Official IDA Plugins repository

This is the official IDA Plugins repository, where you can find plugins developed by Hex-Rays and third parties.

The following contains a brief list of plugins and their descriptions. For a more detailed information, refer to section two of this document.

Brief list of plugins and their descriptions

Heimdallr - by: RobertNotRob (Interrupt Labs)

Heimdallr is a plugin which allows for deep links into an IDA database. This allows for deeper integration with your notes and collaboration with team mates.

AntiDebugSeeker - by: Takahiro Takeda

Automatically identify and extract potential anti-debugging techniques used by malware. #anti-debugging #malware-analysis

HvcallGUI - by: gerhart_x

Plugin allows to extract Microsoft Hyper-V hypercalls from Windows binaries. See README.md for detailed instruction.

q3vm - by: David Catalán @ Outpost24

Loader and processor modules for the Q3VM virtual machine.

IdaClu - by: Sergejs Harlamovs

IdaClu is a version agnostic IDA Pro plugin for grouping similar functions.

hexagon - by: Willem Hengeveld

IDA Processor module for the qualcomm hexagon\/QDSP6 cpu

tidy - by: Flutiflouille @ Thalium

This plugin automates the organization of functions by categorizing them into folders based on their respective C++ namespaces and classes. #arrange #sort #namespace #class #c++

ida_kmdf - by: Arnaud Gatignol & Julien Staszewski @ Thalium

#kernel #windows #kmdf #wdk #til

ShowComments - by: Fernando Mercês

IDA plugin to show all comments (created by the analyst or automatically added by IDA\/plugins) in a database.

Uncertaintifier - by: ramikg

Add question marks to Hex-Rays local variable names

EWS - by: Anthony Rullier

Emulation based debugger & traces explorer.

IA32 VMX Helper - by: BehroozAbbassi & es3n1n

IDA scripts for #hypervisor (Hyper-v) analysis and reverse engineering automation

Wakatime - by: es3n1n

The open source plugin for productivity metrics, goals, leaderboards, and automatic time tracking.

TDInfo Parser - by: ramikg

Turbo\/Borland debug information parser for IDA

Search from IDA - by: ramikg

Look up highlighted IDA strings using your browser

OpenWithIDA - by: ramikg

Right click ---> "Open with IDA"

idapcode - by: binarly-io

IDA plugin displaying the P-Code for the current function.

iBoot64helper - by: Patroklos Argyroudis

IDAPython loader to help with AArch64 iBoot, iBEC, and SecureROM reverse engineering.

ida-iboot-loader - by: matteyeux

IDA loader for Apple's 64 bits iBoot, SecureROM and AVPBooter

ida_migrator - by: Gilad Reich

IDA Migrator is an IDA Pro plugin which helps migrate existing work from one database instance to another. It Conveniently migrates function names, structures and enums.

AMIE - by: Alexandre Adamski

A Minimalist Instruction Extender for the ARM architecture and IDA Pro

Gepetto - by: Ivan Kwiatkowski

Queries OpenAI's davinci-003 language model to speed up reverse-engineering #gpt3

syms2elf - by: Daniel García

The plugin export the symbols (for the moment only functions) recognized by IDA Pro and radare2 to the ELF symbol table.

HashDB - by: OALABS

Malware string hash lookup plugin for IDA Pro. This plugin connects to the OALABS HashDB Lookup Service.

flare-emu - by: MANDIANT

flare-emu marries a supported binary analysis framework, such as IDA Pro or Radare2, with Unicorn's emulation framework to provide the user with an easy to use and flexible interface for scripting emulation tasks.

FIDL - by: MANDIANT

This is a set of utilities wrapping the decompiler API into something sane. This code focus on vulnerability research and bug hunting.

genmc - by: pat0is

genmc is an IDAPython script\/plugin hybrid that displays Hexrays decompiler microcode, which can help in developing microcode plugins.

abyss - by: pat0is

Augmentation of postprocess Hexrays decompiler output.

IDACyber - by: pat0is

IDACyber is an interactive data visualization plugin for IDA Pro. It consists of external "color filters" that transform raw data bytes into a canvas that can be used to inspect and navigate data interactively.

dsync - by: pat0is

IDAPython plugin that synchronizes disassembler and decompiler views.

HRDevHelper - by: pat0is

This plugin for the HexRays decompiler creates a graph of a decompiled function's AST using IDA's internal graph viewer.

Hexrays Toolbox - by: pat0is

Hexrays Toolbox is a script for the Hexrays Decompiler which can be used to find code patterns within decompiled code.

lumen - by: Naim A.

A private Lumina server that can be used with IDA Pro 7.2+.

ComIDA - by: Airbus CERT

An IDA Plugin that help during the analysis of modules using COM.

Virtuailor - by: 0xgalz

IDAPython tool for C++ vtables reconstruction.

Ghida - by: Cisco Talos

GhIDA is an IDA Pro plugin that integrates the Ghidra decompiler in IDA.

xray - by: pat0is

xray is a plugin for the Hexrays decompiler that both filters and colorizes the textual representation of the decompiler's output based on configurable regular expressions.

mkYARA - by: Fox-IT

mkYARA comes with a IDA plugin to easily create YARA signatures by selecting a set of instructions and choosing one of the mkYARA -> Generate YARA rule options.

VT-IDA-PLUGIN - by: VirusTotal

This plugin integrates functionality from VirusTotal web services into the IDA Pro's user interface.

FindYara - by: OALabs

IDA python plugin to scan binary with yara rules.

golang_loader_assist - by: Tim Strazzere

Making GO reversing easier in IDA Pro

J.A.R.V.I.S - by: Carlos Garcia Prado

A plugin for IDA Pro to assist you with the most common reversing tasks. It integrates with the (J.A.R.V.I.S) tracer.

idenLib - by: Noah.

idenLib (Library Function Identification) plugin for IDA Pro

HexRayPytools - by: Igor Kirillov

Assist in the creation of classes\/structures and detection of virtual tables.

Heap Viewer - by: Daniel García

Used to examine the glibc heap, focused on exploit development.

Driver Buddy - by: NCC Group Plc

It assists with the reverse engineering of Windows kernel drivers.

IDASignSrch - by: Sirmabus

It can recognize tons of compression, multimedia and encryption algorithms and many other things like known strings and anti-debugging code.

Yaco - by: DGA\/MI SSI

Collaborative Reverse-Engineering for IDA.

Diaphora - by: Joxean

It is a program diffing plugin for IDA, similar to Zynamics Bindiff.

IDA ARM Highlight - by: Guillaume

Highlighting and decoding ARM system instructions.

IDA for Delphi - by: Coldzer0

IDA Python Script to Get All function names from Event Constructor (VCL).

IDA Patcher - by: Peter Kacherginsky

It is designed to enhance IDA's ability to patch binary files and memory.

IDATropy - by: Daniel García

It is designed to generate charts of entropy and histograms using the power of idapython and matplotlib.

IDA Sploiter - by: Peter Kacherginsky

An exploit development and vulnerability research plugin.

IDA IPython - by: james91b

An IDA Pro Plugin for embedding an IPython.

IDA Skins - by: zyantific

Plugin providing advanced skinning support for IDA Pro utilizing Qt stylesheets, similar to CSS.

Idadiff - by: Adrien C.

A diffing tool using Machoc Hash.

IDA_EA - by: Joe Darbyshire

A set of exploitation\/reversing aids for IDA.

idaemu - by: czl

Use for emulating code in IDA Pro. It is based on unicorn-engine.

LazyIDA - by: Lays

Add functionalities such as function return removing, converting data, scanning for string vulnerabilities.

HexRaysCodeXplorer - by: Alexander Matrosov, Eugene Rodionov, Rodrigo Branco & Gabriel Barbosa

The Hex-Rays Decompiler plugin for better code navigation in RE process. CodeXplorer automates code REconstruction of C++ applications or modern malware like Stuxnet, Flame, Equation, Animal Farm …

IFL - by: Hasherezade

IFL, the Interactive Functions List It's goal is to provide user-friendly way to navigate between the functions and their references.

IDARef - by: Mikhail Sosonkin

IDA Pro Full Instruction Reference Plugin

VMAttack - by: Anatoli Kalysch

an IDA PRO Plugin which enables the reverse engineer to use additional analysis features designed to counter virtualization-based obfuscation.

AutoRE - by: Aliaksandr Trafimchuk

IDA PRO auto-renaming plugin with tagging support.

sk3wldbg - by: Chris Eagle

the Sk3wlDbg plugin for IDA Pro. It's purpose is to provide a front end for using the Unicorn Engine to emulate machine code that you are viewing with IDA.

Keypatch - by: Nguyen Anh Quynh & Thanh Nguyen

a plugin of IDA Pro for Keystone Assembler Engine. See this introduction for the motivation behind Keypatch, and this slides for how it is implemented.

CGEN - by: Yifan Lu

an extension of CGEN (which is an attempt at modeling CPUs in Scheme and then automatically generating simulators, assemblers, etc) to generate IDA Pro modules.

Labeless - by: Aliaksandr Trafimchuk

Labeless is a plugin system for dynamic, seamless and realtime synchronization between IDA Database and debug backend. It consists of two parts: IDA plugin and debug backend's plugin.

Ponce - by: Alberto Garcia Illera, Francisco Oca

an IDA Pro plugin that provides users the ability to perform taint analysis and symbolic execution over binaries in an easy and intuitive fashion. With Ponce you are one click away from getting all the power from cutting edge symbolic execution. Enti

SimplifyGraph - by: Jay Smith

IDA Pro plugin to assist with complex graphs

RetDec - by: Avast

a RetDec plugin for IDA (provides RetDec-decompiled code views to IDA)

Milan's tools - by: Milan Bohacek

api_palette.py: a code-searching\/completion tool, for IDA APIs member_type.py: automatically sets type to structure members, depending on their name paste_name.py: a handy plugin to rename decompiler-view names to whatever is in the clipboard by simp

lighthouse - by: Markus Gaasedelen

a code coverage plugin for IDA Pro. The plugin leverages IDA as a platform to map, explore, and visualize externally collected code coverage data when symbols or source may not be available for a given binary.

IDABuddy - by: Tamir Bahar

...is a reverse-engineer's best friend. Designed to be everything Clippy the Office Assistant was, and more! IDABuddy will always be there for you. Friendly and helpful while you work. Offering tips and friendly chat. And best of all - since it is op

Drop - by: Thomas Rinsma

an experimental IDA Pro plugin capable of detecting several types of opaque predicates in obfuscated binaries. It leverages the power of the symbolic execution engine angr and its components to reason about the opaqueness of predicates based on their

BinCAT - by: Sarah Zennou, Philippe Biondi, Raphaël Rigo, Xavier Mehrenberger

a static Binary Code Analysis Toolkit, designed to help reverse engineers, directly from IDA.

NIOS2 - by: Anton Dorfman

IDA Pro processor module for Altera Nios II Classic\/Gen2 microprocessor architecture

IDArling - by: Alexandre Adamski and Joffrey Guilbon

a collaborative reverse engineering plugin for IDA Pro and Hex-Rays.

IDA-Minsc - by: Ali Rizvi-Santiago

IDA-minsc is a plugin that extends IDAPython with a DWIM syntax. This wraps various IDA features like names, xrefs, instructions, operands, structs, types, etc. into an api that simplifies annotating or exchanging of artifacts from IDA and Hex-Rays.

IDAFuzzy - by: Ga-Ryo

a fuzzy search tool for IDA Pro. This tool helps you to find command\/function\/struct and so on. This tool is inspired by Mac's Spotlight and Intellij's Search Everywhere dialog.

Hyara - by: Yi Hyun, Kwak Kyoung-Ju

a plugin to create pattern-matching rules.

HeapViewer - by: Daniel García Gutiérrez

an IDA Pro plugin to examine the heap (glibc malloc implementation), focused on exploit development.

ActionScript 3 - by: Boris Larin

… an ActionScript 3 processor module and Flash debugger plugin.

Virtuailtor - by: Gal Zaban

an IDAPython tool for C++ vtables reconstruction on runtime.

Karta - by: Eyal Itkin

an IDA Python plugin that identifies and matches open-sourced libraries in a given binary. The plugin uses a unique technique that enables it to support huge binaries (>200,000 functions), with almost no impact on the overall performance.

idapkg - by: jinmo123

Packages for IDA Pro

ifred - by: jinmo123

IDA command palette & more

findrpc - by: Lucas Georges

Idapython script to carve binary for internal RPC structures

deREferencing - by: Daniel Garcia Gutierrez

IDA Pro plugin that implements more user-friendly register and stack views

BRUTAL IDA - by: Tamir Bahar

BRUTAL IDA restores your original workflow by blocking the undo and redo keyboard shortcuts.

SmartJump - by: Adam Prescott (PwC)

SmartJump is designed to improve the 'g' keyboard shortcut in IDA, especially when using IDA to debug binaries.

Renamer - by: Sharon Brizinov

For each function [Renamer] shows you what are the available strings and let you choose what is the appropriate name for the function.

qiling - by: ChenXu WU, ZiQiao KONG and Qiling.io Team

...an advanced binary emulator with instrumentation plugin for IDA

PETree - by: Tom Bonner (BlackBerry Research and Intelligence Team)

PETree is a Python module for viewing Portable Executable (PE) files in a tree-view

PacXplorer - by: Ouri Lipner (Cellebrite Security Research)

PacXplorer analyzes ARM64 PAC instructions to provide a new type of cross-reference: from call site to virtual function, and vice versa.

Lucid - by: Markus Gaasedelen

Lucid is a developer-oriented IDA Pro plugin for exploring the Hex-Rays microcode.

idapm - by: Taichi Kotake

idapm is IDA Plugin Manager.

ida_medigate - by: Uriel Malin

...C++ plugin for IDA Pro

idahunt - by: Aaron Adams, Cedric Halbronn (NCC Group)

idahunt is a framework to analyze binaries with IDA Pro and hunt for things in IDA Pro. It is a command-line tool to analyze all executable files recursively from a given folder.

grap - by: Aurélien Thierry (QuoSec), Jonathan Thieuleux, Léonard Benedetti

grap is a tool to match binaries at the assembly and control flow level (for instance: a loop on a basic block containing a xor)

FingerMatch - by: Jan Prochazka

FingerMatch is an IDA plugin for collecting functions, data, types and comments from analyzed binaries and fuzzy matching them in another binaries.

efiXplorer - by: efiXplorer team

IDA plugin for UEFI firmware analysis and reverse engineering automation

DynDataResolver - by: Holger Unterbrink (Cisco Talos)

DDR is an IDA plugin that instruments binaries using the DynamoRIO framework.

capa explorer - by: Mike Hunhoff, Moritz Raabe, William Ballenthin, Ana Maria Martinez Gomez

capa explorer is an IDA Pro plugin written in Python that integrates the FLARE team's open-source framework, capa, with IDA. capa is a framework that uses a well-defined collection of rules to identify capabilities in a program.

bip - by: Bruno Pujos, Synacktiv

Bip is a project which aims to simplify the usage of Python for interacting with IDA. Its primary goal is to facilitate the use of Python in the interactive console of IDA and the writing of plugins

bf - by: Milan Boháček

Bf_proc.py adds brainfuck language to the Hex-Rays decompiler.

VulFi - by: Martin Petran (Accenture)

A query based function cross-reference finder for vulnerability research

ttddbg - by: Simon Garrelou, Sylvain Peyrefitte of the Airbus CERT Team

ttddbg is a debugger plugin for IDA Pro which can read Time Travel Debugging traces generated by WinDBG or Visual Studio

Quokka - by: Alexis Challande

Quokka is a binary exporter: from the disassembly of a program, it generates an export file that can be used without a disassembler.

ida_names - by: Pavel Maksyutin (Positive Technologies)

IDA-names automatically renames pseudocode windows with the current function name. It can also rename ANY window with SHIFT-T hotkey.

ida_kcpp - by: Uriel Malin and Ievgen Solodovnykov

An IDAPython module for way more convienent way to Reverse Engineering iOS kernelcaches.

ida_bochs_windows - by: David Reguera Garcia

Helper script for Windows kernel debugging with IDA Pro on native Bochs debugger (including PDB symbols)

FirmLoader - by: Martin Petran (Accenture)

An alternative to SVD loader that uses simpler JSON files

FindFunc - by: Felix B.

[…] an IDA Pro python3 plugin to find\/filter code functions that contain a certain assembly or byte pattern, reference a certain name or string, or conform to various other constraints

Condstanta - by: Martin Petran (Accenture)

Plugin to search for constants used in conditional statements

Yagi - by: Sylvain Peyrefitte (Airbus CERT)

Yagi is a C++ plugin that includes the Ghidra decompiler into IDA 7.5 and 7.6.

wilhelm - by: Xinyu Zhuang

wilhelm is a Python API that provides a better interface for working with Hex-Rays. In particular, I designed it with the IDAPython REPL (aka console) in mind; while it works fine in scripts, it's meant to be used interactively to quickly automate so

Tenet - by: Markus Gaasedelen

Tenet is an IDA Pro plugin which enables reverse engineers to explore execution traces of native code. It demonstrates how visualization can augment time-travel-debugging technologies to create more fluid controls for exploring the execution runtime.

SyncReven - by: Cyrille Bagard

Axion is the main application of the Reven platform, which captures a time slice of a full system execution. It can then be connected to many tools, including IDA Pro, for the analysis. A plugin to synchronize the IDA view from inside Reven already e

RefHunter - by: Jiwon Choi

RefHunter find all references in simple and lightweighted manner. - User-friendly view - Runs without any 3rd-party application - Runs without installing itself, it's just portable. - Analyze the function and show tiny little report for you!

qscripts - by: Elias Bachaalany

QScripts is productivity tool and an alternative to IDA's "Recent scripts" (Alt-F9) and "Execute Scripts" (Shift-F2) facilities.

nmips - by: Leonardo Galli

IDA plugin to enable nanoMIPS processor support. This is not limited to simple disassembly, but fully supports decompilation and even fixes up the stack in certain functions using custom microcode optimizers. It also supports relocations and automati

jside - by: David Zimmer

This plugin comes in two parts. There is a small IPC based server which sits in IDA and allows remote automation, and then there is an external Javascript IDE which supports intellisense, syntax highlighting, and a full on javascript debugger.

IPyIDA - by: Marc-Etienne M.Léveillé

IPyIDA is a python-only solution to add an IPython console to IDA Pro. Use Shift-. to open a window with an embedded Qt console. You can then benefit from IPython's autocompletion, online help, monospaced font input field, graphs, and so on. You can

IDAPatternSearch - by: David Lazar

The IDA Pattern Search plugin adds a capability of finding functions according to bit-patterns into the well-known IDA Pro disassembler based on Ghidra's function patterns format. Using this plugin, it is possible to define new patterns according to

IDA2Obj - by: Mickey Jin

IDA2Obj is a tool to implement SBI (Static Binary Instrumentation).

FunctionInliner - by: Tomer Harpaz

FunctionInliner is an IDA plugin that can be used to ease the reversing of binaries that have been space-optimized with function outlining (e.g. clang -moutline).

D-810 - by: Boris Batteux

D-810 is a plugin which aims at removing several obfuscation layer (including MBA, opaque predicate and control flow flattening) during decompilation. It relies on the Hex-Rays microcode API to perform optimization during the decompilation. The plugi

CTO - by: Hiroshi Suzuki

CTO (Call Tree Overviewer) is an IDA Pro plugin for visualizing function call tree. It can also summarize function information such as internal function calls, API calls, static linked library function calls, unresolved function calls, string referen

CollaRE - by: Martin Petran

CollaRE enables collaboration using multiple reverse engineering tools for more complex projects where one tool cant provide all required features or where each member of the team prefers a different tool to do the job.

