Sark Documentation

Release 0.1.0

Tamir Bahar

Sep 02, 2020

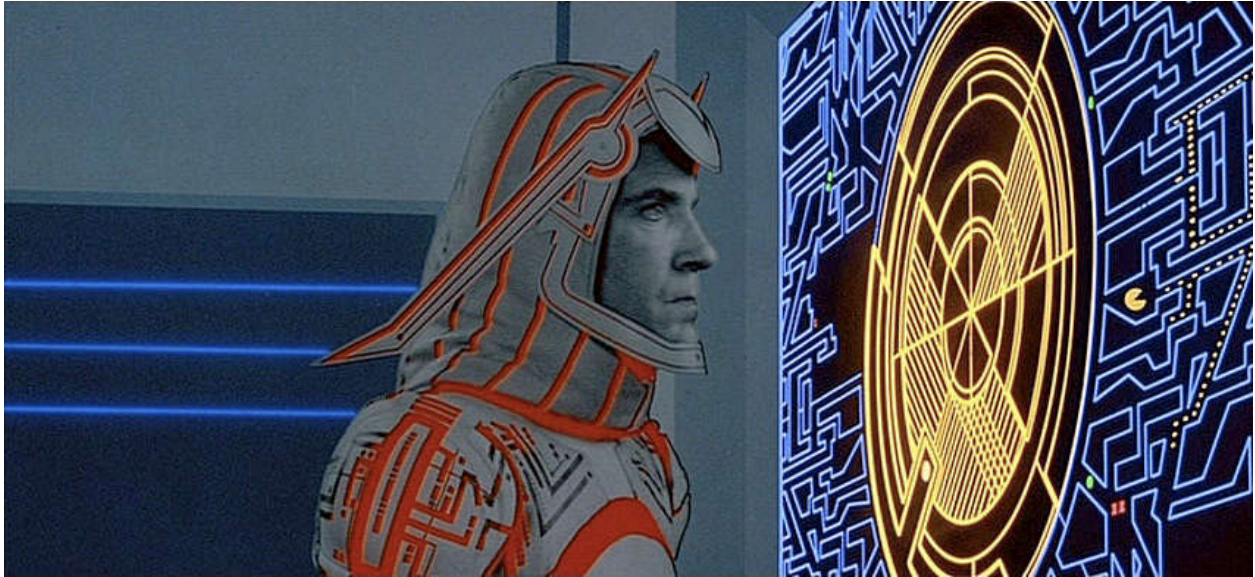# Contents

Sark (named after the notorious Tron villain) is an object-oriented scripting layer written on top of IDAPython. Sark is easy to use and provides tools for writing advanced scripts and plugins.

Getting Started

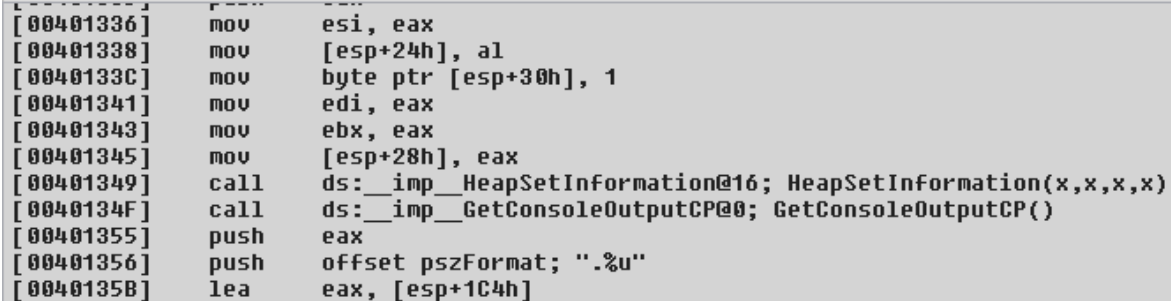Install Sark from the command line:

```
pip install -U git+https://github.com/tmr232/Sark.git#egg=Sark
```

Import inside IDA, and start having fun!

```python
import sark
import idaapi

# Get the current function
func = sark.Function()

# Print all lines in the function
for line in func.lines:
    idaapi.msg("{}\n".format(line))
```

```
[00401336]    mov     esi, eax
[00401338]    mov     [esp+24h], al
[0040133C]    mov     byte ptr [esp+30h], 1
[00401341]    mov     edi, eax
[00401343]    mov     ebx, eax
[00401345]    mov     [esp+28h], eax
[00401349]    call    ds:__imp__HeapSetInformation@16; HeapSetInformation(x,x,x,x)
[0040134F]    call    ds:__imp__GetConsoleOutputCP@0; GetConsoleOutputCP()
[00401355]    push    eax
[00401356]    push    offset pszFormat; ".%u"
[0040135B]    lea     eax, [esp+1C4h]
Python
```

```python
# Mark all the lines containing xrefs outside the function
for xref in func.xrefs_from:
    sark.Line(xref.frm).color = 0x8833FF
```

Table of Contents

Contents:

2.1 Introduction

Even with books like Alexander Hanel's The Beginner's Guide to IDAPython, writing IDA scripts still remains a daunting task. The need to dive into the IDA SDK's header files (all 54 of them), read `idaapi.py`, `idc.py` and `idautils.py`, and preferably some existing plugins as well, wards off many researchers and keeps the script & plugin writing community small.

Being a researcher myself, I wanted to make scripting IDA a bit easier and more intuitive. I wanted to spend the majority of my (scripting) time writing code (be it in a code editor or an interactive shell) and not reading someone else's (I prefer spending my reading efforts on assembly.) So I created Sark.

Sark, (named after the notorious Tron villain,) is an object-oriented scripting layer written on top of IDAPython to provide ease of use, as well as additional tools for writing advanced scripts and plugins.

