the process.

Before we go over some examples, we should first discuss the basics of comments and renaming. There are two types of comments. The first one is a regular comment and the second is a repeatable comment. A regular comment appears at address `0x041136B` as the text regular comment. A repeatable comment can be seen at address `0x0411372`, `0x0411386` and `0x0411392`. Only the last comment is a comment that was manually entered. The other comments appear when an instruction references an address (such as a branch condition) that contains a repeatable comment.

```
00411365           mov     [ebp+var_214], eax
0041136B           cmp     [ebp+var_214], 0    ; regular comment
00411372           jnz     short loc_411392  ; repeatable comment
00411374           push    offset sub_4110E0
00411379           call    sub_40D060
0041137E           add     esp, 4
00411381           movzx   edx, al
00411384           test    edx, edx
00411386           jz      short loc_411392 ; repeatable comment
00411388           mov     dword_436B80, 1
00411392
00411392 loc_411392:
00411392
00411392           mov     dword_436B88, 1   ; repeatable comment
0041139C           push    offset sub_4112C0
```

To add comments, we use `idc.set_cmt(ea, comment,0)` and for repeatable comments we use `idc.set_cmt(ea, comment, 1)`. `ea` is the address, `comment` is a string we would like added, 0 specifies the comment is not repeatable and 1 states the comment as repeatable. The below code adds a comment every time an instruction zeroes out a register or value with `XOR`.

```
for func in idautils.Functions():
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    # skip library & thunk functions
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
```

```
    dism_addr = list(idautils.FuncItems(func))
    for ea in dism_addr:
        if idc.print_insn_mnem(ea) == "xor":
            if idc.print_operand(ea, 0) == idc.print_operand(ea, 1):
                comment = "%s = 0" % (idc.print_operand(ea, 0))
                idc.set_cmt(ea, comment, 0)
```

As previously described, we loop through all functions by calling `idautils.Functions()` and loop through all the instructions by calling `list(idautils.FuncItems(func))`. We read the mnemonic using `idc.print_insn_mnem(ea)` and check it is equal to xor. If so, we verify the operands are equal with `idc.print_operand(ea, n)`. If equal, we create a string with the operand and then make add a non-repeatable comment.

```
0040B0F7            xor     al, al                 ; al = 0
0040B0F9            jmp     short loc_40B163
```

To add a repeatable comment, we would replace `idc.set_cmt(ea, comment, 0)` with `idc.set_cmt(ea, comment, 1)`. This might be a little more useful because we would see references to branches that zero out a value and likely return 0. To get a comment we simple use `idc.get_cmt(ea, repeatable)`. `ea` is the address that contains the comment and `repeatable` is a bool of True (1) or False (0). To get the above comments we would use the following code snippet.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x40b0f7 xor     al, al                 ; al = 0
Python>idc.get_cmt(ea, False)
al = 0
```

If the comment was repeatable, we would replace `idc.get_cmt(ea, False)` with `idc.get_cmt(ea, True)`. Instructions are not the only field that can have comments added. Functions can also have comments added. To add a function comment we use `idc.set_func_cmt(ea, cmt, repeatable)` and to get a function comment we call `idc.get_func_cmt(ea, repeatable)`. `ea` can be any address that is within the boundaries of the start and end of the function. `cmt` is the string comment we would like to add and `repeatable` is a Boolean value marking the comment as repeatable or not. This is represented either as 0 or False for the comment not being repeatable or 1 or True for the comment to be repeatable. Having the function comment as repeatable adds a comment whenever the function is cross-referenced, called or viewed in IDA's GUI.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x401040 push    ebp
Python>idc.get_func_name(ea)
sub_401040
```

```
Python>idc.set_func_cmt(ea, "check out later", 1)
True
```

We print the address, disassembly and function name in the first couple of lines. We then use `idc.set_func_cmt(ea, comment, repeatable)` to set a repeatable comment of "check out later". If we look at the start of the function, we will see our comment.

```
00401040 ; check out later
00401040 ; Attributes: bp-based frame
00401040
00401040 sub_401040 proc near
00401040                                             .
00401040 var_4     = dword ptr -4
00401040 arg_0     = dword ptr  8
00401040
00401040           push    ebp
00401041           mov     ebp, esp
00401043           push    ecx
00401044           push    723EB0D5h
```

Since the comment is repeatable, it is displayed whenever the function is viewed. This is a great place to add reminders or notes about a function.

```
00401C07           push    ecx
00401C08           call    sub_401040     ; check out later
00401C0D           add     esp, 4
```

Renaming functions and addresses is a commonly automated task, especially when dealing with position independent code (PIC), packers or wrapper functions. The reason why this is common in PIC or unpacked code is because the import table might not be present in the dump. In the case of wrapper functions the full function simply calls an API.

```
10005B3E sub_10005B3E proc near
10005B3E
10005B3E dwBytes   = dword ptr  8
10005B3E
10005B3E           push    ebp
10005B3F           mov     ebp, esp
10005B41           push    [ebp+dwBytes]         ; dwBytes
10005B44           push    8                     ; dwFlags
```

```
10005B46            push    hHeap                ; hHeap
10005B4C            call    ds:HeapAlloc
10005B52            pop     ebp
10005B53            retn
10005B53 sub_10005B3E endp
```

In the above code the function could be called `"w_HeapAlloc"`. The `w_` is short for wrapper. To rename an address we can use the function `idc.set_name(ea, name, SN_CHECK)`. `ea` is the address and name are the string name such as `"w_HeapAlloc"`. To rename a function `ea` needs to be the first address of the function. To rename the function of our `HeapAlloc` wrapper we would use the following code.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x10005b3e push    ebp
Python>idc.set_name(ea, "w_HeapAlloc", SN_CHECK)
True
```

`ea` is the first address in the function and name is `"w_HeapAlloc"`.

```
10005B3E w_HeapAlloc proc near
10005B3E
10005B3E dwBytes   = dword ptr  8
10005B3E
10005B3E            push    ebp
10005B3F            mov     ebp, esp
10005B41            push    [ebp+dwBytes]        ; dwBytes
10005B44            push    8                    ; dwFlags
10005B46            push    hHeap                ; hHeap
10005B4C            call    ds:HeapAlloc
10005B52            pop     ebp
10005B53            retn
10005B53 w_HeapAlloc endp
```

Above we can see the function has been renamed. To confirm it has been renamed we can use `idc.get_func_name(ea)` to print the new function's name.

```
Python>idc.get_func_name(ea)
w_HeapAlloc
```

To rename an operand we first need to get the address of it. At address `0x04047B0` we have a dword that we would like to rename.

```
.text:004047AD          lea     ecx, [ecx+0]
.text:004047B0          mov     eax, dword_41400C
.text:004047B6          mov     ecx, [edi+4BCh]
```

To get the operand value we can use `idc.get_operand_value(ea, n)`.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x4047b0 mov     eax, dword_41400C
Python>op = idc.get_operand_value(ea, 1)
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x41400c dd 2
Python>idc.set_name(op, "BETA", SN_CHECK)
True
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0x4047b0 mov     eax, BETA[esi]
```

In the first line we print the current working address. We assign the second operand value `dword_41400C` to op by calling `idc.get_operand_value(ea, n)`. We pass the address of the operand to `idc.set_name(ea, name, SN_CHECK)` and then print the newly renamed operand.

Now that we have a good basis of knowledge, we can use what we have learned so far to automate the naming of wrapper functions. Please see the inline comments to get an idea about the logic.

```
import idautils


def rename_wrapper(name, func_addr):
    if idc.set_name(func_addr, name, SN_NOWARN):
        print("Function at 0x%x renamed %s" % (func_addr, idc.get_func_name(func)))
    else:
        print("Rename at 0x%x failed. Function %s is being used." % (func_addr,
name))
    return



def check_for_wrapper(func):
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    # skip library & thunk functions
    if flags & FUNC_LIB or flags & FUNC_THUNK:
```

```
        return
    dism_addr = list(idautils.FuncItems(func))
    # get length of the function
    func_length = len(dism_addr)
    # if over 32 lines of instruction return
    if func_length > 0x20:
        return
    func_call = 0
    instr_cmp = 0
    op = None
    op_addr = None
    op_type = None
    # for each instruction in the function
    for ea in dism_addr:
        m = idc.print_insn_mnem(ea)
        if m == 'call' or m == 'jmp':
            if m == 'jmp':
                temp = idc.get_operand_value(ea, 0)
                # ignore jump conditions within the function boundaries
                if temp in dism_addr:
                    continue
            func_call += 1
            # wrappers should not contain multiple function calls
            if func_call == 2:
                return
            op_addr = idc.get_operand_value(ea, 0)
            op_type = idc.get_operand_type(ea, 0)
        elif m == 'cmp' or m == 'test':
            # wrappers functions should not contain much logic.
            instr_cmp += 1
            if instr_cmp == 3:
                return
        else:
            continue
    # all instructions in the function have been analyzed
    if op_addr == None:
        return
    name = idc.get_name(op_addr, ida_name.GN_VISIBLE)
    # skip mangled function names
```

```
    if "[" in name or "$" in name or "?" in name or "@" in name or name == "":
        return
    name = "w_" + name
    if op_type == 7:
        if idc.get_func_attr(op_addr, FUNCATTR_FLAGS) & FUNC_THUNK:
            rename_wrapper(name, func)
            return
    if op_type == 2 or op_type == 6:
        rename_wrapper(name, func)
        return

for func in idautils.Functions():
    check_for_wrapper(func)
```