Detailed list of plugins and their descriptions

Heimdallr - by: RobertNotRob (Interrupt Labs)

Plugin description

> Heimdallr is a plugin which allows for deep links into an IDA database. This allows for deeper integration with your notes and collaboration with team mates.

readme for Heimdallr

The Heimdallr IDA plugin exposes a localhost gRPC server for each IDA instances which allows the [Heimdallr client](https://github.com/interruptlabs/heimdallr-client) to navigate to locations in IDA.

Installation

Server
1. Install heimdallr-ida with pip ([ensure the pip you are using matches the python environment IDA is using
- Using git directly pip3 install -e git+https://git@github.com/interruptlabs/heimdallr-ida.git#egg=heimdallr_ida
- From the source repo ‘pip3 install -e ./heimdallr-ida/‘
2. Launch IDA and enter the following into the console:

```
import heimdallr
heimdallr.install()
```

3. Relaunch IDA and verify gRPC server sucessfully started up. You should something like the following in the output console:

```
[Heimdallr RPC] Plugin version 0.0.1

Starting server on 127.0.0.1:51278

Wrote {"pid": 36813, "address": "127.0.0.1:51278", "file_name": "example.i64", "file_hash":
```

Client 1. Install heimdallr-client with pip
- You may need poetry depending on your python version pip3 install poetry
- Using git directly pip3 install -e git+https://git@github.com/interruptlabs/heimdallr-client.git#egg=heimdallr_client
- From the source file here pip3 install -e ./heimdallr-client/
2. Verify settings.json has been created in the relevant application directory
- MacOS/Linux - $HOME/.config/heimdallr/
- Windows - %APPDATA%/heimdallr/
3. Modify settings.json to be accurate for your system
- ida_location is the location of your IDA Installation (should be automatically filled)
- idb_path is an array of locations for the heimdallr client to search for corosponding idbs

MacOS
In MacOS we can use AppleScript to act as a stub to forward URI requestst to our python client script. To configure this:

1. Locate path to ‘heimdallr_client‘ script

```
roberts@RobertS-IL-Mac heimdallr_client % whereis heimdallr_client
heimdallr_client: /opt/homebrew/bin/heimdallr_client
```

2. Open ./heimdallr-client/macos-stub/heimdallrd.scpt in Script Editor (double click)

3. Change the heimdallr_client path to be valid for your system
4. Export it as an Application (File -> Export...)
a. Ensure ”File Format” is ”Application”

5. Modify the ‘Info.plist‘ file to add the following text between it’s first set of dictionary tags

```
    CFBundleURLTypes
```

```
CFBundleURLName
IDA URL
CFBundleURLSchemes

    ida
```

(an example Info.plist is in ./macos-stub/Info.plist)

6. Move the Application Bundle to /Applications/ and resign it:

```
codesign --force -s - /Applications/heimdallrd.app
```

7. Run the Application bundle (double click) to register it with the system as a URI handler.

Other Operating Systems For other operating systems a electron app is available to act as a URL handler. You can get it on github: https://github.com/interruptlabs/heimdallr-client/releases/tag/v0.5.2

Usage

You should now be able to open ida:// URIs from anywhere in the system. This could be a Slack DM, a Confluence page, or a Obsidian note. The format is as follows (these):

```
ida://example.i64%3Foffset%3D0x1002315b6%26hash%3Db058de795064344a4074252e15b9fd39%26view%3D
```

These are automatically generated by creating a note in the ‘heimdallr_ida‘ plugin

The search behaviour for a relevant IDB is as follows:
1. Search for an open IDA instance with this database already open
2. Search IDA recently open files for the location of the database
3. Search your ‘idb_path‘ for matching files

The search pattern is used to ensure links can be used easily within a team - so long as you have a database based on the same source file and is named the same.

IDBs are matched by both the database name and source file hash. As such **changing the database name will cause URIs to no longer be valid**.

You can make notes by highlighting the area of text in IDA you want to copy and pressing "Ctrl+Shift+N". The text will be added to a code block with a link back to where it came from and added to your clipboard.

If you want to make a link to share with someone else, pressing "Ctrl+Alt+N", and the link to where you are in IDA will be added.

This currently only works for the Disassembly and Pseudocode views.

AntiDebugSeeker - by: Takahiro Takeda

Plugin description

> Automatically identify and extract potential anti-debugging techniques used by malware. #anti-debugging #malware-analysis

readme for AntiDebugSeeker

The main functionalities of this plugin are as follows:

- Extraction of Windows API that are potentially being used for anti-debugging by the malware
  (All subsequent API represent the Windows API)
- In addition to API, extraction of anti-debugging techniques based on key phrases that serve as triggers, as some anti-debugging methods cannot be comprehensively identified by API calls alone.

Additionally, the file that defines the detection rules is designed to easily add keywords you want to detect.
This allows analysts to easily add new detection rules or make changes.

For packed malware, running this plugin after unpacking and fixing the Import Address Table is more effective.

Please check the detailed information at the following URL.

https://github.com/LAC-Japan/IDA_Plugin_AntiDebugSeeker

HvcallGUI - by: gerhart_x

Plugin description

> Plugin allows to extract Microsoft Hyper-V hypercalls from Windows binaries. See README.md for detailed instruction.

readme for HvcallGUI

HvCallGUI - is utility for automatically extraction of Hyper-V hypercalls names and codes from Hyper-V core binaries:

securekernel.exe
winhvr.sys
winhv.sys
ntoskrnl.exe

additionally can be added for analyzing

securekernella57.exe
ntkrla57.exe

---

q3vm - by: David Catalán @ Outpost24

Plugin description

> Loader and processor modules for the Q3VM virtual machine.

readme for q3vm

Loader and processor modules for the Q3VM virtual machine.

The rhadamanthys-4.5 folder withing the archive's root directory contains a variant of the modules that add support for the modified VM included within recent versions of the malware.

IdaClu - by: Sergejs Harlamovs

Plugin description

IdaClu is a version agnostic IDA Pro plugin for grouping similar functions.

readme for IdaClu

IdaClu

IdaClu is an ***IDA Pro*** plugin with a straightforward 3-step loop pipeline:

1. 1. Find similarities in functions;
2. 2. Label grouped functions in bulk;
3. 3. Repeat step 1 for labeled functions.

The distinctive feature of the plugin lies in ***"how"*** of search and labeling.

IdaClu won ***1st*** place in ***the Hex-Rays Plugin Contest 2023*** !

The Purpose In addition to the plugin ecosystem, the ***IDA*** user community creates a wealth of incredible scripts, varying in complexity. Some could greatly benefit from standardized input/output data format and *GUI* interface. However, due to time constraints, the relative complexity of learning ***PyQt/PySide*** , and the challenge of navigating several backward-incompatible versions of the ***IDAPython API*** many community members are unable to support these efforts. As a result, even the best of us sometimes resort to parsing custom data formats from text files or *output window* of ***IDA*** .

IdaClu serves as a ***tree/table view*** for ***IDAPython*** scripts.

Providing the following features:

- labeling the functions with any combination of *prefix-folder-color*
- filtering the input/output
- using the progress bar indicator
- using the output of other scripts
- providing the output to other scripts

IDAPython -scripts serve as *sub-plugins* for ***IdaClu*** if they follow several conventions of how the script should be structured. One core intention is to minimize this impact.

The plugin comes with a set of *x18* simplistic and handy scripts available out of the box. They form the basis and make ***IdaClu*** useful as a standalone tool. So support of the community is not a "must".

Standard Scripts Scripts are grouped, and while most names are self-explanatory, some may need explanations:

- **Xref Analysis**
  - **Xref Count -** *group by code-xref count*
  - **Xref Source -** *groups: leaf functions, functions with only explicit calls, functions with virtual calls*
  - **Xref Destination -** *groups: independent functions, VFT-functions, library called functions*
- **Constant Analysis**
  - **Common Constants -** *group by constants being referenced*
  - **String Refs -** *group by string refs*
  - **Global Variable Analysis -** *group by refs of defined names*
  - **Lib Usage Analysis -** *group by library function calls*
  - **API Usage Analysis -** *group by API-function calls*
- **Control Flow**
  - **Control Flow Analysis** - *groups: loop-containing funcs, switch-case funcs, recursive funcs*
  - **Pseudocode Size** - *group by pseudocode line count*
- **Code Coverage**
  - **DynamoRIO Functions** - *groups: touched/untouched functions*
- **Function Similarity**
  - **SSDEEP Similarity** - *groups: similar function clusters with* ***ssdeep***
  - **TLSH Similarity** - *groups: similar function clusters with* ***tlsh***
- **Virtual Functions**
  - **Explicit Calls** - *group by function call with arguments shown in comments*
  - **Implicit Calls** - *group by VFT-function call with arguments shown in comments*
- **Filter Analysis**
  - **Distinct Folders -** *group by folder path*
  - **Distinct Prefixes -** *group by prefix name*
  - **Distinct Colors -** *group by highlight color*

User Interface

The user interface intentionally follows ***Miller's Law*** in *UX* - "The immediate memory span of people is limited to approximately seven items, plus or minus two."

Here's the breakdown of the main widgets:

1. 1. ***Toolkit*** - scrollable area with buttons for recognized ***IDAPython*** scripts
2. 2. ***View*** - table-based view for script output, function selection, and rendering current labels
3. 3. ***Filters*** - input controls to refine the chosen script's focus
4. 4. ***Labels*** - name of *prefix* or *folder* for selected functions
5. - ***R*** - toggle button for recursive mode
6. - ***PREFIX*** - interactive label to switch between *FOLDER* and *PREFIX* labeling modes
7. - ***ADD*** - button that applies labeling settings
8. - ***CLEAR*** - button that clears labeling settings
9. 5. ***Palette*** - a set of 5 mutually exclusive color highlighting buttons

Tips

1. 1. **IdaClu** aims to incorporate new features from new ***IDA*** -versions while maintaining *version-agnostic* approach.
   (a) The solution is *graceful degradation* . So the *UI* will adapt to older ***IDA*** versions, excluding unsupported features.
2. 2. Double-clicking any row in the tree view navigates to the corresponding function in ***IDA***
3. 3. The ***filter*** widget toggles collapse with a header click.
4. 4. The ***toolkit*** widget header click swaps ***tree-view*** and ***sidebar*** with places.
5. 5. ***Rename*** context-menu allows to make cutom changes in the selected function name.

Setup ***IdaClu*** is an ***IDAPython*** plugin without external package dependencies. No building is required. It can be installed by downloading the repos-

itory and copying file ***idaclu.py*** and folder ***idaclu*** to your ***IDA Pro*** plugin directory *(either-or)* :

- *C:\Program Files\IDA Pro \plugins*
- *C:\Users\ \AppData\Roaming\Hex-Rays\IDA Pro\plugins*

While the plugin itself doesn't need external packages, some *sub-plugin scripts* might. This won't prevent the plugin from running, but it will gray out the corresponding buttons with an informative tooltip - what's missing.

Two bundled ***Function Similarity*** section scripts require ***py-tlsh*** and ***ssdeep*** . Follow these instructions if you need these scripts:

```
> pip install py-tlsh
# do not: pip install ssdeep
> git clone https://github.com/MacDue/ssdeep-windows-32_64
> cd ssdeep-windows-32_64-master
> python setup.py install
```

Hint For continuous updates, clone the repository with ***Git*** and create a ***symlink*** in the ***IDA*** plugin folder:

Windows

```
:: C:\Users\\AppData\Roaming\Hex-Rays\IDA Pro\plugins\idaclu.py
> mklink "C:\Program Files\IDA Pro X.X\plugins\idaclu.py" \idaclu.py
:: C:\Users\\AppData\Roaming\Hex-Rays\IDA Pro\plugins\idaclu
> mklink /d "C:\Program Files\IDA Pro X.X\plugins\idaclu" \idaclu
```

Mac

```
% ln -s \idaclu.py /Users//.idapro/plugins/idaclu.py
% ln -s \idaclu /Users//.idapro/plugins/idaclu
```

Script Ecosystem The plugin serves as a *GUI* for scripts without one. By following scripting conventions, you can make them compatible with ***IdaClu*** . When done correctly, a corresponding button will appear in the ***sidebar*** .

Script Description The following block is mandatory and is used to register the script in ***IdaClu*** :

```
SCRIPT_NAME = 'Xref Count'  # arbitrary name that will appear on the corresponding button
SCRIPT_TYPE = 'func'  # 'func' or 'custom' depending on whether the script iterates on funct
SCRIPT_VIEW = 'tree'  # 'tree' is the only currently supported view, 'table' is to be added
SCRIPT_ARGS = []  # experimental feature, supports tuples of the form ('', '', '')
```

Main Function In addition to this, each script must define a single ***get_data()*** function.

Currently there are *x2* possible prototypes:

```
# Case #1: SCRIPT_TYPE == 'func':
def get_data(func_gen=None, env_desc=None, plug_params=None):
    # 1. Iterate over pre-filtered functions via func_gen() generator
    # 2. Progress bar values are calculated automatically

# Case #2: SCRIPT_TYPE == 'custom':
def get_data(progress_callback=None, env_desc=None, plug_params=None):
    # 1. Iterate over custom data structures
    # 2. Use `progress_callback(, )` to report current progress
```

Execution Environment If the script logic depends on a specific IDA configuration, the ***IdaClu*** plugin can offer the following properties in the ***env_desc*** object:

```
- feat_bookmarks
- feat_cpp_oop
- feat_folders
- feat_golang
- feat_ida6
- feat_ioi64
- feat_lumina
- feat_microcode
- feat_microcode_new
- feat_python3
```

```
- feat_undo
  ...
```

Before script execution, these fields are guaranteed to be initialized.

You can refer to the full list in the *output window* of ***IDA*** . Right under the ***banner*** there will be ***ENVIRONMENT*** section with the dump of current values.

Custom Input As an experimental feature, *the plugin* supports custom input for each script.

When defined, the following code will render the input field under the script button upon the first click:

```
SCRIPT_ARGS = [('file_path', 'file', 'input the file path')]
```

The second click sends this data to the target script, accessible via the ***plug_params*** parameter of the ***get_data()*** function:

```
plug_params['']
```

Return Value For hierarchical output data, the script should return ***dictionary of lists*** .

Keys of this dictionary are function group names - collapsible elements of *tree view* .

Vals of this dictionary are *either-or* :

- list of function addresses (simple case)
- list of tuples *(,)* (advanced case)

Remarks However the rest is up to the author of the certain script,

there are several optional tips:

1. 1. Each script should be assigned to a certain script group. A script group is essentially a folder - ***plugins/idaclu/plugins/ /*** containing an ***___init___.py*** file with a single string:

   ```
   PLUGIN_GROUP_NAME = ''
   ```

1. 2. For ***IDAPython*** cross-compatibility consider using bundled shims module:

```
from idaclu import ida_shims
```

1. 3. If the script utilizes ***func-generator*** consider employing the following code for debugging and running the script even outside the ***IdaClu*** environment:

```
def debug():
    data_obj = get_data(func_gen=idautils.Functions)
    ida_shims.msg(json.dumps(data_obj, indent=4))
if __name__ == '__main__':
    debug()
```

In case anything is left uncovered in this ***README*** , refer to example-scripts or contact the ***IdaClu*** author.

Compatibility **Recommended specs:** *IDA Pro v8.2+* with *Python v3.x* .

Minimum specs: *IDA Pro v6.7+* with *Python v2.7+* .

Test environment:

1. 1. *IDA Pro v6.7 + Windows 7*
2. 2. *IDA Pro v7.7 + Windows 10*
3. 3. *IDA Pro v8.2 + Windows 11*
4. 4. *IDA Pro v7.0, MacOS High Sierra v10.13.6*
5. 5. *IDA Pro v7.6, MacOS Ventura v13.2.1*

Other ***IDA*** versions can be added as test environments upon request.

To be cross-compatible ***IdaClu*** relies on ***PyQt*** and ***IdaPython*** shims.

Scripts bundled with ***IdaClu*** currently are ***PE-first*** and ***Intel x86/x64-first*** due to the author's work specifics.

Upcoming Changes

- table view for the scripts that output non-hierarchical data
- several types of controls to provide input to scripts
- lots of code optimizations
- sort data in columns by clicking the column headers for tree/table view
- a more flexible API for the scripts
- more detailed IDA environment detection
- helper context menus for native IDA widgets
- filter refresh if changes to .idb were made outside the plugin
- grayed out narrow folder filters if they are subsumed by broader ones
- "save to file" option for the current view
- time estimation for large binaries
- caption of the processing phase in *UI*
- breakpoint setting feature
- count of functions currently selected
- folder labels with hierarchy
- button to clear all labels at once

Feel free to come up with your ideas/proposals.

They will be carefully considered and their implementation is highly likely to be included in upcoming plugin releases.

Known bugs

- Minor issues with cross-compatibility
- Comment column remains unhidden with empty comments
- Some bundled plugins produce quite dirty output:
  - *Global Variable Analysis*
- Some scripts and their corresponding groups do not match
- Experimental "Code Coverage" plugin fails to load
- Explicit folder filtering with '"/"' as a parameter does not work as expected
- Labels are still present in filters after being removed from .idb
- In recursive mode the function can be prefixed multiple times
- Recursive mode is not considered while highlighting with color

Acknowledgments - ***Caroline 'By0ute' Beyne*** for the simplistic ***PoC of collapsible PyQt widget***

- ***Markus 'gaasedelen' Gaasedelen*** for ***DrCov file format parser***

- ***Willi Ballenthin*** of *Mandiant* for ***PoC of PyQt-shim script*** to support both **PySide (IDA and *PyQt (IDA>=v6.9)***

- Guys from ***Gray Hat Academy*** for ***PoC of IDAPython-shim script*** to support ***v6.x-v7.x*** versions of ***IDAPython API***

Version History - ***2023-10-2x*** - Release of more stable *v1.0 (UPCOMING)*

- ***2023-09-14*** - The original release of ***IdaClu*** *v0.9* !

---

hexagon - by: Willem Hengeveld

Plugin description

IDA Processor module for the qualcomm hexagon\/QDSP6 cpu

readme for hexagon

IDA Processor module for the qualcomm hexagon/QDSP6 cpu

This processor module disassembles code found in most qualcomm baseband (gsm modem) chips. It is a wrapper around a gnu disassembler from 2012, and includes all the bugs present in that disassembler.

For more info, see https://github.com/gsmk/hexagon

included in the archive are:

- the source for the plugin

- binaries for windows, linux and intel-mac.

- the patched binutils sources from sourcery.

---

tidy - by: Flutiflouille @ Thalium

Plugin description

> This plugin automates the organization of functions by categorizing them into folders based on their respective C++ namespaces and classes. #arrange #sort #namespace #class #c++

readme for tidy

Tidy plugin significantly enhances the usability of IDA Pro when analyzing C++ binaries, providing a more structured and efficient approach to reverse engineering. By automatically categorizing functions into namespaces, it simplifies the understanding of complex codebases and accelerates the analysis process.

---

ida_kmdf - by: Arnaud Gatignol & Julien Staszewski @ Thalium

Plugin description

> #kernel #windows #kmdf #wdk #til

readme for ida_kmdf

This plugin provides two main features:

- the .til (type library) file generation based on the WDK headers published by Microsoft
- the application of the types previously generated all along the IDB of a KMDF driver

This plugin helps the reverser and offers some confort at the beginning of an analysis (particularly on the preliminary steps).

After the generation of the til files, no specific actions is required to use the plugin, it is executed automatically during the IDB creation.