This tutorial will show you the basics of Sark, to get you started right away.
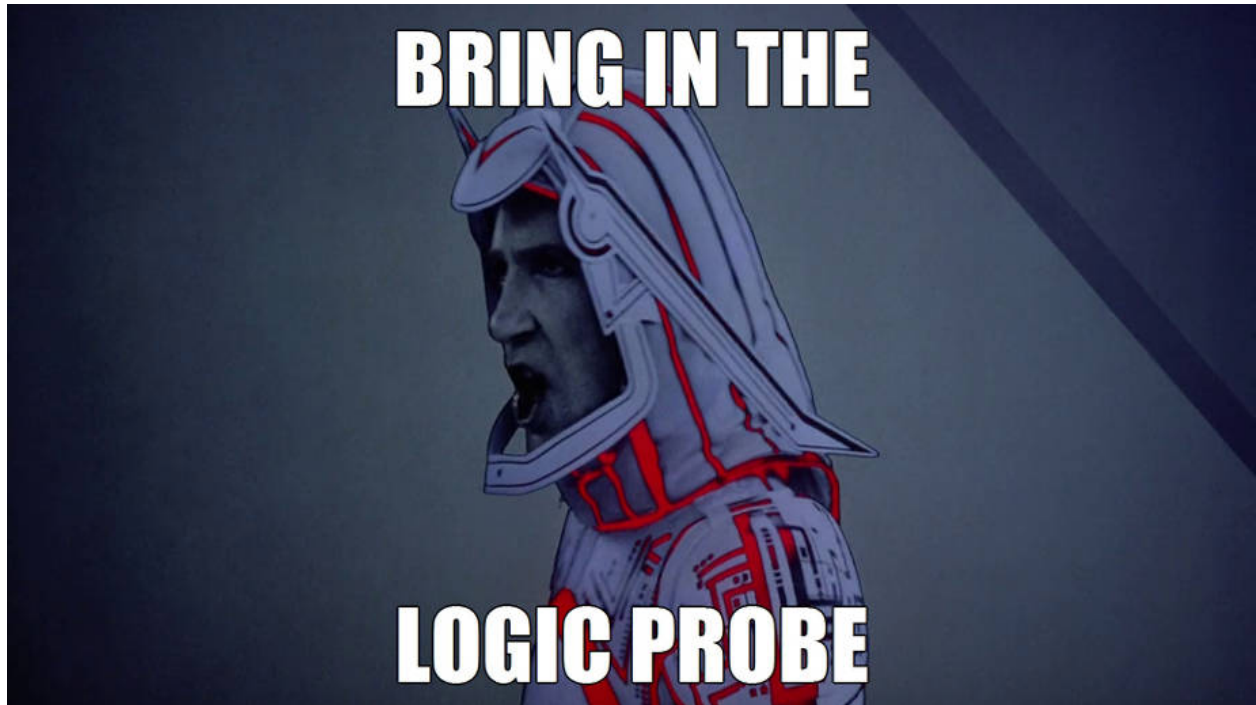
2.2 Installation

2.2.1 For Sark Users

To get the bleeding edge version, use:

```
pip install -U git+https://github.com/tmr232/Sark.git#egg=Sark
```

To install the IDA Plugins (optional) download the entire repository from GitHub and read *Installing Plugins*.

Updates

To update Sark to the latest version, just run the installation command again.

2.2.2 For Sark Developers

If you want to help in the development of Sark, follow this.

Clone the Sark repository to get the latest version

```
git clone https://github.com/tmr232/Sark.git && cd Sark
pip install -e .
```

Updates

To update Sark to the latest version (including all *installed* codecs and plugins) simply pull the latest version from the repo

```
git pull
```

2.3 API

2.3.1 Tutorial Conventions

IDA IPython is used in the examples unless stated otherwise. For brevity, assume the following code precedes any example code.

```python
import idaapi, idc, idautils
import sark
```

As Sark is a large, evolving library, the API documentation provided here will be partial and only include what is needed to get you started. However, the Sark code in itself is heavily documented.

2.3.2 Lines

Lines are the most basic and intuitive object in Sark. A line in the IDA-View is a line in Sark. Let's have a look.

```python
>>> my_line = sark.Line()   # Same as `sark.Line(ea=idc.here())`
>>> print my_line
[00417401]    mov     ebp, esp

>>> my_line.comments.regular = "The line at 0x{:08X}".format(my_line.ea)
>>> print my_line
[00417401]    mov     ebp, esp        ; The line at 0x00417401
```

The `sark.Line` object encapsulates most of the line-relevant functions of IDAPython. Some examples include:

Member	Usage
ea	line's address
comments	line comments
name	the name of the line (if any)
insn	assembly instruction
xrefs_to	cross references to the line
xrefs_from	cross references from the line
bytes	the actual bytes in the line

For the rest, I suggest reading the highly documented code, or using the interactive shell to experiment with the `sark.Line` object.

The line object contains 4 notable members: `comments`, `insn` and the `xrefs_*` pair.

Line Comments

The `comments` member provides access to all comment types: - Regular comments - Repeating comments - Anterior lines - Posterior lines

It allows you to get, as well as set comments. Each change to the comments will cause the UI to refresh.

```
>>> anterior = my_line.comments.anterior
>>> my_line.comments.regular = "My Regular Comment"
```

Line Xrefs

Provide access to `Xref` objects describing the line's cross references. `Xref` objects will be discussed later under *Xrefs*.

Instructions

Provide access to the line's instructions, down to the single operand. `Instruction` objects will be discussed later under *Instructions*.

Getting Lines

There are several ways to get lines. Either directly or from other objects.

Method	Effect
A Single Line	
`sark.Line()`	Get the current line
`sark.Line(ea=my_address)`	Get the line at the given address
`sark.Line(name=some_name)`	Get the line with the given name
Multiple Lines	
`sark.lines()`	Iterate all lines in the IDB
`sark.lines(start=start_ea, end=end_ea)`	Iterate all lines between `start_ea` and `end_ea`
`sark.lines(selection=True)`	Iterate all lines in current selection
`sark.lines(reverse=True)`	Iterate lines in reverse order

Objects that contain lines, such as functions and code blocks, can return their own set of lines. See `sark.Function().lines` for an example.

2.3.3 Functions

Functions are another basic object in Sark. Each one provides access to a single function in IDA.

```
>>> my_func = sark.Function()  # The same arguments as `sark.Line`
>>> print my_func
Function(name="sub_417400", addr=0x00417400)

>>> my_func.name = "my_func"
>>> print my_func
Function(name="my_func", addr=0x00417400)

>>> for line in my_func.lines:
...     print line.disasm
push    ebp
mov     ebp, esp
sub     esp, 0DCh
push    ebx
push    esi
```

<div style="text-align: right">(continues on next page)</div>

```
.
.
.
```

Like the `sark.Line` objects, they encapsulate relevant API into a single object. Some useful members are:

Member	Usage
startEA	starting address
endEA	end address
ea	alias for `startEA` (for comparability with `sark.Line`)
comments	function comments
name	function name
flags	function flags
lines	all the lines in the function (a generator)
xrefs_*	xrefs to and from the function[1]

All similarly named members between `sark.Line` and `sark.Function` work similarly as well to avoid confusion.

Getting Functions

There are 2 ways to get functions:

1. Using the `sark.Function` class, which accepts the same arguments as `sark.Line`;

2. Using `sark.functions` to iterate over functions. It is the same as `sark.lines`, but does not accept a `reverse` argument.

2.3.4 Xrefs

Cross references are a core concept in IDA. They provide us with links between different objects and addresses throughout an IDB.

```
>>> for xref in sark.Line().xrefs_from:
...     print xref
<Xref(frm=0x0041745B, to=0x0041745D, iscode=1, user=0, type='Ordinary_Flow')>
<Xref(frm=0x0041745B, to='loc_4174A4', iscode=1, user=0, type='Code_Near_Jump')>

>>> for xref in sark.Line().xrefs_from:
...     if xref.type.is_jump:
...         print xref
<Xref(frm=0x0041745B, to='loc_4174A4', iscode=1, user=0, type='Code_Near_Jump')>
```

Sark xrefs are pretty compact objects:

Member	Usage
frm	xref source address
to	xref destination address
iscode	is code xref
user	is user defined xref
type	`XrefType` object

---

[1] Xrefs from a function include **only** references with a target outside the function. So recursion will be ignored.