Example Output

```
Function at 0xa14040 renamed w_HeapFree
Function at 0xa14060 renamed w_HeapAlloc
Function at 0xa14300 renamed w_HeapReAlloc
Rename at 0xa14330 failed. Function w_HeapAlloc is being used.
Rename at 0xa14360 failed. Function w_HeapFree is being used.
Function at 0xa1b040 renamed w_RtlZeroMemory
```

Most of the code should be familiar. One notable difference is the use of `idc.set_name(ea, name, flag)` from `rename_wrapper`. We use this function because `idc.set_name` throws a warning dialogue if the function name is already in use. By passing a flag value of `SN_NOWARN` or `256` we avoid the dialogue box. We could apply some logic to rename the function to `w_HeapFree_1` but for brevity we will leave that out.

To determine if a function has been renamed, we can use the addresses' flags. The following code is a function that has been renamed.

```
.text:000000014001FF90 func_example    proc near   ; CODE XREF: sub_140020B52+3A↓p
.text:000000014001FF90                 ; sub_140020BEF+3A↓p ...
.text:000000014001FF90
.text:000000014001FF90 var_18          = qword ptr -18h
.text:000000014001FF90 var_10          = dword ptr -10h
.text:000000014001FF90
.text:000000014001FF90                 sub     rsp, 38h
.text:000000014001FF94                 mov     eax, cs:dword_1400268D4
```

```
.text:000000014001FF9A                 mov     r9, cs:DelayLoadFailureHook
```

To retrieve the flags, we call `ida_bytes.get_flags(ea)`. It takes a single argument of the address we would like to retrieve the flags for. The return is the address flags that is then passed to `idc.hasUserName(flags)` to determine if the address has been renamed by the user.

```
Python>here()
0x14001ff90
Python>ida_bytes.get_flags(here())
0x51005600
Python>idc.hasUserName(ida_bytes.get_flags(here()))
True
```

Coloring

Adding a little bit of color to IDA's output is an easy way to speed up the analysis process. Color can be used to visually add context to instructions, blocks, or segments. When skimming large functions, it can be easy to miss a `call` instruction and therefore miss functionality. If we were to color all lines that contain a `call` instruction it would be much easier to quickly identify calls to sub-function. To change the colors displayed in an IDB we use the function `idc.set_color(ea, what, color)`. The first argument `ea` is the address. The second argument is `what`. It is used to designate what it is supposed to be colored. It can be either `CIC_ITEM` for coloring an instruction, `CIC_FUNC` for coloring a function block and `CIC_SEGM` for coloring a segment. The `color` argument takes an integer value of a hex color code. IDA uses the hex color code format of BGR (`0xBBGGRR`) rather than RGB (`0xRRGGBB`). The latter hex color code is more prevalent due to it being used in HTML, CSS or SVG. To color a `call` instruction with the hex color code `0xDFD9F3`, we could use the following code.

```
for func in idautils.Functions():
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    # skip library & thunk functions
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
    dism_addr = list(idautils.FuncItems(func))
    for ea in dism_addr:
        if idc.print_insn_mnem(ea) == "call":
            idc.set_color(ea, CIC_ITEM ,0xDFD9F3)
```

Except for the last line all the code has been previously described. The code loops through all functions and all instructions. If an instruction contains the mnemonic `call` instruction, it will change the color of the address. The last line calls the function `idc.set_color` with the current address as the first argument. Since we are only interested in identifying a single instruction we define the `what` argument (second) as `CIC_ITEM`. The last argument is the BGR hex encoded color code. If we were to view an IBD that had our color call script executed the below lines `0x0401469` and `0x0401473` would have had their color changed.

```
.text:00401468          push    ecx                     ; int
.text:00401469          call    __setmode  ; color coded
.text:0040146E          lea     edx, [esp+40B8h+var_405C]
.text:00401472          push    edx
.text:00401473          call    constants  ; color coded
.text:00401478          push    esi                     ; FILE *
```

To retrieve the hex color code for an address we use the function `idc.get_color(ea, what)`. The first argument `ea` is the address. The second argument `what` is the type of item we would like to get the color for. It uses the same items as previously described (`CIC_ITEM`, `CIC_FUNC` & `CIC_SEGM`). The following code gets the hex color code for the instruction, function and segment at address `0x0401469.`

```
Python>"0x%x" % (idc.get_color(0x0401469, CIC_ITEM))
0xdfd9f3
Python>"0x%x" % (idc.get_color(0x0401469, CIC_FUNC))
0xffffffff
Python>"0x%x" % (idc.get_color(0x0401469, CIC_SEGM))
0xffffffff
```

The hex color code `0xffffffff` is the default color code used by IDA. If you are interested in changing the color themes of IDA, I would recommend checking out the *IDASkins*[11] project.

Accessing Raw Data

Being able to access raw data is essential when reverse engineering. Raw data is the binary representation of the code or data. We can see the raw data or bytes of the instructions on the left side following the address.

```
00A14380 8B 0D 0C 6D A2 00          mov     ecx, hHeap
00A14386 50                         push    eax
00A14387 6A 08                      push    8
00A14389 51                         push    ecx
00A1438A FF 15 30 11 A2 00          call    ds:HeapAlloc
00A14390 C3                         retn
```

To access the data, we first need to decide on the unit size. The naming convention of the APIs used to access data is the unit size. To access a byte, we would call `idc.get_wide_byte(ea)` or to access a word we would call `idc.get_wide_word(ea)`, etc.

- idc.get_wide_byte(ea)
- idc.get_wide_word(ea)

[11] https://github.com/zyantific/IDASkins

- idc.get_wide_dword(ea)
- idc.get_qword(ea)
- idc.GetFloat(ea)
- idc.GetDouble(ea)

If the cursor was at `0x0A14380` in the assembly from above, we would have the following output.

```
Python>print("0x%x %s" % (ea, idc.generate_disasm_line(ea, 0)))
0xa14380 mov     ecx, hHeap
Python>"0x%x" % idc.get_wide_byte(ea)
0x8b
Python>"0x%x" % idc.get_wide_word(ea)
0xd8b
Python>"0x%x" % idc.get_wide_dword(ea)
0x6d0c0d8b
Python>"0x%x" % idc.get_qword(ea)
0x6a5000a26d0c0d8bL
Python>idc.GetFloat(ea) # Example not a float value
2.70901711372e+27
Python>idc.GetDouble(ea)
1.25430839165e+204
```

When writing decoders, it is not always useful to get a single byte or read a dword but to read a block of raw data. To read a specified size of bytes at an address we can use `idc.get_bytes(ea, size, use_dbg=False)`. The last argument is optional and is only needed if we wanted the debuggers memory.

```
Python>for byte in idc.get_bytes(ea, 6):
    print("0x%X" % byte),
0x8B 0xD 0xC 0x6D 0xA2 0x0
```

Patching

Sometimes when reversing malware, the sample contains strings that are encoded. This is done to slow down the analysis process and to thwart using a strings viewer to recover indicators. In situations like this patching the IDB is useful. We could rename the address but renaming is limited. This is due to the naming convention restrictions. To patch an address with a value we can use the following functions.