ShowComments - by: Fernando Mercês

Plugin description

> IDA plugin to show all comments (created by the analyst or automatically added by IDA\/plugins) in a database.

readme for ShowComments

Press Ctrl+Alt+C to show a convenient window with all comments in a database. **ShowComments** gives you the comments' addresses, their type (regular or repeatable), and the function name these addresses belong to.

Uncertaintifier - by: ramikg

Plugin description

> Add question marks to Hex-Rays local variable names

readme for Uncertaintifier

Uncertaintifier

Celebrating uncertainty as an essential part of life, this IDA plugin lets you uncertaintify (or, add trailing question marks to) the names of local variables in the pseudocode view.

FAQ

How to install? Simply put the Python file in `%APPDATA%\Hex-Rays\IDA Pro\plugins`. Both Python 2 and 3 are supported.

How do I register just the plugin's hotkeys, without the context menu items? In the plugin's source code, change the value of `REGISTER_CONTEXT_MENU_ITEMS` to `False`.

Why does this plugin exist? Without delving too deep into the philosophical niche of plugin existentialism, this plugin has been written because starting with version 7.5, Hex-Rays will complain if a local variable name is "not a valid C identifier", thus prohibiting names containing question marks.

Inspired by a friend's idea, *Uncertaintifier* amends this situation.

EWS - by: Anthony Rullier

Plugin description

> Emulation based debugger & traces explorer.

readme for EWS

EWS (*Emulator Wrapper Solution*) is a IDA PRO plugin that aims to integrate emulation features (such as debugger) from various emulators (currently unicorn, but you can add more).

Writting such plugin was motivated by reversing on x64 machine various embedded binaries from Android native libs to automotive firmwares. "Click ready" trace generator and basic explorer is a gain of time.

Key features are:

1. Support Raw and ELF file. PE is experimental, no support for Mach-O.
2. Automatically loads binary inside the emulator based on IDB information.
3. Debugger view with registers' values for each executed instruction.
4. Debugger capacities such as watchpoints, run / steps the code.
5. Stub mechanism to emulate imported functions.

Usage and examples : https://github.com/deadeert/EWS

IA32 VMX Helper - by: BehroozAbbassi & es3n1n

Plugin description

> IDA scripts for #hypervisor (Hyper-v) analysis and reverse engineering automation

readme for IA32 VMX Helper

IA32 VMX Helper

`It's an IDA script (Updated IA32 MSR Decoder) which helps you to find and decode all MSR/VMC`

Features

```
- Add lots of MSR/VMCS Symbolic constant to IDA
- Find and Decode IA32 MSR/VMCS values
- Highlighting MSR/VMCS values and related instructions, such as `rdmsr` or `vmread` by prop
- Just search and explore all founded values
```

Wakatime - by: es3n1n

Plugin description

> The open source plugin for productivity metrics, goals, leaderboards,
> and automatic time tracking.

readme for Wakatime

```
## ida-wakatime-py

### What is this?
[WakaTime](https://wakatime.com/) integration for [IDA Pro](https://hex-rays.com/)

### Installation:
1. Register at [WakaTime](https://wakatime.com) and copy your [API Key](https://wakatime.com
2. Download this repo
3. Extract `wakatime.py` to the directory `$(IDA_PATH)/plugins`
4. Start IDA Pro
5. Enter your API Key
6. That's pretty much it.

### Tested on:
- [x] v7.7 SP 1
- [x] v7.7
- [x] v7.5
- [x] v7.2

_Please help me in testing this plugin on other versions and open a pull request_

### Screenshot:


### Troubleshooting:
```

```
1. If by any chance on the first run ida doesn't ask you for your api key and **there are no

that means that you've used wakatime plugins before and your apikey was already set in `~\.w

If you're reaching an unknown error you are free to open an issue.

### Thanks to:
[wakatime/sublime-wakatime](https://github.com/wakatime/sublime-wakatime) - Pretty much ever
[williballenthin/ida-netnode](https://github.com/williballenthin/ida-netnode) - `Netnode` cl
```

TDInfo Parser - by: ramikg

Plugin description

Turbo\/Borland debug information parser for IDA

readme for TDInfo Parser

TDInfo Parser for IDA

An IDAPython script which parses Turbo/Borland symbolic debug information (aka TDInfo) and imports it into your IDA database.

Inspired by the pwnable.kr challenge *dos4fun* .
Written with DOS in mind.

Requirements IDA 7.0+ & Python 2/3 `pip install -Ur requirements.txt`

Components _tdinfo *structs.py* : *construct* definitions of the different TDInfo structs.
This file is independent of IDA, and may be utilized to import the symbols into other programs which support Python. _tdinfo *parser.py* : An IDAPython script which parses the executable (using said definitions) and imports its symbols into the IDA database.

Usage Consider applying FLIRT signatures (see below). Press *Alt+F7* to load _tdinfo *parser.py* into IDA. Call `TdinfoParser().apply()` .

Some Turbo Tips I'm usually not one for documentations going out of scope, but hey, if you're reading this then you're probably already plucking forgotten pieces of information from arcane niches of the Internet; why shouldn't I chip in?

The following pointers may help get you near your goal:

IDA's entry point analysis may fail to recognize that an executable was compiled with a Turbo/Borland compiler.
You can still import IDA's TCC/BCC symbols manually by loading the compiler's signature file (*File -> Load file -> FLIRT signature file...*).

TD (*Turbo Debugger* for DOS) – apart from being a decent debugger – automatically parses TDInfo symbols.

TDump (Turbo Dump) may be used for a more complete parsing of the debug information.

Search from IDA - by: ramikg

Plugin description

> Look up highlighted IDA strings using your browser

readme for Search from IDA

Search from IDA

Highlight a string in IDA and look it up using your search engine of choice.

The idea is taken from a plugin by Intezer Labs, which I've decided to rewrite from scratch.

FAQ

How to install? Simply put `search_from_ida.py` and the `icons` directory in the user plugins directory:

OS	User plugins directory
Windows	`%APPDATA%\Hex-Rays\IDA Pro\plugins`
Linux/Mac	`~/.idapro/plugins`

Both Python 2 and 3 are supported.

How to change the search engine? In the plugin's source code, change the value of `QUERY_URL_FORMAT` .
Optionally, add an icon to the `icons` directory.

OpenWithIDA - by: ramikg

Plugin description

Right click ---> ”Open with IDA”

readme for OpenWithIDA

OpenWithIDA

Add ”Open with IDA” to your Windows context menu.

OpenWithIDA efficiently determines a file’s bitness, and proceeds to open it using the correct (32-bit or 64-bit) variant of IDA.

Installation

```
pip install openwithida
```

You should now have *OpenWithIDA* installed using the latest IDA version found on your PC.
If not automatically found, you will be prompted to choose your IDA folder.

FAQ

I’ve upgraded IDA. How to make *OpenWithIDA* point to the newer version?

If you’ve completely uninstalled the previous version of IDA, the upgrade should be picked up automatically the next time you click ”Open with IDA”.

If the old version still exists, simply run

```
pip install --force-reinstall openwithida
```

The context menu item wasn’t installed

To find out the cause for the error, run `pip install` with the `-v` flag.

Alternatively, run the installer manually (see below).

How to run the installer manually?

Manually invoking the installer offers the following additional options:

- Installing using a custom path for IDA or Python

- Installing as an extended verb (meaning you have to hold Shift to display it)
- Uninstalling the context menu extension

For usage information, run

```
python installer.py --help
```

(Even when `pip install` fails to install the context menu extension, `installer.py` should be available in the package's installation folder.)

---

idapcode - by: binarly-io

Plugin description

IDA plugin displaying the P-Code for the current function.

readme for idapcode

IDA P-Code

IDA plugin displaying the P-Code for the current function

Usage

Method	Action
From menu	Edit -> Plugins -> IDA P-Code
With hotkey	Ctrl+Alt+S
As IDAPython script	File -> Script file... -> idapcode.py

Requirements pypcode

---

iBoot64helper - by: Patroklos Argyroudis

Plugin description

IDAPython loader to help with AArch64 iBoot, iBEC, and SecureROM reverse engineering.

readme for iBoot64helper

```
iBoot64helper is now an IDA loader!

Just copy iBoot64helper.py to your ~/.idapro/loaders/ (or your IDA/loaders/)
directory, launch IDA, and open a decrypted iBoot, iBEC, or SecureROM binary image.

iBoot64helper is a utility to help with iBoot and SecureROM reverse
engineering. It a) locates the image's proper loading address, b) rebases
the image, c) identifies functions based on common AArch64 function prologues, and
d) finds and renames some interesting functions.

I will be adding features to it, identifying more functions, etc.

For decrypting images you should use xerub's img4lib; the ultimate IMG4 utility.

If you have a device vulnerable to axi0mX's checkm8 you can use "./ipwndfu --dump-rom"
to get a dump of the SecureROM image from your device and use it with iBoot64helper.
```

---

ida-iboot-loader - by: matteyeux

Plugin description

IDA loader for Apple's 64 bits iBoot, SecureROM and AVPBooter

readme for ida-iboot-loader

IDA iBoot Loader

IDA loader for Apple's iBoot, SecureROM and AVPBooter.

Support

This loader supports IDA 7.5 to IDA 8.2 and works on all Apple ARM64 bootloaders even M1+.

Installation

Copy ‘iboot-loader.py‘ to the loaders folder in IDA user directory :

- Windows: ‘%APPDATA%/Hex-Rays/IDA Pro/loaders‘
- Linux and MacOS: ‘$HOME/.idapro/loaders‘

Usage

Open a decrypted 64 bits iBoot image or a SecureROM file with IDA. IDA should ask to open the file with this loader.

Credits - This code is based on argp’s iBoot64helper - iBoot-Binja-Loader

---

ida_migrator - by: Gilad Reich

Plugin description

> IDA Migrator is an IDA Pro plugin which helps migrate existing work from one database instance to another. It Conveniently migrates function names, structures and enums.

readme for ida_migrator

giladreich / ida_migrator Public

- Code
- Issues 2
- Pull requests
- Actions
- Projects
- Wiki
- Security
- Insights
- Settings

master

ida_migrator / docs / README.md

giladreich Update directory structure and update readme.

1 contributor

70 lines (35 sloc) 3.03 KB

IDA Migrator Plugin

IDA Migrator plugin aids migrating function names, structures and enums from one database instance to another.

This comes in handy when:

- Moving to a newer version of IDA that does better analysis and you don't want to change in the new instance type information or variable names of the decompiled functions.
- The current idb instance fails to decompile a function or the decompilation looks wrong in comparison to another idb instance of the same binary.
- Experimenting on another idb instance before making major changes on the current instance.
- A lightweight easy way of creating small incremental backups from the current work.
- For w/e reason, the current idb instance you're working on gets corrupted.

IDA Migrator plugin developed using PyQt, hence should work on all platforms.

Getting Started

Download links can be found here .

Copy the files under the `source` directory and put them under your IDA installation `plugins` directory.

Start your current IDA instance you want to migrate from and then press `CTRL+SHIFT+D` to show the plugin's UI. Alternative; open it through the `Edit -> Plugins -> IDA Migrator` menu:

Step 1 - Exporting Data

Clicking the `Export` button will show all functions of the current database instance:

Hint: You can uncheck any functions you want to exclude from exporting.

Once you click the `Start Export` button, it will ask you where would you like to export the files; One is the `*symbols*.json` storing addresses and function names and the other is `*types*.idc` having all the structures and enums information:

Step 2 - Importing Data

In the new idb instance, open the plugin again and click on the `Import` button, which will then ask you to provide the `*symbols*.json` file:

Same procedure from here, just that once you click the `Start Import` button, it will ask you if you would like to import structures and enums as well from the exported `*types*.idc` file, that's optional for you to choose.

Note that it will only rename functions that does not have the same name and will output what functions has been affected in IDA's console:

Contributing

Pull-Requests are greatly appreciated should you like to contribute to the project.

Same goes for opening issues; if you have any suggestions, feedback or you found any bugs, please do not hesitate to open an issue .

Authors

- Gilad Reich - *Initial work* - giladreich

See also the list of contributors who participated in this project.

License

This project is licensed under the MIT License - see the LICENSE file for details.

AMIE - by: Alexandre Adamski

Plugin description

> A Minimalist Instruction Extender for the ARM architecture and IDA Pro

readme for AMIE

AMIE

A M inimalist **I** nstruction **E** xtender

AMIE is a Python rework of FRIEND that focuses solely on the ARM architecture (only AArch32 and AArch64 are supported). It is both lightweight and dependency-free, and provides the most relevant and up-to-date information about the ARM system registers and instructions.

Features

Improved processor modules

For `MCR/MRC` and `MCRR/MRCC` instructions on AArch32, and for `MSR/MRS` and `SYS` instructions on AArch64, the system register encoding is detected and replaced by its user-friendly name in the *IDA View* subview.

For `MCR/MRC` and `MSR/MRS` instructions, it also applies to the *Pseudocode* subview.

Hints for instructions and registers

Hovering over a system register in the *IDA View* subview or in the *Pseudocode* subview will display a summary (usually kept under 30 lines) of the relevant

documentation page, including the bitfield when available.

Hovering over an instruction mnemonic in the *IDA View* subview or in the *Pseudocode* subview will also display a summary of the relevant documentation page, and the relevant assembly template when available.

Auto-generated resource files

The biggest difference with FRIEND is that the resource files (`aarch32.json` and `aarch64.json`) are auto-generated from the Exploration Tools . The system registers and instructions (documentation and encodings) are extracted by a home-made script that parses the ARM-provided XML files.

Installation

Copy the plugin file `amie.py` , and its resource files `aarch32.json` and `aarch64.json` to your plugins directory or your user plugins directory (if you want to share it between multiple IDA Pro versions). These are the default paths:

OS	Plugins Directory	User Plugins Directory
Windows	`%PROGRAMFILES%\IDA 7.4\plugins`	`%APPDATA%\Hex-Rays\`
Linux	`~/ida-7.4/plugins`	`~/.idapro/plugins`
macOS	`/Applications/IDA Pro 7.4/idabin/plugins`	`~/.idapro/plugins`

Dependencies

There are no dependencies! :-)

Improvements

Support for implementation-defined system registers is not available yet.

There is no Hex-Rays support for `MCRR/MRRC` as this is an IDA Pro limitation.

Credits

- alexhude for creating the FRIEND plugin;
- gdelugre for creating the ida-arm-system-highlight script;
- The good folks at ARM for releasing the Exploration Tools ;
- patateqbool and 0xpanda for testing the plugin and reporting bugs;
- Quarkslab for allowing this release.

Gepetto - by: Ivan Kwiatkowski

Plugin description

> Queries OpenAI's davinci-003 language model to speed up reverse-engineering #gpt3

readme for Gepetto

Gepetto is a Python script which uses OpenAI's davinci-003 model to provide meaning to functions decompiled by IDA Pro. At the moment, it can ask davinci-003 to explain what a function does, and to automatically rename its variables. Here is a simple example of what results it can provide in mere seconds:

---

syms2elf - by: Daniel García

Plugin description

> The plugin export the symbols (for the moment only functions) recognized by IDA Pro and radare2 to the ELF symbol table.

readme for syms2elf

syms2elf

The plugin export the symbols (for the moment only functions) recognized by IDA Pro and radare2 to the ELF symbol table. This allows us to use the power of IDA/r2 in recognizing functions (analysis, FLIRT signatures, manual creation, renaming, etc), but not be limited to the exclusive use of this tools.

Supports 32 and 64-bits file format.

INSTALLATION

- IDA Pro : Simply, copy `syms2elf.py` to the IDA's plugins folder.
- radare2 : You can install via r2pm: `r2pm -i syms2elf`

EXAMPLE

Based on a full-stripped ELF:

```
$ file test1_x86_stripped
test1_x86_stripped: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically li
```

Rename some functions in IDA or r2, run `syms2elf` and select the output file.

After that:

```
$ file test1_x86_unstripped
test1_x86_unstripped: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
```

Now, you can open it with others tools and analyzing in a more comfortable way.

AUTHORS

- Daniel García (@danigargu)
- Jesús Olmos (@sha0coder)

CONTACT

Any comment or request will be highly appreciated :-)

HashDB - by: OALABS

Plugin description

> Malware string hash lookup plugin for IDA Pro. This plugin connects to the OALABS HashDB Lookup Service.

readme for HashDB

HashDB IDA Plugin

Malware string hash lookup plugin for IDA Pro. This plugin connects to the OALABS HashDB Lookup Service .

Adding New Hash Algorithms

The hash algorithm database is open source and new algorithms can be added on GitHub here . Pull requests are mostly automated and as long as our automated tests pass the new algorithm will be usable on HashDB within minutes.

Using HashDB

HashDB can be used to look up strings that have been hashed in malware by right-clicking on the hash constant in the IDA disassembly view and launching the `HashDB Lookup` client.

Settings

Before the plugin can be used to look up hashes the HashDB settings must be configured. The settings window can be launched from the plugins menu `Edit->Plugins->HashDB` .

Hash Algorithms Click `Refresh Algorithms` to pull a list of supported hash algorithms from the HashDB API, then select the algorithm used in the malware you are analyzing.

Optional XOR There is also an option to enable XOR with each hash value as this is a common technique used by malware authors to further obfuscate hashes.

API URL The default API URL for the HashDB Lookup Service is `https://hashdb.openanalysis.net/` . If you are using your own internal server this URL can be changed to point to your server.

Enum Name When a new hash is identified by HashDB the hash and its associated string are added to an enum in IDA. This enum can then be used to convert hash constants in IDA to their corresponding enum name. The enum name is configurable from the settings in the event that there is a conflict with an existing enum.

Hash Lookup

Once the plugin settings have been configured you can right-click on any constant in the IDA disassembly window and look up the constant as a hash. The right-click also provides a quick way to set the XOR value if needed.

Bulk Import

If a hash is part of a module a prompt will ask if you want to import all the hashes from that module. This is a quick way to pull hashes in bulk. For example, if one of the hashes identified is `Sleep` from the `kernel32` module, HashDB can then pull all the hashed exports from `kernel32`.

Algorithm Search

HashDB also includes a basic algorithm search that will attempt to identify the hash algorithm based on a hash value. The search will return all algorithms that contain the hash value, it is up to the analyst to decide which (if any) algorithm is correct. To use this functionality right-click on the hash constant and select `HashDB Hunt Algorithm`.

All algorithms that contain this hash will be displayed in a chooser box. The chooser box can be used to directly select the algorithm for HashDB to use. If `Cancel` is selected no algorithm will be selected.

Dynamic Import Address Table Hash Scanning

Instead of resolving API hashes individually (inline in code) some malware developers will create a block of import hashes in memory. These hashes are then all resolved within a single function creating a dynamic import address table which is later referenced in the code. In these scenarios the HashDB Scan IAT function can be used.

Simply select the import hash block, right-click and choose `HashDB Scan IAT`. HashDB will attempt to resolve each individual integer type (`DWORD/QWORD`) in the selected range.

Installing HashDB

Before using the plugin you must install the python `requests` module in your IDA environment. The simplest way to do this is to use pip from a shell outside of IDA.

```
pip install requests
```

Once you have the requests module installed simply copy the latest release of `hashdb.py` into your IDA plugins directory and you are ready to start looking up hashes!

Compatibility Issues

The HashDB plugin has been developed for use with the IDA 7+ and Python 3 it is not backwards compatible.