XrefType

To make querying the type of the xref as easy as possible, the `XrefType` object was created:

Member	Usage
name	a string representing the type, mainly for display
type	the numeric type constant, as per IDA SDK
is_call	is the xref a call
is_jump	is the xref a jump
is_*	predicates to check if a specific type applies. Includes all xref types.

Usage is quite simple and looks like plain English (of sorts):

```
>>> if xref.type.is_jump:
...     print "xref is jump."
```

Getting Xrefs

Xrefs can be retrieved from lines or functions. Both objects have `xrefs_from` and `xrefs_to` properties that allow retrieval of the relevant xrefs.

2.3.5 Instructions

As promised - we arrive to discuss the instruction objects. Instruction objects represent the actual assembly code of each line.

```
>>> line = sark.Line()
>>> insn = line.insn
>>> print line
[00417555]    mov    ecx, [eax+8]

>>> print insn.mnem
mov

>>> print insn.operands
[<Operand(n=0, text='ecx')>, <Operand(n=1, text='[eax+8]')>]
```

Out of their members,

Member	Usage
operands	list of operands
mnem	opcode mnemonic
has_reg	is a reg used in the instruction
regs	the registers used in the instruction

`Instruction.operands` is the most interesting one.

Operands

Each operand provides the means to analyze individual operands in the code.

```
>>> print insn.operands[1]
<Operand(n=1, text='[eax+8]')>

>>> print "{0.reg} + {0.offset}".format(insn.operands[1])
eax + 8
```

Member	Usage
n	operand index in instruction
type	numeric type a-la IDA SDK
size	data size of the operand
is_read	is the operand read from
is_write	is the operand written to
reg	the register used in the operand
text	the operand text, as displayed in IDA
base	the `base` register in an address-phrase of the form `[base + index * scale + offset]`
index	the `index` register in a phrase
scale	the `scale` in a phrase
offset	the `offset` in a phrase

Getting Instructions

The best way to retrieve instruction objects is using the `.insn` member of `sark.Line`.

2.3.6 Code Blocks

If you ever looked at a function in the Graph-View, you know what code blocks are. They are the nodes in the function graph, sometimes referred to as a flowchart.

```
>>> block = sark.CodeBlock()
>>> print list(block.next)
[<CodeBlock(startEA=0x00417567, endEA=0x00417570)>,
 <CodeBlock(startEA=0x0041759E, endEA=0x004175D4)>]
```

Sark's `CodeBlock` object inherits from the `idaapi.BasicBlock` objects, and adds a few handy members.

Member	Usage
lines	the lines in the block, as a generator
next	successor nodes, as a generator
prev	predecessor nodes, as a generator
color	the background color of the node

These members allow for easy traversal and analysis of nodes in a graph.

FlowChart

Sark's flowchart, inheriting from `idaapi.FlowChart`, is in every way the same except for returning Sark `CodeBlock` objects instead of `idaapi.BasicBlock` ones. It can be used to quickly fetch all the blocks in a function graph.

Getting Codeblocks

Codeblocks are created using the `sark.CodeBlock(ea)` class. Flowcharts can be retrieved using the `sark.FlowChart(ea)` class accordingly.

In some cases, you may want to go over more than one function. In those cases, you can use the `sark.codeblocks(start=None, end=None, full=True)` function. The `full` parameter controls the way the blocks are generated. With `full=True`, `FlowChart` objects are generated per function, yielding fully capable `CodeBlock` objects. With `full=False`, a single `FlowChart` is generated for the entire address range. This results in faster iteration, but since the blocks are not associated to their containing functions, it is not possible to get or set block colors (line color will change, though.)

Advanced Usage

Since the function flowchart is actually a graph, it makes sense to use it as one. To ease you into it, the `sark.get_nx_graph(ea)` function was added.

```
>>> sark.get_nx_graph(idc.here())
<networkx.classes.digraph.DiGraph at 0x85d6570>
```

The function returns a [NetworkX](#) `DiGraph` object representing the flowchart, with each node being the `startEA` of a matching block. Using NetworkX's functionality, it is easy to trace routes in the graph.

```
>>> import networkx as nx
>>> func = sark.Function()
>>> graph = sark.get_nx_graph(func.ea)
>>> start_address = sark.get_block_start(func.startEA)  # The `get_block_start(ea)`
→is short for `get_codeblock(ea).startEA`
>>> end_address = sark.get_block_start(func.endEA - 1)  # Remember, `endEA` is
→outside the function!
>>> path = nx.shortest_path(graph, start_address, end_address)
>>> print "From {} to {}".format(hex(start_address), hex(end_address))
From 0x417400L to 0x4176a6L

>>>print " -> ".join(map(hex, nx.shortest_path(graph, start, end)))
0x417400L -> 0x41745dL -> 0x417483L -> 0x417499L -> 0x4176a6L
```

2.3.7 Segments

Though not as popular as functions and lines, IDA segments include both. In Sark, `Segment` objects allow access to underlying `Function` and `Line` objects.

```
>>> #
>>> # Reference Lister
>>> #
>>> # List all functions and all references to them in the current section.
>>> #
>>> # Implemented with Sark
>>> #
>>> # See reference implementation here: https://code.google.com/p/idapython/wiki/
→ExampleScripts
>>> #
>>> for function in sark.Segment().functions:
>>>     print "Function %s at 0x%x" % (function.name, function.ea)
```

```
>>>     for ref in function.crefs_to:
>>>         print "  called from %s(0x%x)" % (sark.Function(ref).name, ref)
```

Like the `sark.Line` objects, they encapsulate relevant API into a single object. Some useful members are:

Member	Usage
startEA	starting address
endEA	end address
ea	alias for `startEA` (for comparability with `sark.Segment`)
comments	segment comments
name	segment name
lines	all the lines in the segment (a generator)
functions	all the functions in the segment (a generator)
size	the size of the segment
permissions	the segments permissions (r/w/x). Can be modified.
next	the next segment.
bitness	the bitness of the segment (16, 32 or 64.)

All similarly named members between `sark.Line` and `sark.Segment` work similarly as well to avoid confusion.