- idc.patch_byte(ea, value)
- idc.patch_word(ea, value)
- idc.patch_dword(ea, value)

`ea` is the address and `value` are the integer value that we would like to patch the IDB with. The size of the value needs to match the size specified by the function name we choose. One example that we found the following encoded strings.

```
.data:1001ED3C aGcquEUdg_bUfuD db 'gcqu^E]~UDG_B[uFU^DC',0
.data:1001ED51                 align 8
.data:1001ED58 aGcqs_cuufuD    db 'gcqs\_CUuFU^D',0
.data:1001ED66                 align 4
.data:1001ED68 aWud@uubQU      db 'WUD@UUB^Q]U',0
.data:1001ED74                 align 8
```

During our analysis we were able to identify the decoder function.

```
100012A0          push    esi
100012A1          mov     esi, [esp+4+_size]
100012A5          xor     eax, eax
100012A7          test    esi, esi
100012A9          jle     short _ret
100012AB          mov     dl, [esp+4+_key]      ; assign key
100012AF          mov     ecx, [esp+4+_string]
100012B3          push    ebx
100012B4
100012B4 _loop:                                 ;
100012B4          mov     bl, [eax+ecx]
100012B7          xor     bl, dl                ; data ^ key
100012B9          mov     [eax+ecx], bl         ; save off byte
100012BC          inc     eax                   ; index/count
100012BD          cmp     eax, esi
100012BF          jl      short _loop
100012C1          pop     ebx
100012C2
100012C2 _ret:                                  ;
100012C2          pop     esi
100012C3          retn
```

The function is a standard XOR decoder function with arguments of size, key and a decoded buffer.

```
Python>start = idc.read_selection_start()
Python>end = idc.read_selection_end()
Python>print hex(start)
```

```
0x1001ed3c
Python>print hex(end)
0x1001ed50
Python>def xor(size, key, buff):
    for index in range(0, size):
        cur_addr = buff + index
        temp = idc.get_wide_byte(cur_addr ) ^ key
        idc.patch_byte(cur_addr, temp)
Python>
Python>xor(end - start, 0x30, start)
Python>idc.get_strlit_contents(start)
WSAEnumNetworkEvents
```

We select the highlighted data address start and end using `idc.read_selection_start()` and `idc.read_selection_end()`. Then we have a function that reads the byte by calling `idc.get_wide_byte(ea)`, XOR the byte with key passed to the function and then patch the byte by calling `idc.patch_byte(ea, value)`.

Input and Output

Importing and exporting files into IDAPython can be useful when we do not know the file path or when we do not know where the user wants to save their data. To import or save a file by name we use `ida_kernwin.ask_file(forsave, mask, prompt)`. `forsave` can be a value of `0` if we want to open a dialog box or `1` is we want to open the save dialog box. `mask` is the file extension or pattern. If we want to open only `.dll` files we would use a mask of `"*.dll"` and `prompt` is the title of the window. A good example of input and output and selecting data is the following `IO_DATA` class.

```
import sys
import idaapi

class IO_DATA():
    def __init__(self):
        self.start = idc.read_selection_start()
        self.end = idc.read_selection_end()
        self.buffer = ''
        self.ogLen = None
        self.status = True
        self.run()

    def checkBounds(self):
```

```
        if self.start is BADADDR or self.end is BADADDR:
            self.status = False

    def getData(self):
        """get data between start and end put them into object.buffer"""
        self.ogLen = self.end - self.start
        self.buffer = b''
        try:
            self.buffer = idc.get_bytes(self.start, self.ogLen)
        except:
            self.status = False
        return

    def run(self):
        """basically main"""
        self.checkBounds()
        if self.status == False:
            sys.stdout.write('ERROR: Please select valid data\n')
            return
        self.getData()



    def patch(self, temp=None):
        """patch idb with data in object.buffer"""
        if temp != None:
            self.buffer = temp
            for index, byte in enumerate(self.buffer):
                idc.patch_byte(self.start + index, ord(byte))



    def importb(self):
        '''import file to save to buffer'''
        fileName = ida_kernwin.ask_file(0, "*.*", 'Import File')
        try:
            self.buffer = open(fileName, 'rb').read()
        except:
            sys.stdout.write('ERROR: Cannot access file')
```

```
    def export(self):
        '''save the selected buffer to a file'''
        exportFile = ida_kernwin.ask_file(1, "*.*", 'Export Buffer')
        f = open(exportFile, 'wb')
        f.write(self.buffer)
        f.close()



    def stats(self):
        print("start: 0x%x" % self.start)
        print("end:   0x%x" % self.end)
        print("len:   0x%x" % len(self.buffer))
```

With this class data can be selected saved to a buffer and then stored to a file. This is useful for encoded or encrypted data in an IDB. We can use `IO_DATA` to select the data decode the buffer in Python and then patch the IDB. Example of how to use the `IO_DATA` class.

```
Python>f = IO_DATA()
Python>f.stats()
start: 0x401528
end:   0x401549
len:   0x21
```

Rather than explaining each line of the code it would be useful for the reader to go over the functions one by one and see how they work. The below bullet points explain each variable and what the functions does. `obj` is whatever variable we assign the class. `f` is the `obj` in `f = IO_DATA()`.

- obj.start
  - contains the address of the start of the selected offset
- . obj.end
  - contains the address of the end of the selected offset.
- obj.buffer
  - contains the binary data.
- obj.ogLen
  - contains the size of the buffer.
- obj.getData()
  - copies the binary data between `obj.start` and `obj.end` to `obj.buffer`
- obj.run()
  - the selected data is copied to the buffer in a binary format
- obj.patch()
  - patch the IDB at `obj.start` with the data in the `obj.buffer`.

- obj.patch(d)
    - patch the IDB at `obj.start` with the argument data.
- obj.importb()
    - opens a file and saves the data in `obj.buffer`.
- obj.export()
    - exports the data in `obj.buffer` to a save as file.
- obj.stats()
    - print hex of `obj.start`, `obj.end` and `obj.buffer` length.

PyQt

Most of the interaction with IDAPython documented in this book is through the command line. In some instances, it might be useful to interact with our code using a graphical user interface, commonly referred to as a Form in IDAPython's documentation. IDA's graphical user interface is written in the cross-platform Qt GUI framework. To interact with this framework, we can use the Python bindings for Qt called PyQt[12]. An in-depth overview of PyQt is outside the scope of this book. What is provided in this chapter is a simple skeleton snippet that can be easily modified and built upon for writing forms. The code creates two widgets, the first widget creates a table and the second widget is a button. When the button is clicked, the current address and the mnemonic are added to a row in the table. If the row is clicked, IDA jumps to the address in the row in the disassembly view. Think of this code as simple address book marker. The below figure is the form after adding three addresses to the form by clicking the "Add Address" button and then double clicking the first row.

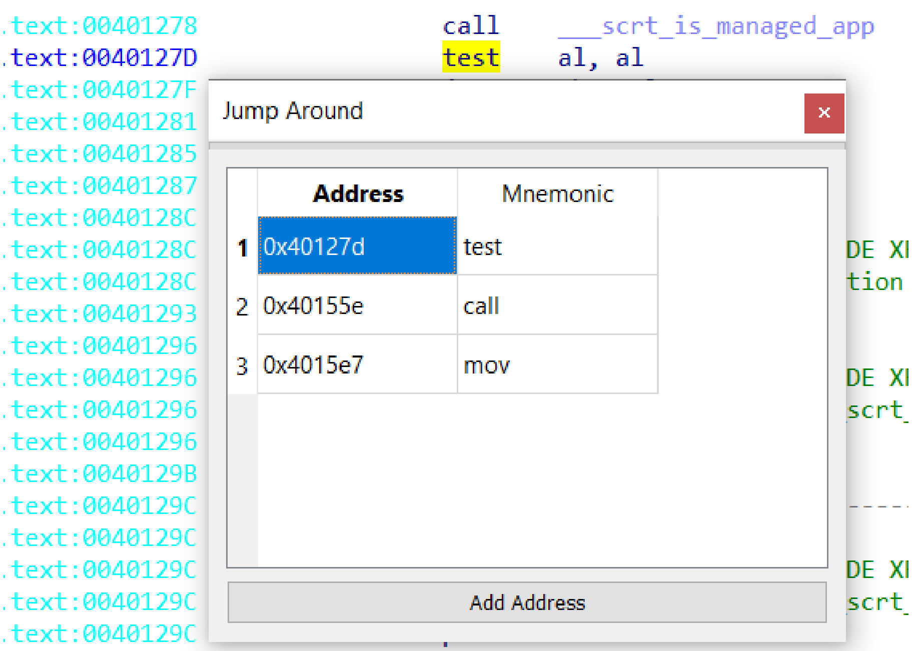


Figure Jump Around Example Form

Not all the APIs in the following code are going to be covered. The reason for this brevity is due to the descriptive nature of PyQt's API names. For example, the function `setColumnCount` sets the