---

flare-emu - by: MANDIANT

Plugin description

> flare-emu marries a supported binary analysis framework, such as IDA Pro or Radare2, with Unicorn's emulation framework to provide the user with an easy to use and flexible interface for scripting emulation tasks.

readme for flare-emu

flare-emu

flare-emu marries a supported binary analysis framework, such as IDA Pro or Radare2 , with Unicorn 's emulation framework to provide the user with an easy to use and flexible interface for scripting emulation tasks. It is designed to handle all the housekeeping of setting up a flexible and robust emulator for its supported architectures so that you can focus on solving your code analysis problems. Currently, flare-emu supports the `x86` , `x86_64` , `ARM` , and `ARM64` architectures.

It currently provides five different interfaces to serve your emulation needs, along with a slew of related helper and utility functions.

1. `emulateRange` – This API is used to emulate a range of instructions, or a function, within a user-specified context. It provides options for user-defined hooks for both individual instructions and for when "call" instructions are encountered. The user can decide whether the emulator will skip over, or call into function calls. This interface provides an easy way for the user to specify values for given registers and stack arguments. If a bytestring is specified, it is written to the emulator's memory and the pointer is written to the register or stack variable. After emulation, the user can make use of `flare-emu` 's utility functions to read data from the emulated memory or registers, or use the Unicorn emulation object that is returned for direct probing. A small wrapper function for `emulateRange` , named `emulateSelection` , can be used to emulate the range of instructions currently highlighted in IDA Pro.

2. `iterate` - This API is used to force emulation down specific branches within a function in order to reach a given target. The user can specify a list of target addresses, or the address of a function from which a list of cross-references to the function is used as the targets, along with a callback for when a target is reached. The targets will be reached,

regardless of conditions during emulation that may have caused different branches to be taken. Like the `emulateRange` API, options for user-defined hooks for both individual instructions and for when "call" instructions are encountered are provided. An example use of the iterate API is to achieve something similar to what our argtracker tool does.

3. `iterateAllPaths` - This API is much like `iterate`, except that instead of providing a target address or addresses, you provide a target function that it will attempt to find all paths through and emulate. This is useful when you are performing code analysis that wants to reach every basic block of a function.

4. `emulateBytes` – This API provides a way to simply emulate a blob of extraneous shellcode. The provided bytes are not added to the IDB and are simply emulated as is. This can be useful for preparing the emulation environment. For example, `flare-emu` itself uses this API to manipulate a Model Specific Register (MSR) for the `ARM64` CPU that is not exposed by Unicorn in order to enable Vector Floating Point (VFP) instructions and register access. The Unicorn emulation object is returned for further probing by the user.

5. `emulateFrom` - This API is useful in cases where function boundaries are not clearly defined as is often the case with obfuscated binaries or shellcode. You provide a starting address, and it will emulate until there is nothing left to emulate or you stop emulation in one of your hooks. With IDA Pro, this can be called with the `strict` parameter set to `False` to enable dynamic code discovery; `flare-emu` will have IDA Pro make instructions as they are encountered during emulation.

Installation

To install `flare-emu` for IDA Pro, simply drop `flare_emu.py`, `flare_emu_ida.py`, and `flare_emu_hooks.py` into your IDA Pro's `python` directory and import it as a module in your IDAPython scripts.

To install `flare-emu` for Radare2, simply ensure that `flare_emu.py`, `flare_emu_radare.py`, and `flare_emu_hooks.py` are in Python's search path for importing modules. When using Radare2 as the binary analysis component for `flare-emu`, r2pipe is required.

In any case, `flare-emu` depends on Unicorn and its Python bindings.

IMPORTANT NOTE
`flare-emu` was written using the new IDA Pro 7x API, it is not backwards compatible with previous versions of IDA Pro.

Usage

While `flare-emu` can be used to solve many different code analysis problems, one of its more common uses is to aid in decrypting strings in malware binaries. FLOSS is a great tool than can often do this automatically for you by attempting to identify the string decrypting function(s) and using emulation to decrypt the strings passed in at every cross-reference to it. However, it is not possible for FLOSS to always be able to identify these functions and emulate them properly using its generic approaches. Sometimes you have to do a little more work, and this is where `flare-emu` can save you a lot of time once you are comfortable with it. Let's walk through a common scenario a malware analyst encounters when dealing with encrypted strings.

Easy String Decryption Scenario with IDA Pro

You've identified the function to decrypt all the strings in an `x86_64` binary. This function is called all over the place and decrypts many different strings. In IDA Pro, you name this function `decryptString`. Here is your `flare-emu` script to decrypt all these strings and place comments with the decrypted strings at each function call as well as logging each decrypted string and the address it is decrypted at.

```
from __future__ import print_function
import flare_emu

def decrypt(argv):
    myEH = flare_emu.EmuHelper()
    myEH.emulateRange(myEH.analysisHelper.getNameAddr("decryptString"), registers = {"arg1":
                            "arg3":argv[2], "arg4":argv[3]})
    return myEH.getEmuString(argv[0])

def iterateCallback(eh, address, argv, userData):
    s = decrypt(argv)
    print("%s: %s" % (eh.hexString(address), s))
    eh.analysisHelper.setComment(address, s, False)

if __name__ == '__main__':
    eh = flare_emu.EmuHelper()
    eh.iterate(eh.analysisHelper.getNameAddr("decryptString"), iterateCallback)
```

In `__main__`, we begin by creating an instance of the `EmuHelper` class from `flare-emu`. This is the class we use to do everything with `flare-emu`. Next, we use the `iterate` API, giving it the address of our `decryptString` function and the name of our callback function that `EmuHelper` will call for each cross-reference emulated up to.

The `iterateCallback` function receives the EmuHelper instance, named `eh` here, along with the address of the cross-reference, the

arguments passed to this particular call, and a special dictionary named `userData` here. `userData` is not used in this simple example, but think of it as a persistent context to your emulator where you can store your own custom data. Be careful though, because `flare-emu` itself also uses this dictionary to store critical information it needs to perform its tasks. One such piece of data is the `EmuHelper` instance itself, stored in the "EmuHelper" key. If you are interested, search the source code to learn more about this dictionary. This callback function simply calls the `decrypt` function, prints the decrypted string and creates a comment for it at the address of that call to `decryptString` .

`decrypt` creates a second instance of `EmuHelper` that is used to emulate the `decryptString` function itself, which will decrypt the string for us. The prototype of this `decryptString` function is as follows: `char * decryptString(char *text, int textLength, char *key, int keyLength)` . It simply decrypts the string in place. Our `decrypt` function passes in the arguments as received by the `iterateCallback` function to our call to `EmuHelper` 's `emulateRange` API. Since this is an `x86_64` binary, the calling convention uses registers to pass arguments and not the stack. `flare-emu` automatically determines which registers represent which arguments based on the architecture and file format of the binary as determined by IDA Pro, allowing you to write at least somewhat architecture agnostic code. If this were 32-bit `x86` , you would use the `stack` argument to pass the arguments instead, like so: `myEH.emulateRange(myEH.analysisHelper.getNameAddr("decryptString"), stack = [0, argv[0], argv[1], argv[2], argv[3]])` . The first stack value is the return address in `x86` , so we just use `0` as a placeholder value here. Once emulation is complete, we call the `getEmuString` API to retrieve the null-terminated string stored in the memory location pointed to by the first argument passed to the function.

Easy String Decryption Scenario with Radare2

Using the same example above, not much changes when working with Radare2 rather than IDA Pro. One difference is that `flare-emu` is currently designed to be run as a command-line script or within a Python shell when working with Radare2. The Python shell is great for ad-hoc problem solving while the command-line script is great for batch processing. The Radare2 version of the script above looks like this:

```
from __future__ import print_function
import flare_emu

def decrypt(argv, eh):
    myEH = flare_emu.EmuHelper(samplePath=sys.argv[1], emuHelper=eh)
    myEH.emulateRange(myEH.analysisHelper.getNameAddr("decryptString"), registers = {"arg1"
```

```
                            "arg3":argv[2], "arg4":argv[3]})
    return myEH.getEmuString(argv[0])

def iterateCallback(eh, address, argv, userData):
    s = decrypt(argv, eh)
    print("%s: %s" % (eh.hexString(address), s))
    eh.analysisHelper.setComment(address, s, False)

if __name__ == '__main__':
    eh = flare_emu.EmuHelper(samplePath=sys.argv[1])
    eh.analysisHelper.setName(<some address>, "decryptString")
    eh.iterate(eh.analysisHelper.getNameAddr("decryptString"), iterateCallback)
```

There are two differences with this script. First, the `EmuHelper` constructor takes a parameter here: `samplePath=sys.argv[1]`. When the `samplePath` parameter is provided, `flare-emu` will use Radare2 with `r2pipe` as its binary analysis engine. You can also see that a second parameter is passed to the second `EmuHelper` instance created in the `decrypt` function. The `emuHelper` parameter takes an existing `EmuHelper` object and clones its memory when creating the new object. Also, if you are using Radare2, the new instance re-uses the existing Radare2 session instead of creating a new one which would add more overhead. Second, `flare-emu` creates a new instance of Radare2 using `r2pipe.open`, so it will likely not have the name `decryptString` for the function we are interested in. You can either set the name yourself using `EmuHelper`'s `analysisHelper` object like so: `eh.analysisHelper.setName(<some address>, "decryptString")`, or you can directly input the address for the calls to `iterate` and `emulateRange`.

Emulation Functions

`emulateRange(startAddr, endAddr=None, registers=None, stack=None, instructionHook=None, callHook=None, memAccessHook=None, hookData=None, skipCalls=True, hookApis=True, strict=True, count=0)` - Emulates the range of instructions starting at `startAddress` and ending at `endAddress`, not including the instruction at `endAddress`. If endAddress is `None`, emulation stops when a "return" type instruction is encountered within the same function that emulation began.

- `registers` is a dictionary with keys being register names and values being register values. Some special register names are created by `flare-emu` and can be used here, such as `arg1`, `arg2`, etc., `ret`, and `pc`.
- `stack` is an array of values to be pushed on the stack in reverse order, much like arguments to a function in `x86` are.

In `x86`, remember to account for the first value in this array being used as the return address for a function call and not the first argument of the function. `flare-emu` will initialize the emulated thread's context and memory according to the values specified in the `registers` and `stack` arguments. If a string is specified for any of these values, it will be written to a location in memory and a pointer to that memory will be written to the specified register or stack location instead.

- `instructionHook` can be a function you define to be called before each instruction is emulated. It has the following prototype: `instructionHook(unicornObject, address, instructionSize, userData)`.
- `callHook` can be a function you define to be called whenever a "call" type instruction is encountered during emulation. It has the following prototype: `callHook(address, arguments, functionName, userData)`.
- `hookData` is a dictionary containing user-defined data to be made available to your hook functions. It is a means to persist data throughout the emulation. `flare-emu` also uses this dictionary for its own purposes, so care must be taken not to define a key already defined. This variable is often named `userData` in user-defined hook functions due to its naming in Unicorn.
- `skipCalls` will cause the emulator to skip over "call" type instructions and adjust the stack accordingly, defaults to `True`.
- `hookApis` causes `flare-emu` to perform a naive implementation of some of the more common runtime and OS library functions it encounters during emulation. This frees you from having to be concerned about calls to functions such as `memcpy`, `strcat`, `malloc`, etc., and defaults to `True`.
- `memAccessHook` can be a function you define to be called whenever memory is accessed for reading or writing. It has the following prototype: `memAccessHook(unicornObject, accessType, memAccessAddress, memAccessSize, memValue, userData)`.
- `strict`, when set to `True` (default), checks branch destinations to ensure the disassembler expects instructions. It otherwise skips the branch instruction. If set to `False` when using IDA Pro, `flare-emu` will make instructions in IDA Pro as it emulates them (DISABLE WITH CAUTION).
- `count` is the maximum number of instructions to emulate, defaults to `0` which means no limit.

`iterate(target, targetCallback, preEmuCallback=None, callHook=None, instructionHook=None, hookData=None, resetEmuMem=False, hookApis=True, memAccessHook=None)` - For each target specified by `target`, a separate emulation is performed from the beginning of the containing function up to the target address. Emulation will be forced down the branches necessary to reach each target. `target` can be the address of a function, in which case the target list is populated with all the cross-references to the specified function. Or, `target` can be an explicit list of targets.

- `targetCallback` is a function you create that will be called by `flare-emu` for each target that is reached during emulation. It has the following prototype: `targetHook(emuHelper, address, arguments, userData)`.
- `preEmuCallback` is a function you create that will be called before emulation for each target begins. You can implement some setup code here if needed.
- `resetEmuMem` will cause `flare-emu` to reset the emulation memory before emulation of each target begins, defaults to `False`.

`iterateAllPaths(target, targetCallback, preEmuCallback=None, callHook=None, instructionHook=None, hookData=None, resetEmuMem=False, hookApis=True, memAccessHook=None, maxPaths=MAXCODEPATHS, maxNodes=MAXNODESEARCH)` - For the function containing the address `target`, a separate emulation is performed for each discovered path through it, up to `maxPaths`.

- `maxPaths` - the max number of paths through the function that will be searched for and emulated. Some of the more complex functions can cause the graph search function to take a very long time or never finish; tweak this parameter to meet your needs in a reasonable amount of time.
- `maxNodes` - the max number of basic blocks that will be searched when finding paths through the target function. This is a safety measure to prevent unreasonable search times and hangs and likely does not need to be changed.

`emulateBytes(bytes, registers=None, stack=None, baseAddress=0x400000, instructionHook=None, hookData=None)` - Writes the code contained in `bytes` to emulation memory at `baseAddress` if possible and emulates the instructions from the beginning to the end of `bytes`.

`emulateFrom(startAddr, registers=None, stack=None, instructionHook=None, callHook=None, memAccessHook=None, hookData=None, skipCalls=True, hookApis=True, strict=True, count=0)` - This API is useful in cases where function boundaries are not clearly defined as is often the case with obfuscated binaries or shellcode. You provide a starting address as `startAddr`, and it will emulate until there is nothing left to emulate or you stop emulation

in one of your hooks. This can be called with the `strict` parameter set to `False` to enable dynamic code discovery; `flare-emu` will have IDA Pro make instructions as they are encountered during emulation.

Utility Functions

The following is an incomplete list of some of the useful utility functions provided by the `EmuHelper` class.

- `hexString(value)` - Returns a hexadecimal formatted string for the value. Useful for logging and print statements.
- `skipInstruction(userData, useAnalysisHelper=False)` - Call this from an emulation hook to skip the current instruction, moving the program counter to the next instruction. `useAnalysisHelper` option was added to handle cases where the binary analysis framework folds multiple instructions into one pseudo instruction and you would like to skip all of them. This function cannot be called multiple times from a single instruction hook to skip multiple instructions. To skip multiple instructions, it is recommended not to write to the program counter directly if you are emulating ARM code as this might cause problems with thumb mode. Instead, try `EmuHelper` 's `changeProgramCounter` API (described below).
- `changeProgramCounter(userData, newAddress)` - Call this from an emulation hook to change the value of the program counter register. This API takes care of thumb mode tracking for the ARM architecture.
- `getRegVal(registerName)` - Retrieves the value of the specified register, being sensitive to sub-register addressing. For example, "ax" will return the lower 16 bits of the EAX/RAX register in `x86` .
- `stopEmulation(userData)` - Call this from an emulation hook to stop emulation. Use this instead of calling the `emu_stop` Unicorn API so that the `EmuHelper` object can handle bookkeeping related to the `iterate` feature.
- `getEmuString(address)` - Returns the string of characters located at an address in the emulated memory, up to a null terminator. Characters are not necessarily printable.
- `getEmuWideString(address)` - Returns the string of "wide characters" located at an address in the emulated memory, up to a null terminator. "Wide characters" is meant loosely here to refer to any series of bytes containing a null byte every other byte, as would be the case for an ASCII string encoded in UTF-16 LE. Characters are not necessarily printable.

- `getEmuBytes(address, length)` - Returns a string of bytes located at an address in the emulated memory.
- `getEmuPtr(address)` - Returns the pointer value located at the given address.
- `writeEmuPtr(address, value)` - Writes the pointer value at the given address in the emulated memory.
- `loadBytes(bytes, address=None)` - Allocates memory in the emulator and writes the bytes to it.
- `isValidEmuPtr(address)` - Returns `True` if the provided address points to valid emulated memory.
- `getEmuMemRegion(address)` - Returns a tuple containing the start and end address of memory region containing the provided address, or `None` if the address is not valid.
- `getArgv()` - Call this from an emulation hook at a "call" type instruction to receive an array of the arguments to the function.
- `addApiHook(apiName, hook)` - Adds a new API hook for this instance of `EmuHelper`. Whenever a call instruction to `apiName` is encountered during emulation, `EmuHelper` will call the function specified by `hook`. If `hook` is a string, it is expected to be the name of an API already hooked by `EmuHelper`, in which case it will call its existing hook function. If `hook` is a function, it will call that function.
- `allocEmuMem(size, addr=None)` - Allocates enough emulator memory to contain `size` bytes. It attempts to honor the requested `address`, but if it overlaps with an existing memory region, it will allocate in an unused memory region and return the new address. If `address` is not page-aligned, it will return an address that keeps the same page-aligned offset within the new region. For example, requesting address `0x1234` when `0x1000` is already allocated, may have it allocate at `0x2000` and return `0x2234` instead.

Learn More

To learn more about flare-emu, please read our introductory blog at https://www.fireeye.com/blog/threat-research/2018/12/automating-objective-c-code-analysis-with-emulation.html .

FIDL - by: MANDIANT

Plugin description

> This is a set of utilities wrapping the decompiler API into something sane. This code focus on vulnerability research and bug hunting.

readme for FIDL

FLARE IDA Decompiler Library

FIDLing with the decompiler API

This is a set of utilities wrapping the decompiler API into something sane. This code focus on vulnerability research and bug hunting, however most of the functionality is generic enough to be used for broader reverse engineering purposes.

Installation

The recommended way to install this is to use python's `pip`. Keep in mind that you have to use the `pip` corresponding to the Python installation IDA is using. In case you have more than one installation (for example 32 and 64 bits), you can find which one IDA uses by typing this into the console:

```
import sys
sys.version
```

`cd` to the directory containing `setup.py`

Release mode: `pip install .`

Development (editable) mode: `pip install -e .[dev]`

In *development mode*, `pip` will install `pytest` and some linters helpful while developing, as well as creating symbolic links under python's packages directory instead of copying FIDL to it. This allows you to modify your `.py` files and test on the fly, without needing to reinstall every time you make a change :)

Documentation

You can find up to date documentation online here

The source distribution has built-in documentation here

---

genmc - by: pat0is

Plugin description

> genmc is an IDAPython script\/plugin hybrid that displays Hexrays decompiler microcode, which can help in developing microcode plugins.

readme for genmc

genmc - Display Hex-Rays Microcode

genmc is an IDAPython script/plugin hybrid that displays Hexrays decompiler microcode, which can help in developing microcode plugins.

Installation / Usage

By running the code as a script within IDA, a Python shell command becomes available which, after typing "install_plugin()", copies the script to $HOME/.idapro/plugins or %APPDATA%Hex-Rays/IDA Pro/plugins.

With the script installed into the plugins folder, it can be invoked from the plugins menu or by using the hotkey 'Ctrl-Shift-M'.

IDA and decompilers >= 7.3 are required.

This IDAPython project is compatible with Python3. For compatibility with older versions of IDA, you may want to check out the Python2 branch of this project.