Getting Segments

There are 2 ways to get segments:

1. Using the `sark.Segment` object, using an address in a segment, a segment name, or the index of a segment.

2. Using `sark.segments` to iterate over segments.

2.3.8 Switch

The switch-case is a common construct in compiled code, and IDA is doing a great job at analyzing it.

```
>>> switch = sark.Switch(idc.here())
>>> for case, target in switch:
...     print "{} -> 0x{:08X}".format(case, target)
0 -> 0x004224C9
1 -> 0x0042249F
2 -> 0x0042251B
3 -> 0x0042251B
4 -> 0x00422475
5 -> 0x0042251B
6 -> 0x0042251B
7 -> 0x0042251B
8 -> 0x004224F3
9 -> 0x0042251B
10 -> 0x0042251B
11 -> 0x00422448
```

It provides the following members

Member	Usage
targets	switch target addresses
cases	switch case values
pairs	iterator of `(case, target)` pairs
get_cases	get the cases matching a target

The `sark.Switch` object is similar to a Python `dict`, mapping cases to targets. `switch[case]` returns the relevant target, and iteration returning the cases.

Getting Switches

To check if an address is a switch address, use `sark.is_switch(ea)`. To get the switch, use `sark.Switch(ea)`.

2.3.9 Enums

Enums in IDA are a great way to name numbers and bit-values for easier reading.

```
>>> for enum in sark.enums():
...     print "{}:".format(enum.name)
...     for member in enum.members:
...         print "    {:<30} = {}".format(member.name, member.value)
...     print
POOL_TYPE:
    NonPagedPool                   = 0
    PagedPool                      = 1
    NonPagedPoolMustSucceed        = 2
    DontUseThisType                = 3
    NonPagedPoolCacheAligned       = 4
    PagedPoolCacheAligned          = 5
    NonPagedPoolCacheAlignedMustS  = 6
    MaxPoolType                    = 7

CREATE_FILE_TYPE:
    CreateFileTypeNone             = 0
    CreateFileTypeNamedPipe        = 1
    CreateFileTypeMailslot         = 2
```

The Sark `Enum` object provides the following members:

Member	Usage
name	the enum name
comments	enum comments, similar to line comments
eid	the enum-id of the enum
bitfield	is the enum a bitfield
members	the enum member constants

Using the `Enum` object you can easily enumerate and manipulate enums in IDA.

Enum Members

The `.members` member of `sark.Enum` returns a members object. The members object allows easy enumeration and manipulation of the members:

```
>>> my_enum = sark.add_enum("MyEnum")
>>> my_enum.members.add("first", 0)
>>> my_enum.members.add("second", 1)
>>> my_enum.members.add("third", 2)
>>> my_enum.members.remove("second")
>>> for member in my_enum.members:
...     print "{} = {}".format(member.name, member.value)
first = 0
third = 2
```

Each member provides the following:

Member	Usage
name	the member name
value	the member value
comments	the member comments
enum	the containing enum

Getting Enums

There are several ways to get an enum. All are summed in the following table:

Code	Explanation
`sark.enums()`	iterate all the enums in the IDB
`sark.Enum("EnumName")`	get an existing enum by name
`sark.Enum(eid=enum_id)`	get an enum using a known id
`sark.add_enum("NewEnumName")`	create a new enum

2.3.10 IDB Graphs

Earlier we discussed codeblock graphs inside functions. Another interesting graph is the call graph connecting all the functions.

As we have already played with graphs earlier, we will not delve into the details.

Getting IDB Graphs

To get an IDB graph, use `sark.graph.get_idb_graph()`. The function traverses all xrefs from and to all functions to create a graph of the IDB, with each node being the address of a function's `startEA`.

2.3.11 UI

Sark provides some basic utilities and wrappers for IDA's UI.

NXGraph

A natural extension to creating and analyzing graphs, is plotting them. IDA provides a generic API via the `idaapi.GraphViewer` interface. As Sark mainly uses NetworkX digraphs, the `sark.ui.NXGraph` class has been created to provide an easy plotting solution.

```
>>> viewer = sark.ui.NXGraph(graph, title="My Graph", handler=sark.ui.
→AddressNodeHandler())
>>> viewer.Show()
```

The `NXGraph` constructor takes several arguments:

Argument	Desctription
graph	the graph to plot
title	(opt.) title for the graph
handler	(opt.) a default handler for nodes
padding	(opt.) visual padding of nodes

After an `NXGraph` is created, use `.Show()` to display it.

Node Handlers

To allow different types of node data, `NXGraph` uses node handlers. Node handlers inherit from `sark.ui.BasicNodeHandler` and implement the callbacks required for them (all are optional).

Callback	Usage
on_get_text	returns the text to display for the node
on_click	handles a click on the node. Return `True` to set the cursor on it.
on_double_click	same as `on_click`
on_hint	the hint to show
on_bg_color	returns the background color for the node
on_frame_color	returns the frame (border) color for the node

There are 2 existing handlers you can use.

Handler	Info
`BasicNodeHandler`	The most basic handler. Calls `str` to get node text, and nothing else. This is the default handler for `NXGraph`.
`AddressNodeHandler`	Assumes all nodes are IDB addresses. For node text, it shows the address' name if it exists, or a hex address otherwise. On double click, it jumps to the clicked address.

Menu Manager

Sark provides a menu-manager class to allow the addition of top-level menus to IDA's GUI. This is done by abusing QT to find the top level menu, but you don't need to worry about that.
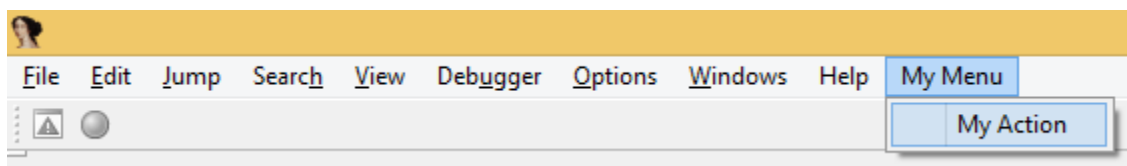
```
>>> # Use the manager to add top-level menus
>>> menu_manager = sark.ui.MenuManager()
>>> menu_manager.add_menu("My Menu")
>>> # Use the standard API to add menu items
>>> # Assume the action's text is "My Action"
>>> idaapi.attach_action_to_menu("My Menu/", "SomeActionName", idaapi.SETMENU_APP)
>>> # When a menu is not needed, remove it
>>> menu_manager.remove_menu("My Menu")
>>> # When you are done with the manager (and want to remove all menus you added.)
>>> # clear it before deleting.
>>> menu_manager.clear()
```

As you can see in the above code, the *MenuManager* class only handles the addition of a top-level menu. After that, IDA's own APIs can be used freely with the created menu to add or remove menu items

2.3.12 Miscellaneous

Sark also has a lot of functionality outside of the core objects.

Function	Description
`sark.fix_addresses(start=None, end=None)`	returns a `start, end` pair, where `None` is replaced with the start-address and end-address of the IDB accordingly
`sark.is_same_function(ea1, ea2)`	checks if the addresses are in the same function
`sark.get_name_or_address(ea)`	returns the name of the address if it exists. Otherwise a hex representation is returned

2.4 Examples

2.4.1 Capture Huge Screenshots

Usage

Click on the image for full scale screenshot.