[12] https://riverbankcomputing.com/software/pyqt/intro

number of columns. If an API does not make sense, please search for the API by name. Qt and PyQt are extremely well documented. Once we understand the basics of the below code it is easy to hack together something. The key concept to understand about PyQt when reviewing the below code is to understand that PyQt is an object-oriented framework.

```
from idaapi import PluginForm
from PyQt5 import QtCore, QtGui, QtWidgets

class MyPluginFormClass(PluginForm):
    def OnCreate(self, form):
        # Get parent widget
        self.parent = self.FormToPyQtWidget(form) # IDAPython
        self.PopulateForm()

    def PopulateForm(self):
        # Create layout
        layout = QtWidgets.QVBoxLayout()
        # Create Table Widget
        self.example_row = QtWidgets.QTableWidget()
        column_names = ["Address", "Mnemonic"]
        self.example_row.setColumnCount(len(column_names))
        self.example_row.setRowCount(0)
        self.example_row.setHorizontalHeaderLabels(column_names)
        self.example_row.doubleClicked.connect(self.JumpSearch)
        layout.addWidget(self.example_row)
        # Create Button
        self.addbtn = QtWidgets.QPushButton("Add Address")
        self.addbtn.clicked.connect(self.AddAddress)
        layout.addWidget(self.addbtn)
        # make our created layout the dialogs layout
        self.parent.setLayout(layout)

    def AddAddress(self):
        ea = here() # IDAPython
        index = self.example_row.rowCount()
        self.example_row.setRowCount(index + 1)
        h = "0x%x" % ea
        item = QtWidgets.QTableWidgetItem(h)
        item.setFlags(item.flags() ^ QtCore.Qt.ItemIsEditable)
```

```
        self.example_row.setItem(index, 0, item)
        self.example_row.setItem(index, 1,
QtWidgets.QTableWidgetItem(idc.print_insn_mnem(ea))) # IDAPython
        self.example_row.update()

    def JumpSearch(self, item):
        tt = self.example_row.item(item.row(), 0)
        ea = int(tt.text(), 16)
        idaapi.jumpto(ea) # IDAPython

plg = MyPluginFormClass()
plg.Show("Jump Around")
```

The first two lines contain import the needed modules. To create a form, a class is created inheriting from `PluginForm` within `idaapi`. Within the `MyPluginFormClass` class is a method called `OnCreate`. This method is called when the plugin form is created. `OnClose` is the opposite and is called when the plugin is closed. The fuction `self.FormToPyQtWidget(form)` creates the necessary parent instance that is used to populate our widgets, which is stored in `self.parent`. The method `PopulateForm(self)` is where all the design and creation of the widgets happens. The core three steps that are important in this method is creating an instance for the layout (`layout = QtWidgets.QVBoxLayout()`), creating (`self.example_row = QtWidgets.QTableWidget()`) and adding the widgets (`layout.addWidget(self.example_row)`), and then setting the layout (`self.parent.setLayout(layout)`). The rest of the code is modifying the widgets or adding actions to the widgets. Once such action is calling the method `self.JumpSearch` if a row is double clicked. If the user double clicks the row, it reads the first row and then calls `idaapi.jumpto(ea)` to redirect the Disassembly view to the address. When the layout has been set, the method `Show(str)` within the Form instance is used to display the Form. `Show(str)` takes a string argument that is the Form's title (e.g. "Jump Around").

Batch File Generation`

Sometimes it can be useful to create IDBs or ASMs for all the files in a directory. This can help save time when analyzing a set of samples that are part of the same family of malware. It is much easier to do batch file generation than doing it manually on a large set. To do batch analysis we need to pass the `-B` argument to the text `idat.exe`. The below code can be copied to the directory that contains all the files we would like to generate files for.

```
import os
import subprocess
import glob

paths = glob.glob("*")
ida_path = os.path.join(os.environ['PROGRAMFILES'], "IDA Pro 7.5", "idat.exe")
```

```
for file_path in paths:
    if file_path.endswith(".py"):
        continue
    subprocess.call([ida_path, "-B", file_path])
```

We use `glob.glob("*")` to get a list of all files in the directory. The argument can be modified if we wanted to only select a certain regular expression pattern or file type. If we wanted to only get files with a `.exe` extension we would use `glob.glob("*.exe")`. `os.path.join(os.environ['PROGRAMFILES'], "IDA", "idat.exe")` is used to the get the path to idat.exe. Some versions of IDA have a folder name with the version number present. If this is the case the argument `"IDA"` needs to be modified to the folder name. Also, the whole command might have to be modified if we choose to use a non-standard install location for IDA. For now, let's assume the install path for IDA is `C:\Program Files\IDA`. After we found the path we loop through all the files in the directory that do not contain a .py extension and then pass them to IDA. For an individual file it would look like `C:\Program Files\IDA\idat.exe -B bad_file.exe`. Once ran it would generate an ASM and IDB for the file. All files will be written in the working directory. An example output can be seen below.

```
C:\injected>dir

0?/**/____  09:30 AM    <DIR>          .
0?/**/____  09:30 AM    <DIR>          ..
0?/**/____  10:48 AM           167,936 bad_file.exe
0?/**/____  09:29 AM               270 batch_analysis.py
0?/**/____  06:55 PM           104,889 injected.dll

C:\injected>python batch_analysis.py

Thank you for using IDA. Have a nice day!

C:\injected>dir

0?/**/____  09:30 AM    <DIR>          .
0?/**/____  09:30 AM    <DIR>          ..
0?/**/____  09:30 AM           506,142 bad_file.asm
0?/**/____  10:48 AM           167,936 bad_file.exe
0?/**/____  09:30 AM         1,884,601 bad_file.idb
0?/**/____  09:29 AM               270 batch_analysis.py
0?/**/____  09:30 AM           682,602 injected.asm
0?/**/____  06:55 PM           104,889 injected.dll
```

```
0?/**/____  09:30 AM         1,384,765 injected.idb
```

`bad_file.asm`, `bad_file.idb`, `injected.asm` and `injected.idb` were generated files.

Executing Scripts

IDAPython scripts can be executed from the command line. We can use the following code to count each instruction in the IDB and then write it to a file named `instru_count.txt`.

```
import idc
import idaapi
import idautils

idaapi.auto_wait()

count = 0
for func in idautils.Functions():
    # Ignore Library Code
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    if flags & FUNC_LIB:
        continue
    for instru in idautils.FuncItems(func):
        count += 1

f = open("instru_count.txt", 'w')
print_me = "Instruction Count is %d" % (count)
f.write(print_me)
f.close()

idc.qexit(0)
```

From a command line perspective, the two most important functions are `idaapi.auto_wait()` and `idc.qexit(0)`. When IDA opens a file, it is important to wait for the analysis to complete. This allows IDA to populate all functions, structures, or other values that are based on IDA's analysis engine. To wait for the analysis to complete we call `idaapi.auto_wait()`. It will wait/pause until IDA is completed with its analysis. Once the analysis is completed it returns control back to the script. It is important to execute this at the beginning of the script before we call any IDAPython functions that rely on the analysis to be completed. Once our script has executed, we need to call `idc.qexit(0)`. This stops execution of our script, close out the database and return to the caller of the script. If not our IDB would not be closed properly.

If we wanted to execute the IDAPython to count all lines, in an IDB we would execute the following command line.

```
"C:\Program Files\IDA Pro 7.5\ida.exe" -Scount.py example.idb
```

`-S` signals for IDA to run a script on the IDB once it has opened. In the working directory we would see a file named `instru_count.txt` that contained a count of all instructions. If we wanted to execute our script on an executable, we would need IDA to run in Autonomous mode by passing `-A`.

```
"C:\Program Files\IDA Pro 7.5\ida.exe" -A -Scount.py example.exe
```

Yara

Yara[13] is a rule-based pattern matching software and library that can be used to search for files. It was written and maintained by Victor M. Alvarez. Yara's rules are defined using patterns based off strings (“`foo`”), bytes (`{66 6f 6f}`), file sizes (`filesize < 37`) or other conditional attributes of the file. Due to its powerful and flexible rules, Yara is rightfully referred to as the “pattern matching swiss knife for malware researchers”. From an IDAPython viewpoint, Yara is an excellent library to add to your toolkit for a couple of reasons. For starters, Yara is substantially faster than IDAPython's search, its rules can be used for automating the analysis process and there are plenty of publicly available Yara signatures. One of my favorite search examples for automating the analysis process is searching for constants used by cryptographic functions. By searching for byte patterns, we can cross-reference the match and infer that the function referencing the bytes is related to a cryptographic algorithm. For example, searching for the constant `0x67452301`can be used to find functions related to the hashing algorithms `MD4`, `MD5` and `SHA1`.