Keyboard shortcuts/modifiers:

With the microcode viewer focussed:

- 'g': display microcode graph
- 'i': display graph for current micro-instruction
- 'Shift': holding this modifier will create floating graph widgets (instead of using the default docking behavior)

Credits:

- https://github.com/RolfRolles/ for his Microcode Explorer plugin whose original ideas and code this script is heavily based on (https://github.com/RolfRolles/HexRaysDeob). Full credit for most of the code and ideas in its original form belongs to Rolf. Check out his related blog post on Hexblog: http://www.hexblog.com/?p=1248
- https://github.com/NeatMonster/ for porting of the Microcode Explorer C++ code to IDAPython using ctypes when Python bindings for HexRays' microcode were not available yet (https://github.com/NeatMonster/MCExplorer).
- https://github.com/icecr4ck/ for porting MCExplorer for IDAPython from 7.x to 7.3

Please consider using Lucid - An Interactive Hex-Rays Microcode Explorer , instead!

---

abyss - by: pat0is

Plugin description

> Augmentation of postprocess Hexrays decompiler output.

readme for abyss

abyss - Postprocess Hexrays Decompiler Output

Installation

Copy abyss.py and abyss_filters to IDA plugins directory

Usage

Right-click within a decompiler view, pick a filter from the abyss context menu.

Per-filter default settings can be changed by editing the config file: "%APPDATA%/Hex-Rays/IDA Pro/cfg/abyss.cfg"

Disclaimer

Experimental/WIP code, use at your own risk :)

Developers

Create a fresh Python module within ”abyss_filters”, make sure to inherit from the abyss_filter_t class (see abyss.py).

Re-running the plugin from the plugins menu or by pressing the Ctrl-Alt-R keycombo reloads all filters dynamically. This allows for development of filters without having to restart IDA.

Example filters (incomplete list)

signed_ops.py

token_colorizer.py

IDACyber - by: pat0is

Plugin description

> IDACyber is an interactive data visualization plugin for IDA Pro. It consists of external ”color filters” that transform raw data bytes into a canvas that can be used to inspect and navigate data interactively.

readme for IDACyber

IDACyber

Data Visualization Plugin for IDA Pro

IDACyber is an interactive data visualization plugin for IDA Pro. It consists of external ”color filters” that transform raw data bytes into a canvas that can be used to inspect and navigate data interactively. Depending on the filter in context, browsing this data visually can reveal particular structures and patterns, literally from a zoomed-out perspective.

Requirements

- IDA 7.3+
- This IDAPython project is compatible with Python3 only. For compatibility with older versions of IDA, you may want to check out the Python2 branch of this project. The Python2 branch is no longer maintained and thus contains outdated code.

Installation

- Updating: It's recommended to delete "idacyber.py" and the "cyber" folder if you're updating from a previous IDACyber version.
- Installation: Copy "idacyber.py" and the "cyber" folder to the IDA Pro "plugins" folder.

Usage

Ctrl-Shift-C starts the plugin and creates a new dockable window. Multiple instances can be created by re-running the plugin which allows several color filters to be run in parallel. The resulting canvas can be interacted with using keyboard and mouse controls. With an instance of IDACyber on focus, a quick manual can be opened by pressing Ctrl-F1, help about the currently active filter can be shown by pressing Ctrl-F2.

Writing custom color filters

IDACyber is meant to be easily customizable by offering the ability to add new "color filters" to it. A color filter is an external IDAPython script that must be placed within the "cyber" folder, which IDACyber will then load during startup. Its main workhorse consists of the callback function "on_process_buffer()" which each color filter is expected to implement. This function is passed the raw data to be processed by a color filter, which then is supposed to return a list of colors in RGB format. IDACyber will take this list of colors and draw it onto the interactive canvas.

For example code, please check out the existing color filters that can be found in the "cyber" folder. The two filters "NES" and "GameBoy" are two simple examples that can be used as a basic skeleton for writing new color filters.

Example filters

Known bugs

Yes :[

---

dsync - by: pat0is

Plugin description

> IDAPython plugin that synchronizes disassembler and decompiler views.

readme for dsync

dsync

IDAPython plugin that synchronizes decompiled and disassembled code views.

Please refer to comments in source code for more details.

Requires IDA Pro 7.3

N.B.: You may want to use the official synchronization feature introduced with IDA 7.3 instead.

HRDevHelper - by: pat0is

Plugin description

> This plugin for the HexRays decompiler creates a graph of a decompiled function's AST using IDA's internal graph viewer.

readme for HRDevHelper

HRDevHelper

HRDevHelper is an extension for the Hexrays decompiler written in IDAPython and is meant to be a helpful tool for debugging and developing your own Hexrays plugins and scripts. The plugin draws its usefulness from displaying a graph of a decompiled function's respective ctree and creating visual links between its underlying decompiled code and the graphs' individual items.

When invoked, HRDevHelper creates and attaches a ctree graph to the currently active decompiler widget and centers the graph's view on the current ctree item. Subsequently navigating the decompiled code visually highlights corresponding ctree items in the graph.

The plugin's default colors and other settings (zoom, dock position etc.) can be configured by editing the plugin's configuration file that is created after running the plugin for the first time. The HRDevhelper.cfg configuration file can be found in the IDA user directory .

Installation

Copy hrdevhelper.py and the hrdh folder to ./IDA/plugins/ and restart IDA.

Plugin Usage & Shortcuts

The plugin's functionality is accessible via right-click in a decompiler view or otherwise via keyboard shortcuts:

- "show ctree" creates a graph of all ctree items of the current decompiled function.
- "show sub-tree" creates a graph of ctree items that belong to the current expression. The subgraph's root is determined via the current decompiler view's text cursor.
- "show context" opens a context viewer that, among other information, displays the current sub-tree's citems as a lambda expression. This expression can be used with and directly copy-pasted into hxtb-shell that comes with the HexraysToolbox script.

By default, HRDevHelper visually highlights all ctree items in a graph that correspond to a current single decompiled line of code. Making a selection of multiple lines highlights nodes accordingly.

In addition to the keyboard shortcuts that are made available in decompiler views, the graphs created by HRDevHelper have additional keyboard shortcuts in place as shown below.

Graph Hotkeys (focus on any HRDevHelper graph/subgraph):

- C: Toggle "center on current item/node" functionality (switches synchronization on/off).
- D: Increase debug/verbosity of particular nodes

Hexrays Toolbox - by: pat0is

Plugin description

> Hexrays Toolbox is a script for the Hexrays Decompiler which can be used to find code patterns within decompiled code.

readme for Hexrays Toolbox

Hexrays Toolbox

Hexrays Toolbox (hxtb) is a powerful script for the Hexrays Decompiler which can be used to find code patterns in decompiled code. Some of its use cases are described below.

Use Cases

- scan binary files for known and unknown vulnerabilities
- locate code patterns from previously reverse engineered executables in newly decompiled code
- malware variant analysis
- find code similarities across several binaries
- find code patterns from one architecture in executable code of another architecture
- many more, limited (almost) only by the queries you'll come up with ;)

The query shown below shows an example of how a particular code pattern can be identified that was responsible for a remote code execution security vulnerability in WHatsApp in 2019 (CVE-2019-3568, libwhatsapp.so). Find the example script .

Usage

There are several ways of using Hexrays Toolbox, each with a varying degree of flexibility.

- run queries on behalft of hxtb_shell , an interactive GUI
- custom scripting
- `interactive.py` , a script that adds convenience functions to be used with the IDA command line interface
- `automation.py` , a script that processes and runs hxtb queries on a given set of files in batch mode

hxtb-shell

Executing the included `hxtb_shell.py` script from within IDA opens a GUI window that can be used to develop and run hxtb queries with. hxtb-shell also accepts Python expressions created by the HRDevHelper plugin's context viewer that can be directly copy-pasted.

Below screenshot shows hxtb-shell with an example query loaded that creates and shows a list of strings referenced by the current function.

Further queries in the hxtb-shell format can be found in the `hxtbshell_queries` folder included with hxtb.

Scripting

Loading `hxtb.py` with IDA (alt-f7) makes functions such as `find_expr()` and `find_item()` available to both the IDAPython CLI and the script interpreter (shift-f2). Among others, these functions can

be used to run queries on the currently loaded ida database. Please check out some of the examples shown below .

```
    find_item(ea, q)
    find_expr(ea, q)

    Positional arguments:
        ea:         address of a valid function within
                    the current database
        q:          lambda function
                    custom lambda function with the following arguments:
                    1. cfunc: cfunc_t
                    2. i/e:   cinsn_t/cexpr_t
    Returns:
        list of query_result_t objects

    Example:
        find_expr(here(), lambda cf, e: e.op is cot_call)

        -> finds and returns all function calls within a current function.
        The returned data is a list of query_result_t objects (see hxtb.py).

        The returned list can be passed to an instance of the ic_t class,
        which causes the data to be displayed by a chooser as follows:

        from idaapi import *
        import hxtb
        hxtb.ic_t(find_expr(here(), lambda cf,e:e.op is cot_call))


    Please find the cfunc_t, citem_t, cinsn_t and cexpr_t structures
    within hexrays.hpp for further help and details.
```

Please also check out the HRDevHelper plugin and the IDAPyHelper script which may assist in writing respective queries.

Examples

get list of expressions that compare anything to zero ("x == 0")

```
         cot_eq
         /   \
      x /     \ y
(anything)  cot_num --- n.numval() == 0
```

```
from idaapi import *
from hxtb import find_expr
query = lambda cfunc, e: e.op is cot_eq and e.y.op is cot_num and e.y.numval() == 0
```

```
r = find_expr(here(), query)
for e in r:
    print(e)
```

get list of function calls

```
        cot_call
         /
      x /
 cot_obj
```

```
from idaapi import *
from hxtb import find_expr
query = lambda cfunc, e: e.op is cot_call and e.x.op is cot_obj
r = find_expr(here(), query)
for e in r:
    print(e)
```

print list of memcpy calls where "dst" argument is on stack

```
        cot_call --- arg1 is cot_var
         /           arg1 is on stack
      x /
 cot_obj --- name(obj_ea) == 'memcpy'
```

```
from idaapi import *
from hxtb import find_expr
r = []
query = lambda cfunc, e: (e.op is cot_call and
           e.x.op is cot_obj and
           get_name(e.x.obj_ea) == 'memcpy' and
           len(e.a) == 3 and
           e.a[0].op is cot_var and
           cfunc.lvars[e.a[0].v.idx].is_stk_var())
for ea in Functions():
    r += find_expr(ea, query)
for e in r:
    print(e)
```

get list of calls to sprintf(str, fmt, ...) where fmt contains "%s"

```
        cot_call --- arg2 ('fmt') contains '%s'
         /
      x /
 cot_obj --- name(obj_ea) == 'sprintf'
```

```
from idaapi import *
from hxtb import find_expr
```

```
r = []
query = lambda cfunc, e: (e.op is cot_call and
    e.x.op is cot_obj and
    get_name(e.x.obj_ea) == 'sprintf' and
    len(e.a) >= 2 and
    e.a[1].op is cot_obj and
    is_strlit(get_flags(get_item_head(e.a[1].obj_ea))) and
    b'%s' in get_strlit_contents(e.a[1].obj_ea, -1, 0, STRCONV_ESCAPE))
for ea in Functions():
    r += find_expr(ea, query)
for e in r:
    print(e)
```

get list of signed operators, display result in chooser

```
from idaapi import *
from hxtb import ic_t
query = lambda cfunc, e: (e.op in
            [hr.cot_asgsshr, hr.cot_asgsdiv,
            hr.cot_asgsmod, hr.cot_sge,
            hr.cot_sle, hr.cot_sgt,
            hr.cot_slt, hr.cot_sshr,
            hr.cot_sdiv, hr.cot_smod])
ic_t(query)
```

get list of "if" statements, display result in chooser

```
from idaapi import *
from hxtb import ic_t
ic_t(lambda cf, i: i.op is cit_if)
```

get list of all loop statements in db, display result in chooser

```
from idaapi import *
from hxtb import ic_t, query_db
ic_t(query_db(lambda cf,i: is_loop(i.op)))
```

get list of loop constructs containing copy operations

```
from hxtb import ic_t, query_db, find_child_expr
from ida_hexrays import *


find_copy_query = lambda cfunc, i: (i.op is cot_asg and
                                    i.x.op is cot_ptr and
                                    i.y.op is cot_ptr)
```

```
find_loop_query = lambda cfunc, i: (is_loop(i.op) and
                                    find_child_expr(cfunc, i, find_copy_query))


ic_t(query_db(find_loop_query))
```

---

lumen - by: Naim A.

Plugin description

A private Lumina server that can be used with IDA Pro 7.2+.

readme for lumen

Lumen

A private Lumina server that can be used with IDA Pro 7.2+.

lumen.abda.nl runs this server.

You can read about the protocol research here .

Features

- Stores function signatures so you (and your team) can quickly identify functions that you found in the past using IDA's built-in Lumina features.
- Backed by PostgreSQL
- Experimental HTTP API that allows querying the database for comments by file or function hash.

Getting Started

Binary releases

Release binaries are available at https://github.com/naim94a/lumen/releases/latest .

Building from source with Rust

1. `git clone https://github.com/naim94a/lumen.git`
2. Get a rust toolchain: https://rustup.rs/
3. `cd lumen`
4. Setup a Postgres database and execute src/schema.sql on it
5. `cargo build --release`

Docker Method

1. Install `docker-engine` and `docker-compose`.
2. If using a custom TLS certificate, copy the private key (`.p12` / `.pfx` extension) to `./dockershare` and set the key password in `.env` as `PKCSPASSWD`.
3. If using a custom Lumen config, copy it to `./dockershare/config.toml`.
4. Otherwise, or if you have finished these steps, just run `docker-compose up`.
5. Regardless, if TLS is enabled in the `config.toml`, a `hexrays.crt` will be generated in `./dockershare` to be copied to the IDA install directory.

Usage

```
./lumen -c config.toml
```

Configuring IDA

IDA Pro >= 8.1

If you used LUMEN in the past, remove the LUMINA settings in the ida.cfg or idauser.cfg files, otherwise you will get a warning about bad config parameters.

Setup under Linux :

```
#!/bin/sh
export LUMINA_TLS=false
$1
```

- save as ida_lumen.sh, "chmod +x ida_lumen.sh", now you can run IDA using "./ida_lumen.sh ./ida" or "./ida_lumen ./ida64"

Setup under Windows :

```
set LUMINA_TLS=false
%1
```

- save as ida_lumen.bat, now you can run IDA using "./ida_lumen.bat ida.exe" or "./ida_lumen.bat ida64.exe"

Setup IDA

- Go to Options, General, Lumina. Select "Use a private server", then set your host and port and "guest" as username and password. Click on ok.

IDA Pro < 8.1

You will need IDA Pro 7.2 or above in order to use *lumen* .

> The following information may get sent to *lumen* server: IDA key, Hostname, IDB path, original file path, file MD5, function signature, stack frames & comments.

- In your IDA's installation directory open "cfg\ida.cfg" with your favorite text editor *(Example: C:\Program Files\IDA Pro 7.5\cfg\ida.cfg)*
- Locate the commented out `LUMINA_HOST` , `LUMINA_PORT` , and change their values to the address of your *lumen* server.
- If you didn't configure TLS, Add "LUMINA_TLS = NO" after the line with `LUMINA_PORT` .

Example:

```
LUMINA_HOST = "192.168.1.1";
LUMINA_PORT = 1234

// Only if TLS isn't used:
LUMINA_TLS = NO
```

Configuring TLS

IDA Pro uses a pinned certificate for Lumina's communcation, so adding a self-signed certificate to your root certificates won't work. Luckily, we can override the hard-coded public key by writing a DER-base64 encoded certificate to "hexrays.crt" in IDA's install directory.

You may find the following commands useful:

```
# create a certificate
openssl req -x509 -newkey rsa:4096 -keyout lumen_key.pem -out lumen_crt.pem -days 365 -nodes

# convert to pkcs12 for lumen; used for `lumen.tls` in config
openssl pkcs12 -export -out lumen.p12 -inkey lumen_key.pem -in lumen_crt.pem

# export public-key for IDA; Copy hexrays.crt to IDA installation folder
openssl x509 -in lumen_crt.pem -out hexrays.crt
```

No attempt is made to merge function data - this may casuse a situation where metadata is inconsistent. Instead, the metadata with the highest calculated score is returned to the user.

Developed by Naim A. ; License: MIT.

ComIDA - by: Airbus CERT

Plugin description

An IDA Plugin that help during the analysis of modules using COM.

readme for ComIDA

ComIDA

An IDA Plugin that help during the analysis of modules using COM. It works by searching data references to known COM GUID (Classes or Interfaces), and for hex-ray plugin user, infers type that use :

- `CoCreateInstance` function
- `CoGetCallContext` function
- `QueryInterface` method

BEFORE:

AFTER:

How To Install ?

Just put the `comida.py` script in `plugins` folder of IDA.

```
git clone git@github.com:Airbus-CERT/comida.git
mklink "C:\\Program Files\\IDA Pro 7.4\\plugins\\comida.py" "comida\comida.py"
```

Launch your IDA and press `Ctrl-Shift-M` to activate it.

How Does It Works ?

COM Object References

`ComIDA` has two main features:

- Finding and Tracking GUID
- Type infering for Hex-Ray plugin users

To find and track GUID, we just search direct operand menmoniques, like `mov ax, GUID` , where GUID matches one present in `HKCR\Classes` or in `HKCR\Interfaces` hives.

We preview the results in a table to interact and easily navigate through all COM object references.

Just double click to go to the interesting code.

Type Infering

The main goal is to facilitate the work of the analyst. When a module uses COM services, it commonly uses:

- Create instance using `CoCreateInstance`
- Retrieve instance using `CoGetCallContext`
- Cast interface using method `QueryInterface` from `IUnknown` interface, which are inherited by every COM classes

comIDA will perform type infering for these patterns.

In comIDA, the infering works as the following:

1. Find interesting function
2. Read GUID parameter and retrieve interface type
3. Change type of the output variable

To find interesting function we have two kinds of heuristics, one for each type:

- For function type, like `CoCreateInstance` or `CoGetCallContext` , we directly search into import table.
- For method type, like `QueryInterface` we compute the method name thanks to the Hex-Rays AST.

After that, we use Hex-Ray hook to navigate into the function AST during decompilation step. When we spot a call to the interesting method, we perform a type change of the output parameter accordingly to the GUID parameter (GUID of the Interface). To choose the correct type, we have two heuristics to select the type:

- We check the name in the registry hive `HKCR\\Interfaces` associated with Interface GUID
- We try to cast directly into the symbol name associated with the third parameter (Mostly named `IID_IWbemLocator` etc... just take IWbemLocator)

And then the magic happened:

BEFORE:

AFTER:

BEFORE:

AFTER:

Virtuailor - by: 0xgalz

Plugin description

> IDAPython tool for C++ vtables reconstruction.

readme for Virtuailor

Virtuailor - IDAPython tool for C++ vtables reconstruction

Virtuailor is an IDAPython tool that reconstructs vtables for C++ code written for intel architecture, both 32bit and 64bit code and AArch64 (New!). The tool constructed from 2 parts, static and dynamic.

The first is the static part, contains the following capabilities:

- Detects indirect calls.
- Hooks the value assignment of the indirect calls using conditional breakpoints (the hook code).

The second is the dynamic part, contains the following capabilities:

- Creates vtable structures.
- Rename functions and vtables addresses.
- Add structure offset to the assembly indirect calls.
- Add xref from indirect calls to their virtual functions(multiple xrefs).
- For AArch64- tries to fix undefined vtables and related virtual functions (support for firmware).

How to Use?

> Virtuailor now supports IDA versions from 7.0 to the newest version (7.5), if you are using IDA versions older than 7.4 you will need to switch to branch beforeIDA-7.4, master branch supports the newest version available (7.5).