Warning: really big image file.

Code

```python
import sark.qt

widget = sark.qt.get_widget("IDA View-A")
sark.qt.resize(widget, 7000, 18000)

# Move the view about a bit to capture the entire function

sark.qt.capture_widget(widget, "huge-screenshot.png")

# Crop the image to remove extra background.
```
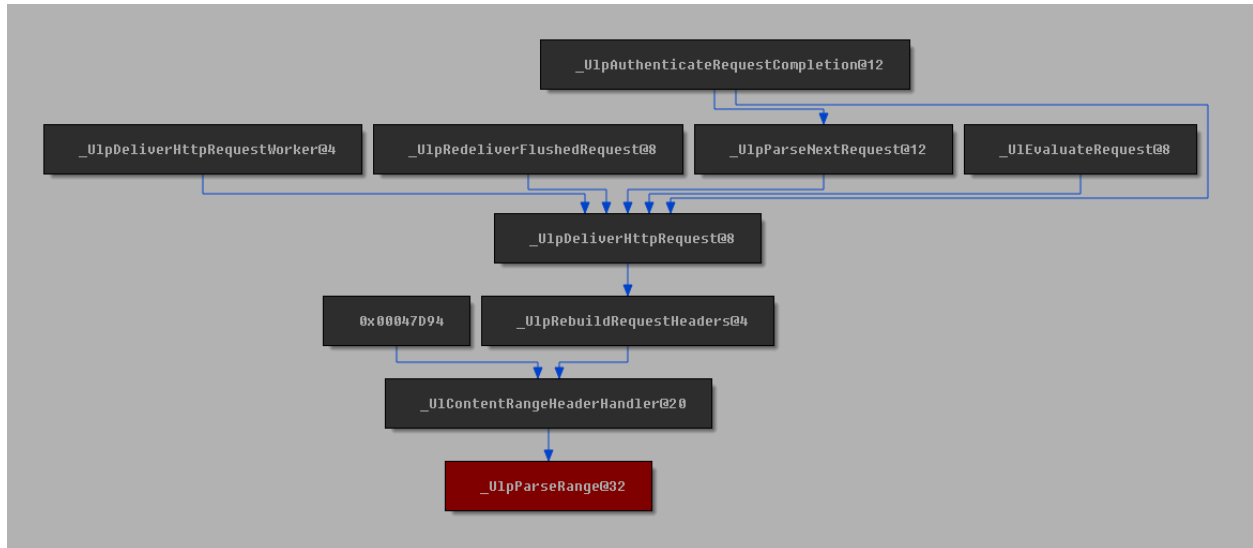
2.4.2 Plotting a Call Graph

Usage

Using Windows 8.1 `http.sys`. Before `MS15-034`.

```
draw_call_graph(sark.Function(name="_UlpParseRange@32").ea, to=True, distance=4)
```


Code

```python
import sark
import networkx as nx

def draw_call_graph(ea, distance=2, to=False):
    # First, get the IDB graph (caching it might be a good idea
    # as this operation can be time consuming on large IDBs)
    idb_graph = sark.graph.get_idb_graph()

    # Get the address of the function to use in the graph
    func_ea = sark.Function(ea).ea

    # Create the call graph
    if to:
        # If we want the calls to our function, we need to reverse
        # the graph
        idb_graph = idb_graph.reverse()

    # Use NetworkX to limit the IDB graph
    call_graph = nx.ego_graph(idb_graph, func_ea, distance)

    # Paint the root node red
    call_graph.node[func_ea][sark.ui.NXGraph.BG_COLOR] = 0x80

    if to:
        # If we reversed it before, we need to reverse it again
```

(continues on next page)

```
        # to make the links point the right way
        call_graph = call_graph.reverse()

    # Create an NXGraph viewer
    viewer = sark.ui.NXGraph(call_graph, handler=sark.ui.AddressNodeHandler())

    # Show the graph
    viewer.Show()
```

2.5 Plugins

2.5.1 Installing Plugins

The IDA Way

IDA provides a single way to install plugins - stick them in the `plugins` subdirectory and you're good to go.

While this is great for compiled plugins, as your build scripts can place the newly compiled plugin there for you, it is not as comfortable when using scripted plugins. Forgetting to copy the latest version, or updating the code in the `plugins` directory instead of your repository can both lead to annoying problems and waste precious time.

Moreover, access to the `plugins` directory requires root access.

The Sark Way

To combat the limitations of IDAs plugin loading mechanism, Sark provides the `plugin_loader.py` plugin. Once installed (in the classic IDA way) it allows you to define plugin-lists - a system-wide list and a user-specific list - to be loaded automatically.

The lists are simple, consisting of full-paths and line-comments:

```
C:\Plugins\my_plugin.py

# This is a comment. Comments are whole lines.
C:\OtherPlugins\another_plugin.py
```

Both lists are named `plugins.list` and are automatically created by IDA as empty lists at the following locations:

System-Wide Under IDA's `cfg` subdirectory. The path can be found using `idaapi.idadir(idaapi.CFG_SUBDIR)`. This list requires root access to modify as it is in IDA's installation directory.

User-Specific Under IDA's user-directory. `$HOME/.idapro` on Linux, `%appdata%/%HexRays/IDA Pro` on Windows. The path can be found using `idaapi.get_user_idadir()`. Each user can set his own plugins to load, thus eliminating the need for root access.

To install your plugins, just add them to one of the lists. This allows you to easily update plugins as you go without ever needing to copy them.

2.5.2 Show Meaningful

When reversing an executable, we often need to deal with a large amount of unknown code. To combat this, we usually look for strings and library functions, and use them as guides as we interpret the code. When those are ample,

we hardly need to look at the assembly code to infer meaning. On the other hand, the need to constantly jump into functions, pan them around to see all the strings, then jump back out is quite time consuming and confusing.

Well, no more!

The "Meaningful" plugin allows you to get all the information you need with a simple hotkey.

Usage

Whenever inside a function, just press `Alt + 0` to get a table of all the meaningful objects in it:

```
Meaningful References in 'py_data_format_to_py_dict' : 0x100DE0A0
Type     Usage        Address       Object
-----------------------------------------
code     0x100DE0B2   0x1068E7F0    __imp__PyGILState_GetThisThreadState
str      0x100DE0C7   0x10459654    'idaapi.cpp'
str      0x100DE0CC   0x10459660    "*** WARNING: Code at %s:%d should have the GIL, but apparently doesn't ***\n"
code     0x100DE0D1   0x10002045    int __cdecl msg(char const *,...)
str      0x100DE13F   0x104576C4    'hotkey'
str      0x100DE148   0x104576B8    'menu_name'
str      0x100DE151   0x1045769C    'name'
str      0x100DE15D   0x104576EC    'value_size'
str      0x100DE169   0x10457724    'text_width'
str      0x100DE174   0x10457740    'cbsize'
str      0x100DE180   0x10457694    'props'
str      0x100DE185   0x104596AC    '{s:i,s:i,s:i,s:k,s:s,s:s,s:s}'
code     0x100DE18A   0x1068E854    __imp__Py_BuildValue

Python
```

Since the output is at the *Output window*, a double click on an address will take you to it.