The first step in the process of using Yara is to create the rule. Yara rules follow a simple syntax that is like the C language. A rule consists of its name, match pattern (aka strings definition in the Yara docs) and condition. The below text is a simple Yara rule. It is not a practical Yara rule but it is useful for demonstrating Yara's rule syntax.

```
/*
      Example Yara Rule
*/
rule pe_md5_constant
{
    strings:
      $pe_header = "MZ"
      $hex_constant = { 01 23 45 67 } // byte pattern
    condition:
```

[13] https://github.com/VirusTotal/yara

```
    $pe_header at 0 and $hex_constant
}
```

The first couple of lines is a multiline comment. As with C and other languages, the comment starts with `/*` and ends with `*/`. Yara rules have a similar syntax as structures in C.A Yara rule starts with the keyword `rule` followed by the name (aka rule identifier). After the rule is an open curly bracket `{`. Following the opening curly bracket is the string definition which starts with the keyword `strings` followed by a colon `:`. The strings definition is used to define the rule that the Yara matches on. Each string has an identifier that starts with a `$` character followed by characters and digits that make up the string definition name. The string definition can be characters (such as `MZ`) or hex strings (such as `{ 01 23 45 67 }`). After the string definition is the condition that Yara matches on. The conditions start with the keyword `condition` followed by a colon `:`. In the example Yara rule above, the condition that matches is of if the string definitions `$pe_header` is located at offset 0 and the file contains the byte pattern defined in `$hex_constant` then Yara has a match. Since no offset was defined for `$hex_constant` than the byte pattern only need to be present anywhere in the file to have a match. Yara supports a wide range of keywords that can be used to define from wide characters, entry point, size and other conditions. It is recommended to read *Writing Yara Rules* within Yara's documentation to learn about all the different keywords, options they support and the different ways a file can be scanned or matched.

The python interface[14] for Yara can be easily installed using `pip` by executing the command `pip install yara-python`. The following steps are needed to scan a file with Yara in Python

1. Yara needs to be imported
    - `import yara`.
2. Yara needs to compile the Yara rule using `yara.compile`
    - `rules = yara.compile(source=signature)`
3. Open a file or have data in a buffer for Yara to match against
    - `data = open(scan_me, "rb").read()`
4. Scan the file using the compiled Yara rule using `yara.match`
    - `matches = rules.match(data=self.mem_results)`
5. Print match(es) or apply logic based off the matches

This is of course is a simplified description of the steps that are needed. Yara contains several methods and configuration that can be used for more advanced scanning options. Examples of these functionality's are function callbacks, scanning running processes and time outs for larger files. Please see Yara's documentation for a complete list of these methods and configurations. In the context of using Yara within IDA, the same steps are needed to scan binary data within the IDB. Except one additional step is needed to convert the Yara match file offset to an executable virtual address, for which is how IDA references addresses. If a Portable Executable file is being scanned with Yara and it matches the pattern at file offset `0x1000` this could be represented as the virtual address `0x0401000` in IDA. The following code is a class that reads the binary data from an IDB and then scans the data using Yara.

[14] https://github.com/VirusTotal/yara-python

```
import yara
import idautils

SEARCH_CASE = 4
SEARCH_REGEX = 8
SEARCH_NOBRK = 16
SEARCH_NOSHOW = 32
SEARCH_UNICODE = 64
SEARCH_IDENT = 128
SEARCH_BRK = 256

class YaraIDASearch:
    def __init__(self):
        self.mem_results = ""
        self.mem_offsets = []
        if not self.mem_results:
            self._get_memory()

    def _get_memory(self):
        print("Status: Loading memory for Yara.")
        result = b""
        segments_starts = [ea for ea in idautils.Segments()]
        offsets = []
        start_len = 0
        for start in segments_starts:
            end = idc.get_segm_end(start)
            result += idc.get_bytes(start, end - start)
            offsets.append((start, start_len, len(result)))
            start_len = len(result)
        print("Status: Memory has been loaded.")
        self.mem_results = result
        self.mem_offsets = offsets

    def _to_virtual_address(self, offset, segments):
        va_offset = 0
        for seg in segments:
            if seg[1] <= offset < seg[2]:
                va_offset = seg[0] + (offset - seg[1])
        return va_offset
```

```
def _init_sig(self, sig_type, pattern, sflag):
    if SEARCH_REGEX & sflag:
        signature = "/%s/" % pattern
        if SEARCH_CASE & sflag:
            # ida is not case sensitive by default but yara is
            pass
        else:
            signature += " nocase"
        if SEARCH_UNICODE & sflag:
            signature += " wide"
    elif sig_type == "binary":
        signature = "{ %s }" % pattern
    elif sig_type == "text" and (SEARCH_REGEX & sflag) == False:
        signature = '"%s"' % pattern
        if SEARCH_CASE & sflag:
            pass
        else:
            signature += " nocase"
        signature += " wide ascii"
    yara_rule = "rule foo : bar { strings: $a = %s condition: $a }" % signature
    return yara_rule

def _compile_rule(self, signature):
    try:
        rules = yara.compile(source=signature)
    except Exception as e:
        print("ERROR: Cannot compile Yara rule %s" % e)
        return False, None
    return True, rules

def _search(self, signature):
    status, rules = self._compile_rule(signature)
    if not status:
        return False, None
    values = []
    matches = rules.match(data=self.mem_results)
    if not matches:
        return False, None
```

```
        for rule_match in matches:
            for match in rule_match.strings:
                match_offset = match[0]
                values.append(self._to_virtual_address(match_offset,
self.mem_offsets))
        return values

    def find_binary(self, bin_str, sflag=0):
        yara_sig = self._init_sig("binary", bin_str, sflag)
        offset_matches = self._search(yara_sig)
        return offset_matches

    def find_text(self, q_str, sflag=0):
        yara_sig = self._init_sig("text", q_str, sflag)
        offset_matches = self._search(yara_sig)
        return offset_matches

    def find_sig(self, yara_rule):
        offset_matches = self._search(yara_rule)
        return offset_matches

    def reload_scan_memory(self):
        self._get_memory()
```

All the APIs in the previous code have been previously covered. The function `_to_virtual_address` was created by Daniel Plohmann (see the section *What's Next*) and can be used to convert the Yara file offset match to an IDA address within the correct address. The following is an example of creating an instance of `YaraIDASearch()` scanning an IDB with a Yara signature and returning the offset the rule matches on. It should be noticed that this rule has been modified from the previous rule. IDA does not always load the Portable Executable's MZ header[15] as a segment.

```
Python>ys = YaraIDASearch()
Status: Loading memory for Yara.
Status: Memory has been loaded.
Python>example_rule = """rule md5_constant
{
    strings:
           $hex_constant = { 01 23 45 67 } // byte pattern
```

[15] https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format#ms-dos-stub-image-only

```
    condition:
      $hex_constant
}"""
Python>
Python>ys.find_sig(example_rule)
[4199976L]
```

The first line creates a `YaraIDASearch` instance and assigns it to `ys`. The Yara rule is saved as a string and assigned to the variable `example_rule`. The rule is passed as an argument to the method `ys.find_sig(yara_rule)`. The `search` method returns a list of all the offset that the Yara rule matched on. If we wanted to search for a binary pattern, we could use `ys.find_binary(bytes)`. A search of `ys.find_binary(01 23 45 67)` would return the same results as the custom Yara rule. `YaraIDASearch` also support searching for strings using `ys.find_text(string)`.

Unicorn Engine

The Unicorn Engine[16] is a lightweight multi-platform, multi-architecture CPU emulator Framework built on top of a modified version of Qemu. Unicorn is written in C but contains bindings for many languages, including Python. Unicorn is a powerful tool that can aid in the reverse engineering process because it essentially allows code to be emulated in a configurable, controlled, and to a specific state. The later adjective *specific* is where the power of the Unicorn Engine comes into play. To execute code and be told the specific output, without truly understanding the assembly, can save time and help with automating the analysis process. For example, using Unicorn to execute hundreds of inline string decryption routines with varying bit-shifting algorithm and/or XOR keys and then writing the decrypted key as a string as a comment is just one such usage for it.

To get a good understanding of using the Unicorn Engine it is useful to go over some of the core concepts and how to implement them using the Unicorn APIs.

Initialize Unicorn Instance

To initialize the Unicorn class, the API `Uc(UC_ARCH, UC_MODE)` is used. `Uc` defines the specifics of how the code should be emulated. For example, should the binary data be executed as MIPS-32 or as X86-64. The first argument is the hardware architecture type. The second argument is the hardware mode type and/or endianness. The following are the current supported architecture types.

- `UC_ARCH_ARM`
  - ARM architecture (including Thumb, Thumb-2)
- `UC_ARCH_ARM64`
  - ARM-64, also called AArch64
- `UC_ARCH_MIPS`
  - Mips architecture
- `UC_ARCH_X86`

[16] https://www.unicorn-engine.org/

- X86 architecture (including x86 & x86-64)
- `UC_ARCH_PPC`
  - PowerPC architecture (currently unsupported)
- `UC_ARCH_SPARC`
  - Sparc architecture
- `UC_ARCH_M68K`
  - M68K architecture

The following is the available hardware types. The comments are from `unicorn.h`.