1. By default Virtuailor will look for virtual calls in ALL the addresses in the code. If you want to limit the code only for specific address range, no problem, just edit the *Main* file to add the range you want to target in the variables start_addr_range and end_addr_range:

```
if __name__ == '__main__':

    start_addr_range = idc.MinEA()  # You can change the virtual calls address range
    end_addr_range = idc.MaxEA()
    add_bp_to_virtual_calls(start_addr_range, end_addr_range)
```

2. Optional, (but extremely recommended), create a snapshot of your idb. Just press ctrl+shift+t and create a snapshot.

3. Press File->Run script... then go to Virtuailor folder and choose to run Main.py, You can see the following gif for a more clear and visual explanation.

Now the GUI will provide you an option to choose a range to target, in case you would like to target all the binary just press OK with the default values in the start and end addresses.

Afterwards the breakpoints will be placed in your code and all you have to do is to execute your code with IDA debugger, do whatever actions you want and see how the vtables is being built! For AArch64 you can setup a remote gdb server and debug using the IDA debuggger.

In case you don't want/need the breakpoints anymore just go to the breakpoint list tab in IDA and delete the breakpoints as you like.

It is also really important for me to note that this is the second version of the tool with both 32 and 64 bit support and aarch64, probably in some cases a small amount of breakpoints will be missed, in these cases please open an issue and contact me so I will be able to improve the code and help fixing it. Thank you in advanced for that :)

Output and General Functions

vtables structures The structures Virtuailor creates from the vtable used in virtual call that were hit. The vtable functions are extracted from the memory based on the relevant register that was used in the BP opcode.

Since I wanted to create a correlation between the structure in IDA and the vtables in the data section, the BP changes the vtable address name in the data section to the name of the structure. As you can see in the following picture:

The virtual functions names are also being changed, take aside situations where the names are not the default IDA names (functions with symbols or functions that the user changed) in those cases the function names will stay the same and will also be add to the vtable structure with their current name.

The chosen names is constructed using the following pattern:

- vtable_
- vfunc_ the rest of the name is either offset from the beginning of the Segment, this is mostly because most binaries nowadays are PIE and PIC and thus ASLR is enforced, (instead of using the full address name, which is also quite long on 64bit environments). The vtable structure also has a comment, "Was called from offset: XXXX", this offset is the offset from the beginning of the Segment.

Adding Structures to the Assembly After creating the vtable Virtuailor also adds a connection between the structure created and the assembly as you can see in the following images:

P.S: The structure offset used in the BP is only relevant for the last call that was made, in order to get a better understanding of all the virtual calls that were made the xref feature was added as explained in the next section

Xref to virtual functions When reversing C++ statically it is not trivial to see who called who, this is because most calls are indirect calls, however after

running Virtuailor every function that was called indirectly now has an xref to those locations.

The following gif shows the added Xrefs with their indirect function call:

Former talks and lectures

The tool was presented in RECon brussels, Troopers and Warcon. The presentation could be found in the following link: https://www.youtube.com/watch?v=Xk75TM7NmtA

Thanks

REcon Brussels, Troopers, Warcon crews, Nana, @tmr232, @matalaz, @oryandp, @talkain, @shiftreduce

License

The plugin is licensed under the GNU GPL v3 license.

Ghida - by: Cisco Talos

Plugin description

> GhIDA is an IDA Pro plugin that integrates the Ghidra decompiler in IDA.

readme for Ghida

GhIDA - Ghidra Decompiler for IDA Pro

GhIDA is an IDA Pro plugin that integrates the Ghidra decompiler in IDA.

How does it work?

Select a function, both in the *Graph view* or in the *Text View* . Then, Press CTRL+ALT+D or (*Edit > Plugins > GhIDA Decompiler*). Wait a few seconds and a new window will open showing the decompiled code of the function.

GhIDA requires either a local installation of Ghidra or the Ghidraaas server.

The plugin correctly handles x86 and x64 PE and ELF binaries.

Read more about all the GhIDA features in the Features section.

If you want to discover how GhIDA works under the hood, read the Technical details section.

Features

GhIDA provides the following features:

- *Synchronization* of the disassembler view with the decompiler view
- Decompiled code *syntax highlight*
- Code navigation by double-clicking on symbols' name
- Add comments in the decompiler view
- Symbols renaming (limited to XML exported symbols and few others)
- Symbols highlight on disassembler and decompiler view
- Decompiled code and comments cache
- Store setting options.

More information are provided in the Features description section.

Requirements

- GhIDA has a Python 2 and Python 3 version:
  - For Python 2 requires IDA Pro 7.x
  - For Python 3 requires IDA Pro >= 7.4
  - GhIDA is not compatible with *IDA Home*
- `requests` and `pygments` Python (2 or 3) packages
- A local installation of Ghidra or Ghidraaas .
  - Use Ghidra version 9.1.2

Installation

- Install `requests` and `pygments` in Python 2 or Python 3.

```
pip install requests
pip install pygments
```

- Download the latest release from the release page.
- Copy `ghida.py` and the `ghida_plugin` folder in the `plugins` folder of your IDA Pro installation.

- The first time GhIDA is launched (Ctrl+Alt+D or *Edit > Plugins > GhIDA Decompiler*), a settings form is displayed, as shown in the previous image.
  - If you want to use GhIDA with a local installation of Ghidra:
    * Download and install Ghidra 9.1.2
    * Fill the *installation path* (path of the `ghidra_9.1.2_PUBLIC` folder)
  - Otherwise, if you want to use Ghidraaas:
    * Launch a local instance of the server using the Ghidraaas docker container
    * Check the *Use Ghidraaas server* box, and insert `http://0.0.0.0:8080/ghidra/api` .
- To run GhIDA:
  - Ctrl+Alt+D
  - *Edit > Plugins > GhIDA Decompiler*
  - (in the disassembler view) *right click > Decompile function with GhIDA*
- To reopen the decompiler view:
  - Either run GhIDA again (see the previous point), or *View > Open subviews > GhIDA decomp view* .

Suggestions for the best user experience

- Open the decompile view side-to-side with the disassembler view, and keep active the synchronization between the two views.
- When you rename a symbol (e.g., a function name), rename it in the decompile view, it will be updated automatically in the disasm view too. Otherwise, you will need to delete the decompiled code from the cache and decompile the function again.
- If the program is rebased, all the caches (decompiler, comments, symbol table) are invalidated. Functions must be decompiled again.
- It's possible to change the `TIMEOUT` value for the Ghidra analysis in ghida_plugin/lib.py . By default, it's set to 300 seconds, but it may be increased if needed. Please, do not modify the value of `COUNTER_MAX` or `SLEEP_LENGTH` , since they are all related.

Features description

Synchronization

By default, the disassembler view is synchronized with the decompiler view. By clicking on different functions both in IDA *Graph view* or *Text View* , the

decompiler view is updated accordingly. This behaviour is particularly useful if the decompiler view is displayed side-to-side with the disassembler view.

To disable the synchronization (in the disassembler view) *right-click > Disable decompiler view synchronization* .

Code syntax highlight

Decompiled code is syntax-highlighted using the pygments Python library.

Code navigation

In the decompiler view, double click (or *right-click > Goto*) over the name of a function to open it in the decompile and disassembler view. If the function has not been decompiled yet, then press CTRL+ALT+D if you want to decompile it.

Comments

GhIDA allows to insert and update comments in the decompile view. The comment will be displayed at the end of the selected line, separated by `//` .

To add a comment (in the decompiler view) press `:` or *right-click > Add comment* and insert the comment in the dialog.

Comments are stored internally, and are automatically added whenever a function is decompiled. They also persist when the GhIDA cached code is invalidated. Moreover, if the corresponding option is selected in the configuration menu, cached comments are dumped to file and then loaded at the next opening. The cache is saved in JSON format in the temporary folder.

Symbols renaming

To rename a symbol (in the decompiler view) select the symbol you want rename, press N (or *right-click > Rename*), then insert the new name in the dialog.

Due to the different syntax used by Ghidra and IDA, only a subset of the symbols can be renamed.

Symbols highlight

In the decompiler view, when clicking on a symbol, all the other occurrences of the same symbol are highlighted. The plugin also highlights the corresponding symbols in the disassembler view, but it is limited to XML exported symbols and few others.

Decompiled code and comments cache

GhIDA cache the results of the decompilation and automatically shows the decompiled code when a cached decompilation is requested. However, if the user updates the symbols in IDA, or performs any other action that requires the code to be decompiled again, the user can remove a decompiled code from the cache.

To remove the code from the cache (in the disassembler view) *right-click > Clear cache for current function* .

If the corresponding option is selected in the configuration, cached code is dumped to file and loaded at the next opening. The cache is saved in JSON format in the temporary folder.

Store setting options

To avoid retype GhIDA configuration each time IDA is opened, the configuration is saved in a JSON file in the temporary folder.

Technical details

Under the hood, GhIDA exports the IDA project using idaxml.py , a Python library shipped with Ghidra, then it directly invokes Ghidra in headless mode without requiring any additional analysis. When GhIDA is called the first time, it uses `idaxml` to create two files: a XML file which embeds a program description according to the IDA analysis (including functions, data, symbols, comments, etc) and a `.bytes` file that contains the binary code of the program under analysis. While the binary file does not change during the time, the XML file is recreated each time the user invalidates the GhIDA cache, in order to take into account the updates the user did in the program analysis. To obtain the decompiled code, GhIDA uses `FunctionDecompile.py` , a Ghidra plugin in Python that exports to a JSON file the decompiled code of a selected function.

Exporting the IDA's IDB and calling Ghidra in headless mode add a small overhead to the decompilation process, but it allows to abstract the low-level communication with the Ghidra decompiler.

Development

Ghida outputs to the IDA console some messages related to the main operations, using the following syntax:

- `GhIDA [DEBUG]` display a debug message
- `GhIDA [WARNING]` display a warning message
- `GhIDA [!]` display an error message

Improvements

- Check the support for other file formats (other than *PE* and *ELF*) and architectures (other than *x86* and *x64*): `idaxml` may require specific checks during the project export phase.
- Improve the syntax conversion algorithm from Ghidra to IDA and vice versa. This will increase the number of symbols that can be highlighted or renamed.
- Add a batch-mode option that decompiles all the functions in the background and cache them.

Bugs and suggestion

If you discover a bug, or you have any improvements or suggestions, please open an issue .

Be sure to include as many details as possible in order to reproduce the bug.

License

GhIDA is licensed under the Apache License 2.0 .

`idaxml.py` is a library shipped with Ghidra and it is distributed under the Apache License 2.0 .

Acknowledgement

Thanks to all the people from Talos Malware Research Team for the insightful comments and suggestions.

---

xray - by: pat0is

Plugin description

> xray is a plugin for the Hexrays decompiler that both filters and colorizes the textual representation of the decompiler's output based on configurable regular expressions.

readme for xray

xray - Filter Hex-Rays Decompiler Output

xray is a plugin for the Hexrays decompiler that both filters and colorizes the textual representation of the decompiler's output based on configurable regular

expressions.

This helps highlighting interesting code patterns which can be useful in malware analysis and vulnerability identification.

Installation/Updating:

xray installs or updates itself as a plugin by loading it as a script using the "File->Script file..." (Alt-F7) menu item within IDA.

Running the plugin for the first time creates a default configuration file "xray.cfg" within the folder "%APPDATA%/Hex-Rays/IDA Pro/plugins/", which can and should then be customized by the user.

While still under development, updating from a previous installation of the plugin may introduce changes to the configuration file which may cause incompatibility. If this is the case, the current configuration file should be ported to the new format or deleted.

xray requires IDA 7.2+ (with some effort it may be backported to 7.0).

This IDAPython project is compatible with Python3. For compatibility with older versions of IDA, you may want to check out the Python2 branch of this project.

Usage:

The plugin offers two distinct filtering/highlighting features:

- "xray", a persistent, configurable regular expression parser that applies color filters to the output of the Hexrays decompiler. This filter can be turned on and off using a keyboard shortcut as described in the next section.

  Persistent filtering attempts to match regular expressions taken from the plugin's configuration file against each of the decompiler's text lines. Successful matches will cause the background color of a matching text line to be changed accordingly. Optionally, changing the "high_contrast" setting to "1" in the configuration file will cause a visual "xray" effect.

  For more settings and details, please refer to the comments in the configuration file.

- a dynamic filter that filters/highlights Hexrays output. This filter works similar to how the built-in filters for IDA "choosers" work. Possible "filter types" are "Regex" and "ASCII". Additional "filter options" determine how the filters are applied to respective Hexrays output:

  – "Text" removes any lines from the decompiler's output that a specified search term could not be matched against.

– "Color" does not remove non-matching lines but only their respective color tags instead. This will cause matching text to be highlighted visually.

Popup Menus/Keyboard shortcuts:

- F3: Toggle xray
- Ctrl-R: Reload xray configuration file and apply changes (edit and reload the configuration file on-the-fly)
- Ctrl-F: Find ascii string/regular expression and apply filters based on Filter type and options. "Text": removes any non-matching lines from the outpout "Color": removes colors from non-matching lines

---

mkYARA - by: Fox-IT

Plugin description

> mkYARA comes with a IDA plugin to easily create YARA signatures by selecting a set of instructions and choosing one of the mkYARA -> Generate YARA rule options.

readme for mkYARA

mkYARA

Writing YARA rules based on executable code within malware can be a tedious task. An analyst cannot simply copy and paste raw executable code into a YARA rule, because this code contains variable values, such as memory addresses and offsets. The analyst has to disassemble the code and wildcard all the pieces in the code that can change between samples. mkYARA aims to automate this part of writing rules by generating executable code signatures that wildcard all these little pieces of executable code that are not static.

Installation

Installation is as easy as installing the pip package.

```
pip install mkyara
```

Usage

```
import codecs
from capstone import CS_ARCH_X86, CS_MODE_32
```

```
from mkyara import YaraGenerator

gen = YaraGenerator("normal", CS_ARCH_X86, CS_MODE_32)
gen.add_chunk(b"\x90\x90\x90", offset=1000)
gen.add_chunk(codecs.decode("6830800000E896FEFFFFC3", "hex"), offset=0x100)
gen.add_chunk(b"\x90\x90\x90\xFF\xD7", is_data=True)
rule = gen.generate_rule()
rule_str = rule.get_rule_string()
print(rule_str)
```

Standalone Tool

mkYARA comes with a standalone tool that is cross platform, as in, it can create signatures for Windows binaries running under Linux.

```
usage: mkyara [-h] [-i {x86}] [-a {32,64,x86,x64}] -f FILE_PATH [-n RULENAME]
              -o OFFSET -s SIZE [-m {loose,normal,strict}] [-r RESULT] [-v]

Generate a Yara rule based on disassembled code

optional arguments:
  -h, --help            show this help message and exit
  -i {x86}, --instruction_set {x86}
                        Instruction set
  -a {32,64,x86,x64}, --instruction_mode {32,64,x86,x64}
                        Instruction mode
  -f FILE_PATH, --file_path FILE_PATH
                        Sample file path
  -n RULENAME, --rulename RULENAME
                        Generated rule name
  -o OFFSET, --offset OFFSET
                        File offset for signature
  -s SIZE, --size SIZE  Size of desired signature
  -m {loose,normal,strict}, --mode {loose,normal,strict}
                        Wildcard mode for yara rule generation
                        loose = wildcard all operands
                        normal = wildcard only displacement operands
                        strict = wildcard only jmp/call addresses
  -r RESULT, --result RESULT
                        Output file
  -v, --verbose         Increase verbosity
```

IDA Plugin

mkYARA comes with a IDA plugin to easily create YARA signatures by selecting a set of instructions and choosing one of the mkYARA -> Generate YARA

rule options. Installation is as easy as installing the pip package and copying the mkyara_plugin.py to your IDA plugin directory.

VT-IDA-PLUGIN - by: VirusTotal

Plugin description

> This plugin integrates functionality from VirusTotal web services into the IDA Pro's user interface.

readme for VT-IDA-PLUGIN

VT-IDA Plugin

This is the official VirusTotal plugin for Hex-Rays IDA Pro. This plugin integrates functionality from VirusTotal web services into the IDA Pro's user interface.

The current version is v0.10. This plugin is not production-ready yet, and unexpected behavior can still occur. This release integrates VTGrep into IDA Pro, facilitating the searching for similar code, strings, or sequences of bytes.

Requirements

This plugin has been developed for IDA Pro 7.0 and beyond and supports both Python 2.7 and 3.x. It requires the "requests" module, the easiest way of installing it is by using `pip` :

```
$ pip install requests
```

Installation

Copy the content of the `plugin` directory into the IDA Pro's plugin directory and start IDA Pro.

OS	Plugin path
Linux	`/opt/ida-7.X/plugins`
macOS	`~/.idapro/plugins`
Windows	`%ProgramFiles%\IDA 7.X\plugins`

Usage

While in the disassembly window, select an area of a set of instructions and right-click to chose one of the following actions:

- *Search for bytes* : it searches for the bytes contained in the selected area.
- *Search for string* : it searches for the same string as the one selected in the Strings Window.
- *Search for similar code* : identifies memory offsets or addresses in the currently selected area and ignores them when searching.
- *Search for similar code (strict)* : same as above but it also ignores all the constants in the currently selected area.
- *Search for similar functions* : same as "similar code" but you don't need to select all the instructions that belong to a function. Just right-click on one instruction, and it will automatically detect the function boundaries, selecting all the instructions of the current function.

Another option is to look for similar strings. To search for similar ones, open the `Strings Windows` in IDA Pro, right-click on any string (one or many) and select `Virus Total -> Search for string` .

These actions will launch a new instance of your default web browser, showing all the matches found in VTGrep. Remember that your default web browser must be logged into your VirusTotal Enterprise account in order to see the results.

Check IDA Pro's output window for any message that may need your attention.

Note : This version supports Intel 32/64 bits and ARM processor architectures when searching for similar code. Probably more architectures are supported but it hasn't been tested yet.

FindYara - by: OALabs

Plugin description

IDA python plugin to scan binary with yara rules.

readme for FindYara

FindYara

Use this IDA python plugin to scan your binary with yara rules. All the yara rule matches will be listed with their offset so you can quickly jump to them!

? All credit for this plugin and the code goes to David Berard (@ *p0ly*) ?

This plugin is copied from David's excellent findcrypt-yara plugin . This plugin just extends his to use any yara rule.

Using FindYara

The plugin can be launched from the menu using `Edit->Plugins->FindYara` or using the hot-key combination `Ctrl-Alt-Y` . When launched the FindYara will open a file selection dialogue that allows you to select your Yara rules file. Once the rule file has been selected FindYara will scan the loaded binary for rule matches.

All rule matches are displayed in a selection box that allows you to double click the matches and jump to their location in the binary.

Rules Not Matching Binary

FindYara scans the loaded PE sections in IDA, this means that yara rules that include matches on the PE header will not match in IDA . IDA does not load the PE header as a scannable section. Also, if you have not selected `Load resources` when loading your binary in IDA then the resources section will be unavailable for scanning.

This can lead to frustrating situations where a yara rule will match outside of IDA but not when using FindYara. If you encounter this try editing the yara rule to remove the matches on the PE header and resources sections.

Installing FindYara

Before using the plugin you must install the python Yara module in your IDA environment. The simplest way to do this is to use pip from a shell outside of IDA.
`pip install yara-python` .

Do not install the `yara` module by mistake. The `yara` python module will mess with your `yara-python` module so it must be uninstalled if it was installed by mistake.

Once you have the yara module installed simply copy the latest release of `findyara.py` into your IDA plugins directory and you are ready to start Yara scanning!

Compatibility Issues

FindYara has been developed for use with the IDA 7+ and Python 3 it is not backwards compatible.