To make things even more agile, you can press `Ctrl + Alt + 0` whenever on a line referencing a function to get the values displayed for that function:
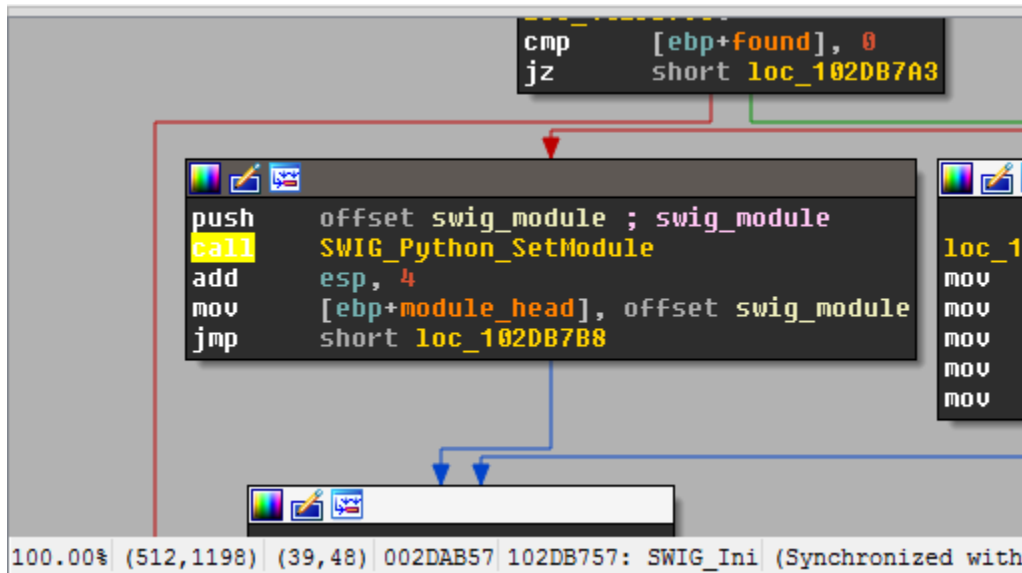
```
Meaningful References in 'SWIG_InitializeModule' : 0x102DB700
Type      Usage           Address         Object
------------------------------------------------
code      0x102DB741      0x102DB960      SWIG_Python_GetModule
code      0x102DB757      0x102DB9B0      SWIG_Python_SetModule
code      0x102DB812      0x1001EF70      SWIG_MangledTypeQueryModule
code      0x102DB89A      0x1001EF70      SWIG_MangledTypeQueryModule
code      0x102DB8D7      0x10026E90      SWIG_TypeCheck
```

[Python]

2.5.3 Quick Copy

The quick copy allows you to easily copy data from the IDB.

Usage

Copying Addresses

To copy the address of the current line, just press `Ctrl + Alt + C`. It will get copied as a hex number, prefixed with `0x`.

To copy the file offset of the current line, simply press `Ctrl + Alt + C + O`. It will get copied as a hex number (like line address copying), prefixed with `0x`.

Copying Bytes

Pressing `Ctrl + Shift + C` copies the bytes of the current line or selection.

Copying the current line:

```
.text:100DE0FC                  ; -------------------------------
.text:100DE0FC
.text:100DE0FC                  loc_100DE0FC:                    ; CODE XREF: py_data
.text:100DE0FC 8B 4D 08                         mov     ecx, [ebp+arg_0]
.text:100DE0FF 8B 51 14                         mov     edx, [ecx+14h]
.text:100DE102 89 55 FC                         mov     [ebp+var_4], edx
.text:100DE105

000DD4FC 100DE0FC: py_data_format_to_py_dict:loc_100DE0FC (Synchronized with Hex View-1)
```

Will result in `8b 4d 08`, while copying a selection:

Will result in `8b 4d 08 8b 51 14`.

2.5.4 Autostruct

Creating and applying structs in IDA can be quite a hassle.

Go to the structure window, create a structure, define members at specific offsets, go to the disassembly to apply them, go back to the structure view to correct errors, apply other members… And on and on it goes.

The Autostruct plugin does all this work for you, without ever having to leave the IDA view. It automatically creates structs, defines member offsets, and applies them in the disassembly view.

Usage

1. Select a line (or lines) containing struct references

2. Press `Shift + T`

3. Set the struct name

4. Choose the register holding the struct base for the selected code. Autostruct will automatically suggest the most likely candidate in the selection.

   At this point, Autostruct will try and create a new struct, populate it with relevant offsets, and apply it to the selection.

5. If a structure of the given name already exists, you will need to select whether to modify the existing structure, apply without modification, or cancel.
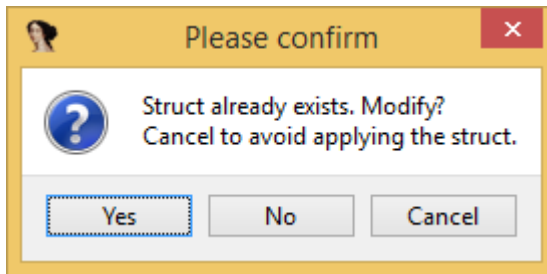
6. Struct creation and modification happen seamlessly