Endianness

- `UC_MODE_LITTLE_ENDIAN`
  - little-endian mode (default mode)
- `UC_MODE_BIG_ENDIAN`
  - big-endian mode

Arm

- `UC_MODE_ARM`
  - ARM mode
- `UC_MODE_THUMB`
  - THUMB mode (including Thumb-2)

Mips

- `UC_MODE_MIPS32`
  - Mips32 ISA
- `UC_MODE_MIPS64`
  - Mips64 ISA

x86 / x64

- `UC_MODE_16`
  - 16-bit mode
- `UC_MODE_32`
  - 32-bit mode
  -
- `UC_MODE_64`
  - 64-bit mode

sparc

- `UC_MODE_SPARC32`
  - 32-bit mode
- `UC_MODE_SPARC64`
  - 64-bit mode

There are many different combinations of architecture and hardware types. The Unicorn Engine Python bindings directory contains several example scripts[17]. All the examples have the pattern sample`_.*.py`.

Read and Write Memory

Before memory can read to or write to, the memory needs to be mapped. To map memory the APIs `uc.mem_map(address, size, perms=uc.UC_PROT_ALL)` and `uc.mem_map_ptr(address, size, perms, ptr)` are used. The following memory protections are available.

- `UC_PROT_NONE`
- `UC_PROT_READ`
- `UC_PROT_WRITE`
- `UC_PROT_EXEC`
- `UC_PROT_ALL`

To protect a range of memory the API `uc.mem_protect(address, size, perms=uc.UC_PROT_ALL)` is used. To unmap memory the API `uc.mem_unmap(address, size)` is used. Once the memory is mapped it can be written to by calling `uc.mem_write(address, data)`. To read from the allocated memory `uc.mem_read(address, size)` is used.

Read and Write Registers

Registers can be read by calling `uc.reg_read(reg_id, opt=None)`. The `reg_id` is defined in the appropriate architecture constant Python file in the Python bindings directory[18].

- `ARM-64` in `arm64_const.py`
- `ARM` in `arm_const.py`
- `M68K` in `m68k_const.py`
- `MIPS` in `mips_const.py`
- `SPARC` in `sparc_const.py`
- `X86` in `x86_const.py`

To reference the constants, they must be first imported. The constants are imported by calling the `from unicorn.x86_const import *` for x86. To write the contents of a register `uc.reg_write(reg_id, value)` is used.

Start and Stop Emulation

To start the Unicorn Engine emulating the API `uc.emu_start(begin, until, timeout=0, count=0)` is called. The first function start is the first address that is emulated. The second argument until is the address (or above) that the Unicorn Engine stops emulating at. The argument `timeout=` is

[17] https://github.com/unicorn-engine/unicorn/tree/master/bindings/python
[18] https://github.com/unicorn-engine/unicorn/tree/master/bindings/python/unicorn

used to define the number of milliseconds that the Unicorn Engine executes until its times out. `UC_SECOND_SCALE * n` can be used to wait `n` number of seconds. The last argument `count=` can be used to define the number of instructions that are executed before the Unicorn Engine stops executing. If `count=` is zero or less than counting by the Unicorn Engine is disabled. To stop emulating the API `uc.emu_stop()` is used.

Memory and Hook Management with User-Defined Callbacks

The Unicorn Engine supports a wide arrange of hooks. The following describes a subset of those hooks. The hooks are inserted before the call to start the emulation. To add a hook the API `uc.hook_add(UC_HOOK_*, callback, user_data, begin, end, ...)`. The first two arguments are mandatory. The last three are optional and are usually populated with default values of `None, 1, 0, UC_INS`. To delete a hook the API `emu.hook_del(hook)` is used. To delete a hook, it must be assigned to a variable. For example, the following snippet is how to delete a hook.

```
i = emu.hook_add(UC_HOOK_CODE, hook_code, None)
emu.hook_del(i)
```

Hooks and their corresponding callbacks allow for instrumentation of the emulated code. These callbacks is where we can apply logic for doing analysis, modify the code or simply print out values. These callbacks are extremely useful when debugging errors or ensuring the correct initialization values. Some of the below examples are from Unicorn Engine's sample repo[19].

UC_HOOK_INTR

`UC_HOOK_INTR` is used to hook all interrupt and syscall events. The first argument to the callback `hook_intr` is the Unicorn instance. The instance can be used to call Unicorn API's previously described. The second argument `intno` is interrupt number. The third argument `user_data` is a variable that can be passed from the hook to the callback. The following example prints the interrupt number (if not equal to 0x80) and stops the emulation by calling `uc.emu_stop()`.

```
def hook_intr(uc, intno, user_data):
    # only handle Linux syscall
    if intno != 0x80:
        print("got interrupt %x ???" %intno);
        uc.emu_stop()
        return
uc.hook_add(UC_HOOK_INTR, hook_intr)
```

UC_HOOK_INSN

`UC_HOOK_INSN` adds a hook when the x86 instructions `IN`, `OUT` or `SYSCALL` are executed. The following snippet adds a `UC_HOOK_INSN` and calls the callback function `hook_syscall` whenever a `UC_X86_INS_SYSCALL` is executed. The callback reads the RAX register, if RAX is equal to 0x100, it is patched with 0x200 and the Unicorn Engine continues emulating the code.

```
def hook_syscall(uc, user_data):
```

[19] https://github.com/unicorn-engine/unicorn/blob/master/bindings/python/sample_x86.py

```
            rax = uc.reg_read(UC_X86_REG_RAX)
            if rax == 0x100:
                uc.reg_write(UC_X86_REG_RAX, 0x200)
uc.hook_add(UC_HOOK_INSN, hook_syscall, None,1, 0, UC_X86_INS_SYSCALL)
```

UC_HOOK_CODE

`UC_HOOK_CODE` can hook a range of code. The hook is called before every instruction is executed. The callback `hook_code` contains four arguments. The following snippet implements the `UC_HOOK_CODE` hook and prints the address and size being emulated. The first argument `uc` is the Unicorn instance, `address` is the address of the code to be executed, `size` is the size of emulated instruction and `user_data` has been previously covered.

```
def hook_code(uc, address, size, user_data):
    print("Tracing instruction at 0x%x, instruction size = 0x%x" %(address, size))
uc.hook_add(UC_HOOK_CODE, hook_code)
```

UC_HOOK_BLOCK

UC_HOOK_BLOCK is a hook that can implement a callback for tracing basic blocks. The arguments are the same as described in `UC_HOOK_CODE`.

```
def hook_block(uc, address, size, user_data):
     print("Tracing basic block at 0x%x, block size = 0x%x" %(address, size)
uc.hook_add(UC_HOOK_BLOCK, hook_block
```

UC_HOOK_MEM_*

The Unicorn Engine has a few hooks specifically for the reading, fetching, writing, and accessing of memory. They all start with UC_HOOK_MEM_*. Their callback all have the same arguments as seen below.

```
def hook_mem_example(uc, access, address, size, value, user_data):
    pass
```

The first argument is the Unicorn instance and the second argument is `access`. Their values can be seen below.

```
UC_MEM_READ = 16
UC_MEM_WRITE = 17
UC_MEM_FETCH = 18
UC_MEM_READ_UNMAPPED = 19
UC_MEM_WRITE_UNMAPPED = 20
UC_MEM_FETCH_UNMAPPED = 21
UC_MEM_WRITE_PROT = 22
```

```
UC_MEM_READ_PROT = 23
UC_MEM_FETCH_PROT = 24
UC_MEM_READ_AFTER = 25
```

UC_HOOK_MEM_INVALID

The example code of `UC_HOOK_MEM_INVALID` contains an example of comparing the access error. The callback `hook_mem_invalid` is executed when an invalid memory access occurs.

```
def hook_mem_invalid(uc, access, address, size, value, user_data):
       eip = uc.reg_read(UC_X86_REG_EIP)
       if access == UC_MEM_WRITE:
           print("invalid WRITE of 0x%x at 0x%X, data size = %u, data value = 0x%x"
% (address, eip, size, value))
       if access == UC_MEM_READ:
           print("invalid READ of 0x%x at 0x%X, data size = %u" % (address, eip,
size))
       if access == UC_MEM_FETCH:
           print("UC_MEM_FETCH of 0x%x at 0x%X, data size = %u" % (address, eip,
size))
       if access == UC_MEM_READ_UNMAPPED:
           print("UC_MEM_READ_UNMAPPED of 0x%x at 0x%X, data size = %u" % (address,
eip, size))
       if access == UC_MEM_WRITE_UNMAPPED:
           print("UC_MEM_WRITE_UNMAPPED of 0x%x at 0x%X, data size = %u" %
(address, eip, size))
       if access == UC_MEM_FETCH_UNMAPPED:
           print("UC_MEM_FETCH_UNMAPPED of 0x%x at 0x%X, data size = %u" %
(address, eip, size))
       if access == UC_MEM_WRITE_PROT:
           print("UC_MEM_WRITE_PROT of 0x%x at 0x%X, data size = %u" % (address,
eip, size))
       if access == UC_MEM_FETCH_PROT:
           print("UC_MEM_FETCH_PROT of 0x%x at 0x%X, data size = %u" % (address,
eip, size))
       if access == UC_MEM_FETCH_PROT:
           print("UC_MEM_FETCH_PROT of 0x%x at 0x%X, data size = %u" % (address,
eip, size))
       if access == UC_MEM_READ_AFTER:
           print("UC_MEM_READ_AFTER of 0x%x at 0x%X, data size = %u" % (address,
eip, size))
       return False
uc.hook_add(UC_HOOK_MEM_INVALID, hook_mem_invalid)
```

UC_HOOK_MEM_READ_UNMAPPED

`UC_HOOK_MEM_READ_UNMAPPED` is a hook that executes the callback when the emulated code attempts to read unmapped memory. The following snippet is an example.

```
def hook_mem_read_unmapped(uc, access, address, size, value, user_data):
    pass
uc.hook_add(UC_HOOK_MEM_READ_UNMAPPED, hook_mem_read_unmapped, None)
```

The following is a list of other memory hooks with a minimal description. Previous example snippets can be modified to use the below hooks.