FindYara requires a the python Yara module with version 4+ installed. Earlier versions of Yara are not compatible with the plugin and may cause issues due to breaking changes in the Yara match format.

Acknowledgments

A huge thank you to David Berard (@ *p0ly*) - Follow him on GitHub here ! This is mostly his code and he gets all the credit for the original plugin framework.

Also, hat tip to Alex Hanel @nullandnull - Follow him on GitHub here . Alex helped me sort through how the IDC methods are being used. His IDA Python book is a fantastic reference!!

---

golang_loader_assist - by: Tim Strazzere

Plugin description

> Making GO reversing easier in IDA Pro

readme for golang_loader_assist

golang_loader_assist.py

This is the `golang_loader_assist.py` code to accompany the blog I wrote, Reversing GO binaries like a pro (in IDA Pro) . There is also the `hello-go` directory which contains the simple hello world code I used as an example.

Important notes

If you're using IDA Pro 7.3 or below, you likely will need to take a look at the older release tagged IDA-7.3-and-Below . This is due to changes in the IDA Python libraries which where introduced in 7.4 which do not look to be backwards compatible.

TODO

- Support IDA Pro 7.5 w/ Python3 (tested with a go1.13.6 and go1.14.4 binary on IDA 7.5.200519 Linux x86_64)
- Support IDA Pro 7.4
- Retain IDA Pro 7.3 support via old release taggin
- Convert all code to Python3 syntax
- Get all code style into the same format

- Clean up imports due to IDA Python changes

J.A.R.V.I.S - by: Carlos Garcia Prado

Plugin description

> A plugin for IDA Pro to assist you with the most common reversing tasks. It integrates with the (J.A.R.V.I.S) tracer.

readme for J.A.R.V.I.S

J.A.R.V.I.S.

FAQ

- Why *JARVIS* ?
  - *JARVIS* means "Just Another ReVersIng Suite" or whatever other bullshit you can think of :)
- What is it?
  - It is a small *bughunting* suite comprising three elements, namely:
  - A fuzzer (to be released)
  - A tracer based on INTEL PIN
  - A plugin for IDA Pro thought to assist you with the most common reversing tasks. It integrates with the tracer.
- Isn't there already something similar?
  - Yes, "curious minds often converge on the same idea" :) and by the way, there is nothing new under the sun. Now seriously, Alex and Daniel got it right with the code organization of their IDAScope plugin , so I used it as a skeleton for mine. Kudos to them!
- Why does the code suck so much?
  - Don't let physicists do computer science.

Installation

JARVIS is written entirely in Python and it is thought to be completely standalone. That means, although it runs within IDA, you can just copy its whole directory to anywhere you want. However, it is recommended (as a matter of convenience) to copy the contents of the IDAPlugin directory to IDA's plugins/

The auxiliary plugin *jarvis_launcher.py* registers a shortcut (Alt-J) which launches the actual plugin.

Dependencies

- Python 2.7.x (grab it here)
  - Recommended Python 2.7.9+ (includes pip)
- NetworkX (pip install networkx)
- IDA Pro :)

Graphical User Interface

JARVIS is written in PySide (Qt). It consists of a dockable Widget with several tabs , one for each different *category* .

There are three different kinds of widgets, namely:

- Table view (for example *calls within current function*)
- Tree view (for example *dword compares*)
- Text view (the *logging* at the bottom)

Binary Analysis

Functionality related to binary analysis (dough!) without any special emphasis in vulnerability finding.

At least I tried, sometimes the line between both is too thin...

The options available in this tab are:

- Show most referenced functions
- Show dword compares
  - Checks whether global variables are being compared to immediate values (binary-wide). These usually hold things like error codes or alike. Identifying and renaming them allow a better understanding of new code when found again.
- Mark immediate compares within the current function
  - It is helpful when analyzing something like a parser, for example.
- Show calls within the current function
  - When reversing large functions it is nice to have all information at once. For example, if at once all calls are UI related, we can probably just rename this function and move on.
- Show comments within the current function
- Show referenced strings within the current function
- Show input (file and network) connecting to the current function
  - Functions accepting any kind of input, for which there is a path connecting them and the current function.
- Show a list of all functions in the binary
  - This is no longer necessary, will probably disappear in next versions.

- Display connection graph (functions)
- Get all connections between functions accepting input and functions calling dangerous APIs
  - This is very CPU intensive, use with caution.
- Paint path(s) connecting two basic blocks
- XOR selected bytes
  - useful with malware / simple encryption schemes

Bug Hunting

Functionality specifically designed around bug hunting goes here.

There is not much at the moment...

- Detection of banned APIs (MS)
- Integer issues
  - Right now this is too x86 specific
  - Buggy and it gets confused sometimes by loops or alike. However, false positives are quickly detected and discarded by a human agent.
  - There will be false positives and negatives but as a rule of thumb you will need 10 minutes (tops) to go through them. On the bright side you will have found a handful of *potentially* exploitable code spots (stack, heap based buffer overflows).
  - Now you need to determine whether you can influence the parameters with your input.

Import & Export

This is for interacting with external tools.

- Import a call trace from a *PinTool*
- Export the trace to file in *GraphML* format
  - Import it to an external graph editor / viewer like Yed
- Import dynamic call resolution from a *PinTool*
- Export current function as hex
  - This is needed by some tools
- Export patched bytes to file
  - Creates a copy of the original file replacing the bytes patched in the IDA database.
  - Requested by m1k3 :)

Scratchpad

This is the killer feature :) You can even save your notes to a file!

Yes, I know IDA Pro already has this feature and saves the data to the IDB file but it looked convenient to have everything in one place.

Options

Runtime options controlling things like the amount of information being displayed, etc.

- Show unique strings
- Show unique comments
- Show unique calls
- Calculate entropy (for strings)
- Connect BB cutoff
  - The cutoff parameter for the algorithm calculating paths between basic blocks
- Connect functions cutoff
  - Analogous to the previous but for paths between functions

Other GUI augmentation

NOTE : a large amount of people still using an older version of IDA, for unknown reasons (cough, chinese guy, cough, australian company). Since the GUI features described below are based on changes introduced in version 6.7 of IDA, JARVIS was unable to start.

A quick workaround has been introduced to allow JARVIS to run in older versions of IDA, with limited functionality. However, the "connect functions" graph can still be used by selecting the origin and destination functions clicking the "show all functions" button and using the context menu (see screenshot below)

And now for the IDA 6.7+ users

The main goal of *JARVIS* is to get out of the way while adding some nice features which allow you to work faster. With this goal in mind, some *GUI augmentation* has been added for your clicking pleasure.

Calculating paths between basic blocks

A new context menu has been added to the *disassembly view (the graph thingy)* . Of course it is labeled *JARVIS* .

Simply right click the basic block on the graph view and select the appropiate menu entry. You will see some logging on IDA's *output window*

Once you got both start and end basic blocks selected, click the *connect basic blocks* button to get a list of all possible paths connecting them.

Double clicking the *Path ID* you will be presented with a color chooser widget where you can select a custom color to paint the choosen path on the graph view.

A penetrating yellow, for example, always does the trick :)

Voila! There you have your very visual path between basic blocks.

Calculating paths between functions

Similarly to the method explained above, *JARVIS* adds a new menu item to the popup for the *functions window*

The graph is of course *clickable* and will take you to the corresponding function in IDA's *disassembly view* by double clicking on a node.

PinTracer

The complementary tracing tool is Intel PIN based with a *PySide* GUI.

Since IDA Pro has a version of PySide itself, it is advised to install PySide in a *virtual environment* (read this)

To install it:

- Move the PinTracer folder to some location of your choice.
- Run the *install.bat* file. It will create a JARVISVE directory within PinTracer, create a virtual environment and install PySide
- That should be it :)

The batch script *run.bat* takes care of transparently preparing the virtual environment and running the PinTracer program.

About the PinTool itself. A version compiled for Windows 32 bits against Pin 71313 (vc12) is included. If you have another version of Pin, you will need to compile the PinTool against it. The source code is included (PinTracer/src_pintool/PinTracer.cpp) and the simplest way to compile it is to use the *MyPinTool* trick, well described here

---

idenLib - by: Noah.

Plugin description

> idenLib (Library Function Identification) plugin for IDA Pro

readme for idenLib

idenLib - Library Function Identification

When analyzing malware or 3rd party software, it's challenging to identify statically linked libraries and to understand what a function from the library is doing.

`idenLib.exe` is a tool for generating library signatures from `.lib` / `.obj` / `.exe` files.

`idenLib.dp32` / `idenLib.dp64` is a `x32dbg` / `x64dbg` plugin to identify library functions.

`idenLib.py` is an `IDA Pro` plugin to identify library functions.

Any feedback is greatly appreciated: @__qaz__qaz

How does idenLib.exe generate signatures?

1. Parses input file(`.lib` / `.obj` file) to get a list of function addresses and function names.
2. Gets the last opcode from each instruction
3. Compresses the signature with zstd
4. Saves the signature under the `SymEx` directory, if the input filename is `zlib.lib` , the output will be `zlib.lib.sig` or `zlib.lib.sig64` , if `zlib.lib.sig(64)` already exists under the `SymEx` directory from a previous execution or from the previous version of the library, the next execution will append different signatures. If you execute `idenLib.exe` several times with different version of the `.lib` file, the `.sig` / `sig64` file will include all unique function signatures.

Inside of a signature (it's compressed):

Usage:

- Generate library signatures: `idenLib.exe /path/to/file` or `idenLib.exe /path/to/directory`
- Generate `main` function signature: `idenLib.exe /path/to/pe -getmain`

Generating library signatures

`x32dbg` / `x64dbg` , `IDA Pro` plugin usage:

1. Copy `SymEx` directory under `x32dbg` / `x64dbg` / `IDA Pro` 's main directory
2. Apply signatures:

   `x32dbg` / `x64dbg` :

   `IDA Pro` :

Generating `main` function signature:

If you want to generate a signature for `main` function compiled using `MSVC 14` you need to create a `hello world` application with the corresponding compiler and use the application as input for `idenLib`

`main` function signature files are `EntryPointSignatures.sig` and `EntryPointSignatures.sig64`

Notes Regarding to `main` Function Signatures

- `idenLib` uses the `DIA APIs` to browse debug information stored in a PDB file. To run `idenLib` with `-getmain` parameter you will need to ensure that the msdia140.dll (found in `Microsoft Visual Studio\2017\Community\DIA SDK\bin`) is registered as a COM component, by invoking regsvr32.exe on the dll.

Applying Signatures

There are two ways to apply signatures, exact match and using `Jaccard index`

Useful links:

- Detailed information about `C Run-Time Libraries (CRT)`

Third-party

- Zydis (MIT License)
- Zstandard (BSD License)
- Icon by freepik

HexRayPytools - by: Igor Kirillov

Plugin description

> Assist in the creation of classes\/structures and detection of virtual tables.

readme for HexRayPytools

Plugin for IDA Pro

Table of Contents

About

The plugin assists in the creation of classes/structures and detection of virtual tables. It also facilitates transforming decompiler output faster and allows to do some stuff which is otherwise impossible.

Note : The plugin supports IDA Pro 7.x with Python 2/3.

Installation

Just copy `HexRaysPyTools.py` file and `HexRaysPyTools` directory to Ida plugins directory.

Configuration

Can be found at `IDADIR\cfg\HexRaysPyTools.cfg`

- `debug_message_level` . Set 10 if you have a bug and want to show the log along with the information about how it was encountered in the issue.
- `propagate_through_all_names` . Set `True` if you want to rename not only the default variables for the Propagate Name feature.
- `store_xrefs` . Specifies whether to store the cross-references collected during the decompilation phase inside the database. (Default - True)
- `scan_any_type` . Set `True` if you want to apply scanning to any variable type. By default, it is possible to scan only basic types like `DWORD` , `QWORD` , `void *` e t.c. and pointers to non-defined structure declarations.

Features

Recently added

Structure reconstruction

The reconstruction process usually comprises the following steps:

1. Open structure builder.
2. Find a local variable that points to the structure you would like to reconstruct.
3. Apply "Scan variable". It will collect the information about the fields that were accessed in the boundaries of one function. As an option, you can apply "Deep Scan variable", which will do the same thing but will also recursively visit other functions that has the same variable as its argument.
4. After applying steps 2 and 3 enough times, resolve conflicts in the structure builder and finalize structure creation. All the scanned variables will get a new type. Also, cross-references will be remembered and usable anytime.

Now, a few more details.

Structure Builder (Alt + F8)

The place where all the collected information about the scanned variables can be viewed and modified. Ways of collecting information:

- Right Click on a variable -> Scan Variable. Recognizes fields usage within the current function.
- Right Click on a variable -> Deep Scan Variable. First, recursively touches functions to make Ida recognize proper arguments (it happens only once for each function during a session). Then, it recursively applies the scanner to variables and functions, which get the structure pointer as their argument.
- Right Click on a function -> Deep Scan Returned Value. If you have the singleton pattern or the constructor is called in many places, it is possible to scan all the places, where a pointer to an object was recieved or an object was created.
- API [TODO]

Structure builder stores collected information and enables interaction:

- Types with the **BOLD** font are virtual tables. A double click opens the list with all virtual functions, which helps to visit them. The visited functions are marked with a cross and color:
- Types with the *ITALIC* font have been found as passed argument. It can help in finding substructures. [TODO]
- Double click on field `Name` or `Type` to edit.

- Double click on `Offset` opens a window with all the places, where this field has been extracted. Click the "Ok" button to open a selected place in the pseudocode window:

Buttons serve the following purpose:

Finalize - opens a window with an editable C-like declaration and assigns new types to all scanned variables.

Disable , Enable - are used for collision resolution.

Origin - switches the base offset which is used to produce new fields to structure (this value will be added to every offset of a newly-scanned variable, default = 0).

Array - renders a selected field as an array the size of which is automatically calculated.

Pack - creates and substitutes a substructure for selected items (collisions for these items should be resolved).

Unpack - dismembers a selected structure and adds all its fields to the builder.

Remove - removes the information about selected fields.

Clear - clears all.

Recognize Shape - looks for appropriates structure for selected fields.

Resolve Conflicts (new) - attempts to disable less meaningful fields in favor of more useful ones. (`char` > `_BYTE` , `SOCKET` > `_DWORD` etc). Doesn't help to find arrays.

Structure Cross-references (Ctrl + X)

With HexRaysPyTools, every time the F5 button is pressed and code is decompiled, the information about addressing to fields is stored inside cache. It can be retrieved with the "Field Xrefs" menu. So, it is better to apply reconstructed types to as many locations as possible to have more information about the way structures are used.

Note: IDA 7.4 has now an official implementation of this feature, available through Shift-X hotkey.

Guessing Allocation

Warning!! Very raw feature. The idea is to help find where a variable came from so as to run Deep Scan Process at the very top level and not to skip large amounts of code.

Structures with given size

Usage:

1. In Pseudocode viewer, right click on a number -> "Structures with this size". (hotkey "W")
2. Select a library to be looked for structures.
3. Select a structure. The Number will become `sizeof(Structure Name)` , and type will be imported to Local Types.

Recognition of structures by shapes

Helps find a suitable structure by the information gleaned from pseudocode after variable scanning.

Usage:

- *Method 1*
  i. Right click on a variable with -> Select "Recognize Shape".
  ii. Select Type Library.
  iii. Select structure.
  iv. Type of the variable will be changed automatically.
- *Method 2*
  i. Clear Structure Builder if it's currently used.
  ii. Right click on the variables that are supposed to be the same -> "Scan Variable".
  iii. Edit types (will be implemented later), disable or remove uninteresting fields, and click the "Recognize Shape" button.
  iv. You can select several fields and try to recognize their shapes. If found and selected, they will be replaced with a new structure.
  v. After final structure selection, types of all scanned variables will be changed automatically.

Disassembler code manipulations

Containing structures

Helps find containing structure and makes code prettier by replacing pointers with CONTAINING_RECORD macro

Before:

After:

Usage:

If a variable is a structure pointer and there's an access to outside of the boundaries of that structure, then:

1. Right click -> Select Containing Structure.
2. Select Type Library.

3. Select appropriate Structure and Offset.
4. If the result does not satisfy the requirements, then Right Click -> Reset Containing Structure and go back to step 1.

Function signature manipulation

1. Right click first line -> "Remove Return" converts return type to void (or from void to _DWORD).
2. Right click on argument -> "Remove Argument" disposes of this argument.
3. Right click on convention -> "Convert to ___usercall" switches to ___usercall or ___userpurge (same as ___usercall but the callee cleans the stack).

Recasting (Shift+R, Shift+L), Renaming (Shift+N, Ctrl+Shift+N)

Every time you have two sides in an expression, where each side may be a local or global variable, argument or return value of the function signature, it is possible to right-click or press the hotkey to give both sides of the expression similar types. Below, there is the table of possible conversions:

Original	Shift+L	Shift+R
var = (TYPE) expr	var type -> TYPE	
exp = (TYPE) var		var type -> TYPE
function(..., (TYPE) var, ...)	functions' argument -> TYPE	var type -> TYPE
(TYPE) function(...)		functions' return type -> TYPE
return (TYPE) var	functions' return type -> TYPE	var type -> TYPE
struct.field = (TYPE) var	type(field) -> TYPE	
pstruct->field = (TYPE) var	type(field) -> TYPE	

When you have an expression like `function(..., some_good_name, ...)`, you can rename function parameter.

When you have an expression like `function(..., v12, ...)`, and function has an appropriate parameter name, you can quickly apply this name to the variable.

Also possible to rename `vXX = v_named` into `_v_named = v_named` and vice versa.

And there's a feature for massive renaming functions using assert statements. If you find a function that looks like an assert, right-click the string argument with the function name and select "Rename as assert argument". All the functions where a call to assert statement has happened will be renamed (provided that there is no conflicts, either way, you'll see the warning in the output window)

Name Propagation (P)

This feature does the same recursive traversal over functions as the Deep Scan Variable does. But this time, all elements that have a connection with the selected one receive its name. It's possible to rename it or use names of both local and global variables, as well as structure members. By default, the plugin propagates names only over default names like `v1` , `a2` . See Configuration in order to change that.

Untangling 'if' statements

- Clicking `if` manually allows to switch `then` and `else` branches
- Automatically applies the following transformations:

Before:

```
...
if (condition) {
    statement_1;
    statement_2;
    ...
    return another_value;
}
return value;
```

After:

```
...
if (opposite_condition) {
    return value;
}
statement_1;
statement_2;
...
return another_value;            // if 'then' branch has no return, than `return value;`
```

Classes

Also, it can be found at *View->Open Subview->Classes* . Helps to manage classes (structures with virtual tables).

!! Better to rename all functions before debugging, because Ida can mess up default names, and the information in virtual tables will be inconsistent. Class, virtual tables, and functions names are editable. Also a function's declaration can be edited. After editting, the altered items change font to *italic* . Right click opens the following menu options:

- Expand All / Collapse All.

- Refresh - clear all and rescan local types for information again.
- Rollback - undo changes.
- Commit - apply changes. Functions will be renamed and recasted both in virtual tables in Local Types and disassembly code.
- Set First Argument type - allows selecting the first argument for a function among all classes. If right click was used on class name, then its type will be automatically applied to the virtual table at offset 0.

You can also filter classes using Regexp either by class_name or by existence of specific functions. Simply input an expression in line edit for filtering by class_name or prepend it with "!" to filter by function name.

Structure Graph

Shows relationship between structures:

Also: dark green node is union, light green - enum.