Known Issues

Misaligned Member Creation When attempting to create a member at an offset belonging to another member, Autostruct will fail. This usually happens when a previous definition was incorrect (wrong member size) or when
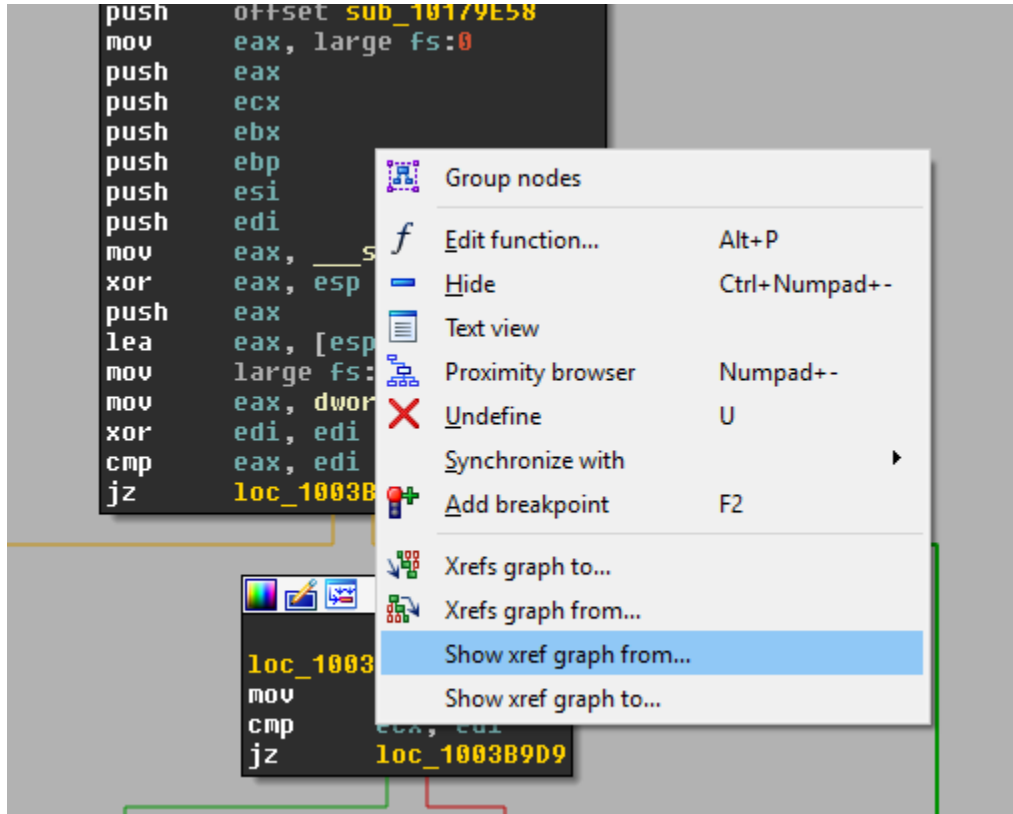
members are unions. At this point, manual handling (redefining the large member as a smaller one) is required.
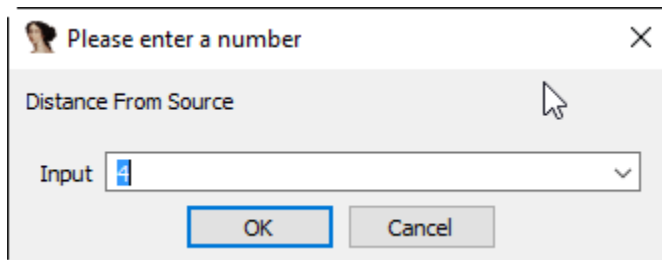
2.5.5 Xrefs Graph

The Xrefs-Graph is used to easily generate interactive xref graphs.

Usage

Anywhere within the IDA-View, just right-click[1], and select the desired option:
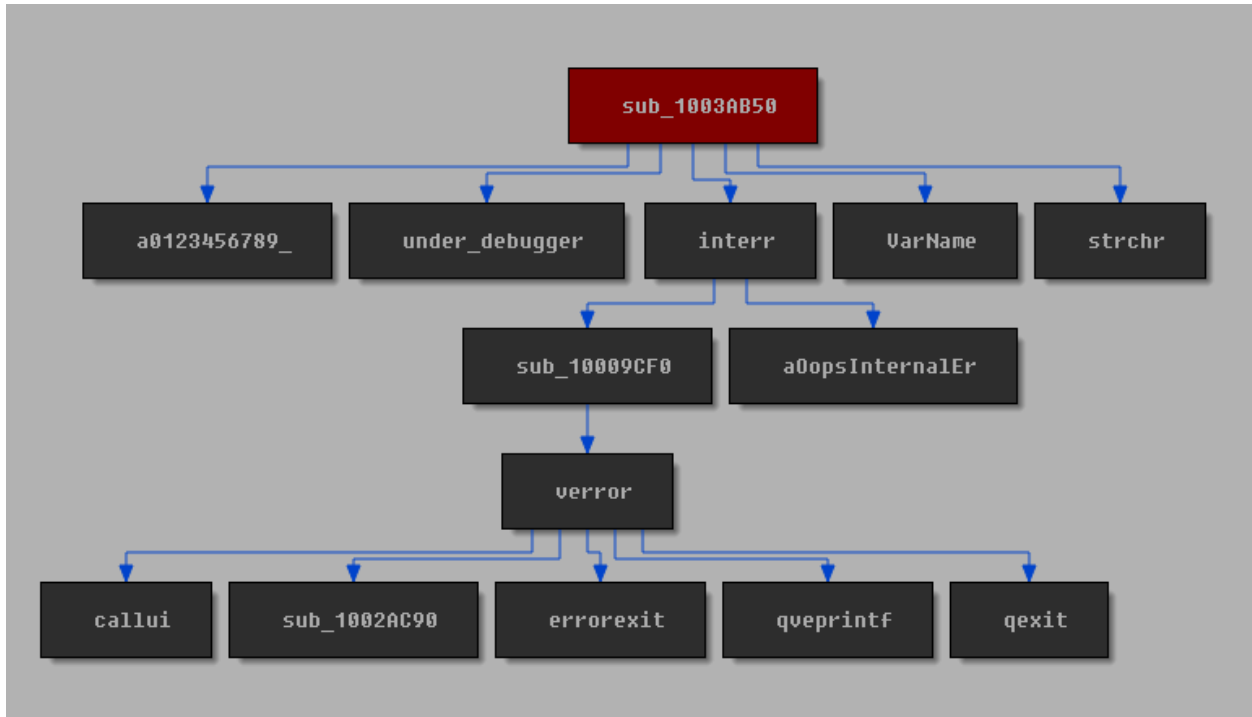

In the popup dialog, enter the distance (recursion level) desired from the source:

Once you press OK, the plugin will generate an interactive xrefs graph:

---

[1] In IDA 6.6 or earlier, use `View/Graph/Xrefs from source` or `View/Graph/Xrefs to source`, as context menus cannot be augmented.

A double-click on any block will take you to the relevant address. Also, names in the blocks will be updated as you rename functions.

Known Issues

Node Groups While creation of node groups *is* possible via IDA's GUI, it is not presently supported in the plugin. Creation of node groups may cause unpredictable errors.

2.6 Debugging IDAPython Scripts

While IDAPython is extremely useful, it can be a bit of a hassle to debug IDA Pro plugins. This tutorial will give get you started on debugging IDAPython scripts and plugins using Python Tools for Visual Studio.

And yes, it is completely free.