- **UC_HOOK_MEM_WRITE_UNMAPPED**
  - Executes a callback when an invalid memory write event occurs
- **UC_HOOK_MEM_FETCH_UNMAPPED**
  - Executes a callback when an invalid memory fetch for execution event
- **UC_HOOK_MEM_READ_PROT**
  - Executes a callback when a memory read on read-protected memory occurs
- **UC_HOOK_MEM_WRITE_PROT**
  - Executes a callback when a memory write on write-protected memory occurs
- **UC_HOOK_MEM_FETCH_PROT**
  - Executes a callback when a memory fetch on non-executable memory occurs
- **UC_HOOK_MEM_READ**
  - Executes a callback when a memory read event occurs
- **UC_HOOK_MEM_WRITE**
  - Executes a callback when a memory write events"
- **UC_HOOK_MEM_FETCH**
  - Executes a callback when a memory fetch for execution event occurs
- **UC_HOOK_MEM_READ_AFTER**
  - Executes a callback when a successful memory read event occurs.

Now that we understand how the Unicorn Engine works, we use it the context of IDA. The assembly below allocates memory by calling `malloc`, copies the offset of an encrypted string, then XORs each byte of the string with a key and the stores the results in the allocated memory.

```
.text:00401034     push    esi
.text:00401035     push    edi
.text:00401036     push    0Ah                        ; Size
.text:00401038     call    ds:malloc
.text:0040103E     mov     esi, eax
.text:00401040     mov     edi, offset str_encrypted
.text:00401045     xor     eax, eax                   ; eax = 0
.text:00401047     sub     edi, esi
.text:00401049     pop     ecx
```

```
.text:0040104A
.text:0040104A loop:                                    ; CODE XREF: _main+28↓j
.text:0040104A     lea     edx, [eax+esi]
.text:0040104D     mov     cl, [edi+edx]
.text:00401050     xor     cl, ds:b_key
.text:00401056     inc     eax
.text:00401057     mov     [edx], cl
.text:00401059     cmp     eax, 9                       ; index
.text:0040105C     jb      short loop
.text:0040105E     push    esi
```

The code above is simple, but it contains several nuances that must be accounted for when emulating code. The first issues is at the call to `malloc` at offset `0x0401038`. The Unicorn Engine emulates instructions as if it were a CPU processor but not as if it was an emulator of the operating system. It does not act as the Window's Loader does. It does not initialize memory for an executable so it can execute. Memory mappings, loading of dynamic link libraries or the populating of the import table are not handled by the Unicorn Engine. It can execute position independent code if its self-contained and therefore doesn't rely on memory structures populated by the operating system (e.g. Process Environment Block). If those are attributes that are needed for successful execution, then those attributes need to be manually created and mapped or manually handled through a Hook and Callback. The second issue is at offset `0x0401040` with the moving of the offset of the encrypted string. The offset of the string is at the virtual offset `0x0402108`. If the code within the executable was treated as raw data with no memory mappings then executed offset would be `0x440` but trying to read the virtual address would return an invalid memory read because the memory hasn't been mapped. When an executable or the executable data within IDA's IDB the executed needs to map to the correct address. The last issue is executing only the XOR loop and ignoring other code and exceptions. The following code assumes that the user as highlighted the assembly from above from `0x401034` to `0x40105e`.

```
from unicorn import *
from unicorn.x86_const import *
import idautils
import math

VIRT_MEM = 0x4000

def roundup(x):
    return int(math.ceil(x / 1024.0)) * 1024

def hook_mem_invalid(uc, access, address, size, value, user_data):
    if uc._arch == UC_ARCH_X86:
```

```
        eip = uc.reg_read(UC_X86_REG_EIP)
    else:
        eip = uc.reg_read(UC_X86_REG_RIP)
    bb = uc.mem_read(eip, 2)
    if bb != b"\xFF\x15":
        return
    if idc.get_name(address) == "malloc":
        uc.mem_map(VIRT_MEM, 8 * 1024)
    if uc._arch == UC_ARCH_X86:
        uc.reg_write(UC_X86_REG_EAX, VIRT_MEM)
        cur_addr = uc.reg_read(UC_X86_REG_EIP)
        uc.reg_write(UC_X86_REG_EIP, cur_addr + 6)
    else:
        cur_addr = uc.reg_read(UC_X86_REG_RIP)
        uc.reg_write(UC_X86_REG_RIP, cur_addr + 6)

def hook_code(uc, address, size, user_data):
    """For Debugging Use Only"""
    print('Tracing instruction at 0x%x, instruction size = 0x%x' % (address, size))

def emulate():
    try:
        # get segment start and end address
        segments = []
        for seg in idautils.Segments():
            segments.append((idc.get_segm_start(seg), idc.get_segm_end(seg)))

        # get base address
        BASE_ADDRESS = idaapi.get_imagebase()

        # get bit
        info = idaapi.get_inf_structure()
        if info.is_64bit():
            mu = Uc(UC_ARCH_X86, UC_MODE_64)
        elif info.is_32bit():
            mu = Uc(UC_ARCH_X86, UC_MODE_32)

        # map 8MB memory for this emulation
        mu.mem_map(BASE_ADDRESS - 0x1000, 8 * 1024 * 1024)
```

```
        # write segments to memory
        for seg in segments:
            temp_seg = idc.get_bytes(seg[0], seg[1] - seg[0])
            mu.mem_write(seg[0], temp_seg)

        # initialize stack
        stack_size = 1024 * 1024
        if info.is_64bit():
            stack_base = roundup(seg[1])
            mu.reg_write(UC_X86_REG_RSP, stack_base + stack_size - 0x1000)
            mu.reg_write(UC_X86_REG_RBP, stack_base + stack_size)
        elif info.is_32bit():
            stack_base = roundup(seg[1])
            mu.reg_write(UC_X86_REG_ESP, stack_base + stack_size - 0x1000)
            mu.reg_write(UC_X86_REG_EBP, stack_base + stack_size)

        # write null bytes to the stack
        mu.mem_write(stack_base, b"\x00" * stack_size)

        # get selected address range
        start = idc.read_selection_start()
        end = idc.read_selection_end()
        if start == idc.BADADDR:
            return

        # add hook
        mu.hook_add(UC_HOOK_MEM_READ, hook_mem_invalid)
        mu.hook_add(UC_HOOK_CODE, hook_code)

        mu.emu_start(start, end)
        decoded = mu.mem_read(VIRT_MEM, 0x0A)
        print(decoded)

    except UcError as e:
        print("ERROR: %s" % e)
        return None
    return mu
```

```
emulate()
```

Starting at the function `emulate`, it iterates through all the segments in IDB by calling `idautils.Segments()` and extracting the segments start address by calling `idc.get_segm_start(seg)` and then getting the end address using `idc.get_segm_end(seg)`. The base address is retrieved by calling `idaapi.get_imagebase()`. After the base address is retrieved, we call `idaapi.get_inf_structure()` to get an instance of the `idainfo` structure. Using the `idainfo` structure stored in `info`, we call `info.is_64bit()` or `info.is_32bit()` to determine the bit of the IDB. Since this is a 32bit executable, `Uc(UC_ARCH_X86, UC_MODE_32)` is called. This setup the Unicorn instance to execute the code as X86 in 32-bit mode and stores the instance in the variable `mu`. The 8MB of memory is allocated at the base address minus 1000 by calling `mu.mem_map(BASE_ADDRESS - 0x1000, 8 * 1024 * 1024)`. After the memory is mapped, it can then be written to. If a write, read, or access is attempted to memory that is not mapped an error will occur. The segment data are written to their corresponding memory offsets. After the segments are written, the stack memory is allocated. The Base Pointer and Stack Pointer registers are written to by calling `mu.reg_write(UC_X86_REG_ESP, stack_base + stack_size - 0x1000)` and `mu.reg_write(UC_X86_REG_EBP, stack_base + stack_size)`. The first argument is the `regid` (e.g. `UC_X86_REG_ESP)` and the second value is the value to be written to the register. The stack is initialized with null bytes ("`\0x00`") and the selected addresses is extracted. The hook `UC_HOOK_MEM_READ` with a callback of `hook_mem_invalid` and the hook `UC_HOOK_CODE` with a callback of `hook_code`. The code is emulated by calling `mu.emu_start(start, end)`, with `start` and `end` being populated with the selected offsets. Once the emulation is finished, it reads the XORed string and prints it.

The first hook that is triggered is `UC_HOOK_MEM_READ` which occurs when Unicorn tries to read the address that malloc should be mapped to. Once the hook occurs, the callback function `hook_mem_invalid` is executed. In this callback, we write our own custom malloc by allocating memory, writing the offset to EAX or RAX and then returning. To determine if EAX or RAX should be written to, we can retrieve the architecture stored with the Unicorn instance by comparing `uc._arch` to `UC_ARCH_X86`. Another option would be to pass the results of `info.is_32bit()` as an optional argument in `user_data`. EIP is read by calling `uc.reg_read(reg_id)` with an argument of `UC_X86_REG_EIP`. Next, we read two bytes by calling `uc.mem_read(int, size)` with the first argument being the offset stored in EIP and the second argument being the size of the data to read. The two bytes are compared to "`\xFF\x15`" to ensure that the exception happened at a `call` instruction. If a `call` instruction, the name of the address is retrieved by using `idc.get_name(ea)` and checking the API address's name is `malloc`. If `malloc`, memory is mapped using `uc.mem_map(address, size)` and then we write the address to the register EAX `uc.reg_write(UC_X86_REG_EAX, VIRT_MEM)`. To bypass the memory exception, we need to set EIP to the address the call to malloc (`0x40103E`). To write to EIP, we use `uc.reg_write(reg_id, value)` with the value being the address of EIP + 6. The second hook of `UC_HOOK_CODE`, with a callback of `hook_code` print the current address that is being emulated and the size of it. Below is the output of the Unicorn emulated ran in IDA. The last line contains the decrypted string.

```
Tracing instruction at 0x401034, instruction size = 0x1
Tracing instruction at 0x401035, instruction size = 0x1
```

```
Tracing instruction at 0x401036, instruction size = 0x2
..removed..
Tracing instruction at 0x401059, instruction size = 0x3
Tracing instruction at 0x40105c, instruction size = 0x2
Tracing instruction at 0x40105e, instruction size = 0x1
bytearray(b'test mess\x00')
```

Debugging

Hey **OAlabs**, *psyche*. Check the next version 😉

What's Next?

If you have made it this far odds are you looking to dig into some projects to learn from. You might as well check out HexRays and the IDAPython source code.

HexRays
Website: https://www.hex-rays.com/
Blog: https://www.hex-rays.com/blog/

IDAPython Source Code
Repo: https://github.com/idapython/src

The following is a list (alphabetic order by last name) of individuals that I would I recommend reading through their projects. I personally know or have met all these individuals and can't say enough good things about their work.

Tamir Bahar
Twitter: @tmr232
Repo: https://github.com/tmr232

Willi Ballenthin
Twitter: @williballenthin
Repo: https://github.com/williballenthin
Blog: http://www.williballenthin.com/

Daniel Plohmann
Twitter @push_pnx
Repo: https://github.com/danielplohmann
https://bitbucket.org/daniel_plohmann/simplifire.idascope/
Blog: http://byte-atlas.blogspot.com
https://pnx-tf.blogspot.com/

Rolf Rolles
Twitter: @RolfRolles
Repo: https://github.com/RolfRolles

Blog: http://www.msreverseengineering.com/
Training: https://www.msreverseengineering.com/training/

Open Analysis
Site: https://www.openanalysis.net/
Twitter: @herrcore & @seanmw
Youtube: https://www.youtube.com/channel/UC--DwaiMV-jtO-6EvmKOnqg/videos

Closing

I hope you gained some knowledge on how to use IDAPython or a trick to solve an issue you are working on. As I stated in the beginning of this book, I commonly forget IDA's API usage. Something that has helped me (along with writing this book) to remember the APIs is cut and paste almost all my IDAPython snippets to GitHub's Gist[20]. You would be surprised how often you can write the same functionality over and over, once you know how powerful they are. Having them quickly accessible saves a lot of time.

If you have any questions, comments or feedback please send me an email. I plan to keep editing the book. Please note the version number and check it out again in the future. Cheers.

Future Chapters

This is a list of future chapters that I plan on adding.

- HexRays Decompiler
- Interacting with IDA
- Debugger…
- Reimplement Pintools

A current list of chapters or issues can be found on GitHub[21]. I will also be creating a video training series in the upcoming months. Further details will be released in future versions.

Appendix

Unchanged IDC API Names

```
GetLocalType
AddSeg
SetType
GetDisasm
SetPrcsr
GetFloat
GetDouble
```

[20] https://gist.github.com/alexander-hanel
[21] https://github.com/alexander-hanel/BeginnersGuideToIDAPython/issues

```
AutoMark
is_pack_real
set_local_type
WriteMap
WriteTxt
WriteExe
CompileEx
uprint
form
Appcall
ApplyType
GetManyBytes
GetString
ClearTraceFile
FindBinary
FindText
NextHead
ParseTypes
PrevHead
ProcessUiAction
SaveBase
eval
MakeStr
GetProcessorName
SegStart
SegEnd
SetSegmentType
CleanupAppcall
DelUserInfo
```

PeFile

Pefile is a multi-platform Python module to parse Portable Executables files. It was written and maintained by Ero Carrera. The following Python code contains some of the most common usages and output of pefile. Please see pefile GitHub repo for more information.

```
import pefile
import sys
import datetime
import zlib
```

```
"""
    Author:     Alexander Hanel
    Summary:    Most common pefile usage examples
"""


def pefile_example(_file, file_path=True):
    try:
        if file_path:
            # load executable from file path to create PE class
            pe = pefile.PE(_file)
        else:
            # load executable from buffer/string to create PE class
            pe = pefile.PE(data=_file)
    except Exception as e:
        print("pefile load error: %s" % e)
        return
    print("IMAGE_OPTIONAL_HEADER32.AddressOfEntryPoint=0x%x" %
pe.OPTIONAL_HEADER.AddressOfEntryPoint)
    print("IMAGE_OPTIONAL_HEADER32.ImageBase=0x%x" % pe.OPTIONAL_HEADER.ImageBase)
    # Now use AddressOfEntryPoint to get the preferred Virtual Address of Entry
Point
    print("RVA (preferred) Entry Point=0x%x" % (pe.OPTIONAL_HEADER.ImageBase +
pe.OPTIONAL_HEADER.AddressOfEntryPoint))
    print("CPU TYPE=%s" % pefile.MACHINE_TYPE[pe.FILE_HEADER.Machine])
    print("Subsystem=%s" % pefile.SUBSYSTEM_TYPE[pe.OPTIONAL_HEADER.Subsystem])
    print("Compile Time=%s" %
datetime.datetime.fromtimestamp(pe.FILE_HEADER.TimeDateStamp))
    ext = ""
    if pe.is_dll():
        ext = ".dll"
    elif pe.is_driver():
        ext = '.sys'
    elif pe.is_exe():
        ext = '.exe'
    if ext:
        print("FileExt=%s" % ext)
    # parse sections
    print("Number of Sections=%s" % pe.FILE_HEADER.NumberOfSections)
```

```
    print("Section VirtualAddress VirtualSize SizeofRawData CRC Hash")
    for index, section in enumerate(pe.sections):
        # how to read the section data
        sec_data = pe.sections[index].get_data()
        # simple usage
        crc_hash = zlib.crc32(sec_data) & 0xffffffff
        print("%s 0x%x 0x%x 0x%x 0x%x" % (section.Name, section.VirtualAddress, 
section.Misc_VirtualSize, section.SizeOfRawData, crc_hash))
    print("Imported DLLs")
    for entry in pe.DIRECTORY_ENTRY_IMPORT:
        # print dll name
        print(entry.dll)
        print("\tImport Address, Name, File Offset")
        for imp in entry.imports:
            # calculate virtual address to file offset
            file_offset = pe.get_offset_from_rva(imp.address - 
pe.OPTIONAL_HEADER.ImageBase)
            # print symbol name
            print("\t0x%x %s 0x%x" % (imp.address, imp.name, file_offset))


path = sys.argv[1]
pefile_example(path)
```