2.6.1 The Setup

For this tutorial, we will be using the following software:

1. IDA Pro 6.8
2. Visual Studio Community
3. Python Tools for Visual Studio, documentation can be found here.
4. Python's `ptvsd` module. Install using `pip install ptvsd`.
5. The following plugin:

```python
# filename: ptvsd_enable.py

import idaapi
import ptvsd

try:
    # Enable the debugger. Raises exception if called more than once.
    ptvsd.enable_attach(secret="IDA")
except:
    pass


class DebugPlugin(idaapi.plugin_t):
    flags = idaapi.PLUGIN_FIX
    comment = "PTVSD Debug Enable"
    help = "Enable debugging using PTVSD"
    wanted_name = "PTVSD"
    wanted_hotkey = ""

    def init(self):
        return idaapi.PLUGIN_KEEP

    def term(self):
        pass

    def run(self, arg):
        pass


def PLUGIN_ENTRY():
    return DebugPlugin()
```

For the purposes of this tutorial, you can try and debug this plugin:

```python
# filename: sample_debuggee.py

import idaapi


def my_debugged_function():
    # Set breakpoint here!
    pass


class SamplePlugin(idaapi.plugin_t):
    flags = idaapi.PLUGIN_PROC
    comment = "Sample Debuggee"
    help = "Sample Debuggee"
    wanted_name = "Sample Debuggee"
    wanted_hotkey = "Shift+D"

    def init(self):
        return idaapi.PLUGIN_KEEP

    def term(self):
        pass
```

```python
    def run(self, arg):
        my_debugged_function()


def PLUGIN_ENTRY():
    return SamplePlugin()
```
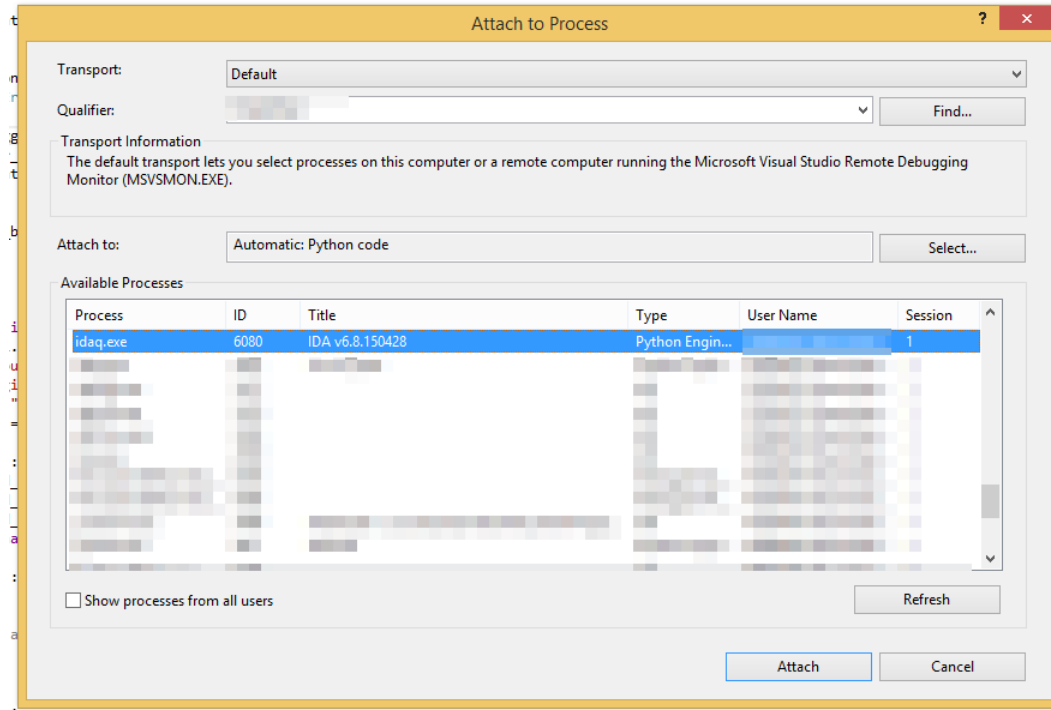
2.6.2 Debugging

1. Put `ptvsd_enable.py` (provided above) in IDA's plugins directory. If you want to use the sample debuggee, put it in the plugins directory as well.

2. Start IDA and load an IDB (otherwise weird issues arise)

3. Load the code you want to debug into Visual Studio and set breakpoints.

4. In Visual Studio (with the plugin file open), use `DEBUG->Attach to process`
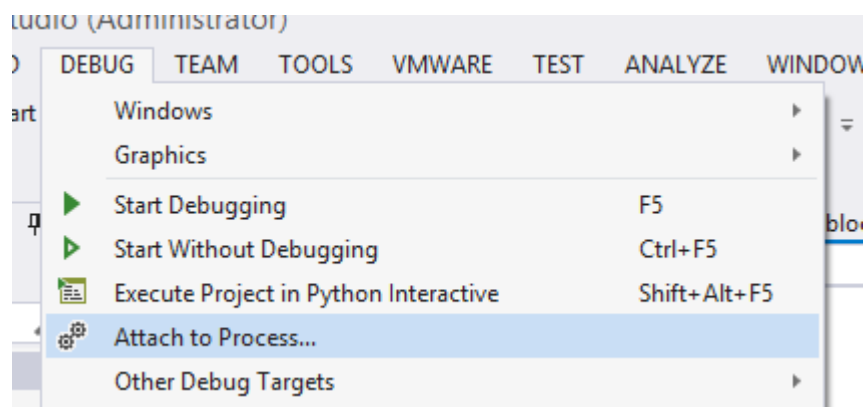
5. In the dialog, select `idaq.exe` and click `Attach`

6. We are now attached. Once a breakpoint is hit, Visual Studio will break and let you debug.

7. Have fun debugging!

2.6.3 Important Notes

1. When debugging (breaking and stepping), IDA will be frozen.

2. Load your IDB prior to attaching the debugger.

3. For easy debug-on-demand, keep `ptvsd_enable.py` in IDA's plugins directory at all times.

4. To set breakpoints, make sure you load into VS the files that are actually loaded by IDA.

If you find any issues with the tutorial, please submit them here.

2.7 How To Contribute

The Sark project was created to provide an intuitive, easy to use scripting layer for IDA Pro. If something seems like the right-way to do something, it should probably be added to Sark.

If you have something you think is worth adding, either submit a new issue or (preferably) create a pull-request.

When contributing, try and follow these guidelines:

- Add yourself to the AUTHORS.rst file in alphabetical order. Every contribution shall be credited.

- Each new feature must have a reproducible test-case or usage. If it can't be used, it will not get in.

- Obey PEP 8 and PEP 257.

- All code should be documented. Usage samples and references to the IDASDK headers are a bonus.

- Write meaningful commit messages.

- No change is too small. You are welcome to fix typos, convention violations, or plain ugly code. All contributions are welcome.

- When submitting a fix to a bug, describe the bug being fixed. Include both the bug and the desired results. Creating an issue for the bug is good practice.

- When reporting bugs, make sure you mention your OS and IDA version.

If you can't adhere to the guidelines, **submit your pull requests anyway**. Maybe someone else can improve on it.

Thanks for contributing!

2.8 Credits

Sark is written and maintained by Tamir Bahar.

2.8.1 Contributors

The following people have contributed directly or indirectly to this project:

- darx0r

- ynvb

- OfficialMan

Please add yourself here alphabetically when you submit your first pull request.