Usage:

1. Open Local Types.
2. Select structures and right click -> "Show Graph" (Hotkey "G").
3. Plugin creates a graph of all structures that have relationship with selected items.
4. Double clicking on a node recalculates the graph for it.
5. Every node has a hint message that shows C-like typedef.

API

Under construction

Presentations

- ZeroNights 2016
- Insomni'hack 2018
- Infosec in the City 2018 (Slides)

Heap Viewer - by: Daniel García

Plugin description

> Used to examine the glibc heap, focused on exploit development.

readme for Heap Viewer

HeapViewer

An IDA Pro plugin to examine the heap, focused on exploit development.

Currently supports the glibc malloc implementation (ptmalloc2).

3rd place winner of the 2018 Hex-Rays Plugin Contest

Requirements

- IDA Pro >= 7.0

Tested on

- glibc 2.23 <= 2.31 (x86, x64)

Features

- Heap tracer (malloc/free/calloc/realloc)
  - Detection of overlaps and double-frees
  - Visualization using villoc
- Malloc chunk info
- Chunk editor
- Multi-arena info (chunks, top, last-remainder)
- Bins info (fastbins, unsortedbin, smallbins y largebins)
- Tcache info (glibc >= 2.26)
- GraphView for linked lists (bins/tcache)
- Structs view (malloc_state / malloc_par / tcache_perthread)
- Magic utils:
  - Unlink merge info
  - Freeable/merge info
  - Fake fastbin finder
  - House of force helper
  - Useful libc offsets
  - Calc chunk size (request2size)
  - IO_FILE structs

Install

Just drop the `heap_viewer.py` file and the `heap_viewer` folder into IDA's plugin directory.

To install just for the current user, copy the files into one of these directories:

OS	Plugin path
Linux/macOS	`~/.idapro/plugins`
Windows	`%AppData%\Hex-Rays\IDA Pro\plugins`

Configuration

Currently the plugin does not require to be configured, since it tries to obtain automatically the necessary offsets to analyze the heap.

However, in case the plugin fails, it is possible to set the different offsets in the configuration tab. To obtain these offsets, you can use any of the tools located in the `utils` folder.

If you find any inconsistency, let me know :)

Screenshots

Tracer

Arena & chunk info

Tcache entries

Bins

Bin graph

Fastbin graph

Tcache graph

Find fake fastbin

Unlink merge info

Useful libc offsets

Learning Resources

I'd recommend the following resources alongside this tool for learning heap exploiting.

- shellphish's how2heap

Author

- Daniel García Gutiérrez - @danigargu

Contributors ?

Special mention to my colleagues soez , wagiro and DiaLluvioso for give me some ideas during the development of the plugin. And of course, the @pwndbg project, from which I picked up some things about heap parsing.

Contributors

Bugs / Feedback / PRs

Any comment, issue or pull request will be highly appreciated :-)

Driver Buddy - by: NCC Group Plc

Plugin description

> It assists with the reverse engineering of Windows kernel drivers.

readme for Driver Buddy

Quickstart

DriverBuddy Installation Instructions

1. Copy DriverBuddy folder and DriverBuddy.py file into the IDA plugins folder C:\Program Files (x86)\IDA 6.8\plugins or wherever you installed IDA

DriverBuddy Usage Instructions

1. Start IDA and open a Windows kernel driver
2. Go to Edit->Plugins and select Driver Buddy or press ctrl-alt-d
3. Check Output window for DriverBuddy analysis results
4. To decode IOCTLs, highlight the suspected IOCTL and press ctrl-alt-i

DriverBuddy

DriverBuddy is an IDAPython plugin that helps automate some of the tedium surrounding the reverse engineering of Windows Kernel Drivers. It has a number of handy features, such as:

- Identifying the type of driver
- Locating DispatchDeviceControl and DispatchInternalDeviceControl functions
- Populating common structs for WDF and WDM drivers
  - Attempts to identify and label structs like the IRP and IO_STACK_LOCATION
  - Labels calls to WDF functions that would normally be unlabeled
- Finding known IOCTL codes and decoding them
- Flagging functions prone to misuse

Finding DispatchDeviceControl

Being able to automatically locate and identify the DispatchDeviceControl function is a time saving task during driver reverse engineering. This function is used to route all incoming DeviceIoControl codes to the specific driver function associated with that code. Automatically identifying this function makes finding the valid DeviceIoControl codes for each driver much quicker. Additionally, when investigating possible vulnerabilities in a driver due to a crash, knowing the location of this function helps narrow the focus to the specific function call associated with the crashing DeviceIoControl code.

Labeling WDM Structs

Several driver structures are shared among all WDM drivers. Being able to automatically identify these structures, such as the IO_STACK_LOCATION, IRP, and DeviceObject structures, can help save time during the reverse engineering process. DriverBuddy attempts to locate and identify many of these structs.

Labeling WDF Functions

As with WDM drivers, there are several functions and structures that are shared among all WDF drivers. Automatically identifying these functions and structures will save time during the reverse engineering process and provide context to unindentified areas of the driver where these functions are in use.

Decoding DeviceIoControl Codes

While reversing drivers, it is common to come across IOCTL codes as part of the analysis. These codes, when decoded, reveal useful information to reverse engineers and may draw focus to specific parts of the driver where vulnerabilities are more likely to exist.

Future things:

1. Add obref and deref checks of some sort to help find refcount issues
2. Polish output, gui?
3. Strengthen/polish current features
    - Improve reliablity of DispatchDeviceControl finder
    - Write short blurbs about why things are flagged
    - MSDN doc importer

Stretch Goals:

1. Find IOCTLs automatically
2. IRP taint analysis aka follow aliasing of sysbuf/inbuf, size
3. Identify other common structures
4. Uninitialized variables, etc

Credits

- We are using Satoshi Tanda's IOCTL decoder, originally found here https://github.com/tandasat/WinIoCtlDecoder .
- The WDF functions struct is based on Red Plait's work (http://redplait.blogspot.ru/2012/12/wdffunctionsidc.html) and was ported to IDA Python by Nicolas Guigo, later updated by us.

License

This software is released under the MIT License, see LICENSE.

IDASignSrch - by: Sirmabus

Plugin description

> It can recognize tons of compression, multimedia and encryption algorithms and many other things like known strings and anti-debugging code.

readme for IDASignSrch

IDA Pro plug-in conversion of Luigi Auriemma's signsrch signature matching tool.

* Deprecated, will no longer be updated, please see my use my superior YARA for IDA plugin here:
https://github.com/kweatherman/yara4ida

* July 2018, updated to IDA 7.1

Luigi's original signsrch description:
"Tool for searching signatures inside files, extremely useful as help in reversing jobs like figuring or having an initial idea of what encryption/-compression algorithm is used for a proprietary protocol or file.
It can recognize tons of compression, multimedia and encryption algorithms and
many other things like known strings and anti-debugging code which can be also
manually added since it's all based on a text signature file read at run-time and easy to modify."

Yaco - by: DGA\/MI SSI

Plugin description

> Collaborative Reverse-Engineering for IDA.

readme for Yaco

YaCo - Collaborative Reverse-Engineering for IDA

YaDiff - Symbols Propagation between IDA databases

Latest Releases

Yaco

YaCo is a Hex-Rays IDA plugin enabling collaborative reverse-engineering on IDA databases for multiple users. Incremental database changes are stored & synchronized through Git distributed version control system. Both offline & online work is supported.

Motivation

IDA does not allow multiple users to work on the same binary. During large malware analysis, we had to use a team of reversers and manual synchronization is a tedious and error-prone process.

YaCo goals are:

- Support all IDA events
- Be fast, users must not wait for synchronisation events
- Prevent conflicts between users
- Be user-friendly & easy to install

YaDiff

YaDiff is a standalone command-line tool allowing symbol, comment, enum & struct propagation between distinct IDA databases.

Motivation

There are two major use cases for YaDiff

- Merging previously-analyzed binary symbols into an updated binary
- Merging debug symbols from an external library into another stripped binary

Usage

- Uncompress the release into a directory
- Put each of your two IDA databases in a different directory
- Call merge_idb.py on those two databases

```
python $yatools_directory/YaTools/bin/merge_idb.py $source_dir/source.idb $destination_dir/d
```

- Open `$destination_dir/destination.yadiff_local.idb` and check results

Installation

Debian stretch/x64

YaTools is 64-bit only, like IDA 7.1.

Install dependencies

```
sudo apt install build-essential git cmake python2.7 libpython2.7 libpython2.7-dev
```

Configure & build YaTools

```
export NUM_CPU=4
export IDA_DIR=/opt/ida7.1/
export IDASDK_DIR=/opt/idasdk71/
$yaco/build> ./configure.sh
$yaco/out/x64_RelWithDebInfo> make -j $NUM_CPU
$yaco/out/x64_RelWithDebInfo> ctest . --output-on-failure -C RelWithDebInfo -j $NUM_CPU
$IDA_DIR/plugins> $yaco/build/deploy.sh
```

Windows

CMake & Python 2.7 64-bit must be installed and in the PATH Only visual studio 2017 is currently supported

Configure and build YaTools

```
# export directories without quotes
set NUM_CPU=4
set IDA_DIR=C:\Program Files\IDA Pro 7.1
set IDASDK_DIR=C:\idasdk71
$yaco/build> configure_2017.cmd
$yaco/out/x64> cmake --build . --config RelWithDebInfo
```

```
$yaco/out/x64> ctest . --output-on-failure -C RelWithDebInfo -j %NUM_CPU%
$IDA_DIR/plugins> $yaco/build/deploy.cmd
```

YaCo Usage

First user

To create the YaCo environment:

1. open binary or idb file as usual
2. click on Edit menu, Plugins, YaCo
3. enter path to Git remote (could be a file system path, or empty to use current dir)
4. a warning will inform you that IDA have to be re-launch with correct idb
5. IDA auto close
6. launch IDA for your FILE_local.idb file
7. save database
8. start working as usual

Warning, in order to use it with multiple users, YaCo project must be in a bare Git repository.

Other users

Setup YaCo environment:

1. clone a YaCo project
2. open idb/i64 file with ida
3. click on Edit menu, Plugins, YaCo
4. a warning will inform you that IDA has to be re-launched with correct idb
5. IDA auto close
6. launch IDA for your FILE_local.idb file
7. save database
8. start working as usual

How it works

YaCo uses a Git server to synchronize changes between users.

In the local repository, YaCo stores the original IDB and incremental changes as xml files & commits.

Note that the database is not modified anymore unless you force a synchronisation. When saving the database, we fetch remote changes, rebase local changes on top of those, import this new state into IDA and push this state to the remote Git server.

Any Git server should work, for example github, gitlab or gitea instances. Note that some Git hosts have a file size limit, which can be an issue for large IDB files. See #13 .

Currently, YaCo only supports SSH authentication. To keep the plugin user-friendly, there is no mechanism asking for passwords & passphrases on every Git operation. Instead, it is recommended to use an ssh agent, like pageant under windows or ssh-agent under linux.

How to use YaCo with git & SSH on Windows

You have a git repository, on github or equivalent

- Clone the git repository `git clone git@server_name:group_name/repo_name.git`
- Import/generate your SSH key with puttygen.exe & save the private key into a ppk file
- Associate your public key in your server profile
- Start pageant.exe & add your ppk key
- Try to connect once using bare SSH on your server with plink `plink git@server_name:group_name/repo_name.git`
- It should connect & exit immediately
- Copy the file.idb into file_local.idb in the repository
- Open file_local.idb with IDA & check YaCo is properly starting
- Save the file once & check whether YaCo is able to fetch, rebase & push on the git server

Contributors

- Benoît Amiaux
- Frederic Grelot
- Jeremy Bouetard
- Martin Tourneboeuf
- Maxime Pinard
- Valerian Comiti

License

YaCo is licensed under the GNU General Public License v3.0 that can be found in the LICENSE file

---

Diaphora - by: Joxean

Plugin description

It is a program diffing plugin for IDA, similar to Zynamics Bindiff.

readme for Diaphora

Diaphora

Diaphora (, Greek for 'difference') version 2.0 is the most advanced program diffing tool, working as an IDA plugin, available as of today (2022). It was released first during SyScan 2015 and is actively maintained.

Diaphora supports IDA 6.9 to 8.0, but the main branch has support only for IDA >= 7.4 because the code only runs in Python 3.X. If you are looking for an IDA >= 7.4 port with support for Python 2.X, check this issue .

Support for Ghidra is in development, but it will take very long. Support for Binary Ninja is also planned but will probably come after Ghidra's port. If you are looking for Radare2 support, you can check this very old fork .

For more details, please check the tutorial in the "doc" directory.

NOTE: If you're looking for a tool for diffing or matching functions between binaries and source codes, you might want to take a look to Pigaios .

Unique Features

Diaphora has many of the most common program diffing (bindiffing) techniques you might expect, like:

- Diffing assembler.
- Diffing control flow graphs.
- Porting symbol names and comments.
- Addig manual matches.
- Similarity ratio calculation.
- Batch automation.
- Call graph matching calculation.
- Dozens of heuristics based on graph theory, assembler, bytes, functions' features, etc...

However, Diaphora has also many features that are unique, not available in any other public tool. The following is a non extensive list of unique features:

- Parallel diffing.
- Pseudo-code based heuristics.
- Pseudo-code patches generation.
- Ability to port structs, enums and typedefs.
- Diffing pseudo-codes (with syntax highlighting!).

- Scripting support (for both the exporting and diffing processes).
- ...

It's also actively maintained, and the following is a list of the features that are 'in the making':

- Support for compilation units (finding and diffing compilation units).
- Direct integration with Pigaios .
- 'Machine Learning' based techniques so reverse engineers can teach Diaphora what is a good match or a bad one, and how to search for more.

Python 2.7 and IDA versions 6.95 to 7.3

TLDR: if you're looking for a version of Diaphora supporting Python 2.X and IDA versions 6.95 to 7.3, check this release or this branch .

Since IDA 7.4, Diaphora will only support Python 3.X. It means that the code in Github will only run in IDA 7.4 and Python 3.X. I've tried to make it compatible but it caused the code to be horrible and unmaintainable. As so, I've decided that it was best to drop support for Python 2.X and IDA versions <= 7.3 and focus in Python 3.X and IDA versions >= 7.4.

Donations

You can help (or thank) the author of Diaphora by making a donation, if you feel like doing so:

License

Versions of Diaphora prior to 1.2.4, including version 1.2.4, are licensed under the GNU GPL version 3 . Since version 2.0, Diaphora is now licensed under the GNU Affero GPL version 3 license . The license has been changed so companies wanting to modify and adapt Diaphora cannot offer web services based on these modified versions without contributing back the changes.

For 99.99% of users, the license change doesn't affect them at all. If your company needs a different licensing model, check the next section...

Licensing

Commercial licenses of Diaphora are available. Please contact admin@joxeankoret.com for more details.

Documentation

You can check the tutorial https://github.com/joxeankoret/diaphora/blob/master/doc/diaphora_help.pdf

Screenshots

This is a screenshot of Diaphora diffing the PEGASUS iOS kernel Vulnerability fixed in iOS 9.3.5:

And this is an old screenshot of Diaphora diffing the Microsoft bulletin MS15-034 :

These are some screenshots of Diaphora diffing the Microsoft bulletin MS15-050 , extracted from the blog post Analyzing MS15-050 With Diaphora from Alex Ionescu.

Here is a screenshot of Diaphora diffing iBoot from iOS 10.3.3 against iOS 11.0 :

IDA ARM Highlight - by: Guillaume

Plugin description

> Highlighting and decoding ARM system instructions.

readme for IDA ARM Highlight

Decoding ARM system instructions

This script will give you the list of ARM system instructions used in your IDA database. This is useful for locating specific low-level pieces of code (setting up the MMU, caches, fault handlers, etc.).

One hassle of reverse engineering low-level ARM code is that IDA Pro does not decode the internal registers accessed by co-processor instructions (`MCR` / `MRC` and `MSR` / `MRS` on AArch64).

After applying the script, the system registers accessed will be automatically commented in the database, as defined in the official ARM reference manuals.

The script will also try to automatically detect the accessed fields for some registers:

Usage

`Alt-F7` in IDA Pro, then run the script on your open database.

Compatibility

Should work with ARMv7 and ARMv8 processors.

IDA for Delphi - by: Coldzer0

Plugin description

> IDA Python Script to Get All function names from Event Constructor (VCL).

readme for IDA for Delphi

IDA-For-Delphi

IDA Python Script to Get All function names from Event Constructor

New Update

Now it supports x64

+ at first IDA didn't recognise it as Delphi file

+ now select the local debugger

+ then load the script file

+ it will load and Stop at EP

+ hit the Greeeen button or [F9] :p

+ now we have all the Functions named and have a BP [on]

the only issue here is it needs to run the file & the file to be unpacked ?

but if u can get the pattern address on unpacked file on memory it will work fine .

Peace

IDA Patcher - by: Peter Kacherginsky

Plugin description

> It is designed to enhance IDA's ability to patch binary files and memory.

readme for IDA Patcher

```
Welcome to IDA Patcher, a binary and memory patch management plugin for Hex-Ray's IDA Pro di

To install IDA Patcher simply copy 'idapatcher.py' to IDA's plugins folder.
The plugin will be automatically loaded the next time you start IDA Pro.

You can find the latest IDA Patcher version and documentation here:
    http://thesprawl.org/projects/ida-patcher/

Happy patching!
  -Peter Kacherginsky <iphelix@thesprawl.org>
```

IDATropy - by: Daniel García

Plugin description

> It is designed to generate charts of entropy and histograms using the power of idapython and matplotlib.

readme for IDATropy

IDAtropy

IDAtropy is a plugin for Hex-Ray's IDA Pro designed to generate charts of entropy and histograms using the power of idapython and matplotlib.

DEPENDENCIES

IDAtropy requires the matplotlib python library:

```
pip install matplotlib
```

The current version of IDAtropy only runs in Python 3.X and IDA >= 7.4. If you want an older version with support for Python 2.X and IDA < 7.4, check this release .

INSTALLATION

Simply, copy `IDAtropy.py` to the IDA's plugins folder.

To install just for the current user, copy the file into one of these directories:

OS	Plugin path
Linux/macOS	`~/.idapro/plugins`
Windows	`%AppData%\Hex-Rays\IDA Pro\plugins`

SCREENSHOTS

Plugin options

Entropy - All segments

Histogram

Entropy on-click

Xrefs finder

CONTACT

Any comment or pull request will be highly appreciated :-)

IDA Sploiter - by: Peter Kacherginsky

Plugin description

An exploit development and vulnerability research plugin.

readme for IDA Sploiter

```
Welcome to IDA Sploiter, an exploit development and vulnerability research
plugin for Hex-Ray's IDA Pro disassembler.

To install IDA Sploiter simply copy all the python files to IDA's plugins folder.
The plugin will be automatically loaded the next time you start IDA Pro.

IDA Sploiter currently supports the following architectures:
  - x86/amd64
```

```
  - PowerPC

You can find the latest IDA Sploiter version and documentation here:
    http://thesprawl.org/projects/ida-sploiter/

Happy sploiting!
  -Peter Kacherginsky <iphelix@thesprawl.org>
```

IDA IPython - by: james91b

Plugin description

> An IDA Pro Plugin for embedding an IPython.

readme for IDA IPython

What's New

- Improve python plugin load order (prevents crashes when python does not load correctly)
- Add in safe IDA process termination (Special thanks to @tmr232 for this)

What and Why?

This is a plugin to embed an IPython kernel in IDA Pro. The Python ecosystem has amazing libraries (and communities) for scientific computing. IPython itself is great for exploratory data analysis. Using tools such as the IPython notebook make it easy to share code and explanations with rich media. IPython makes using IDAPython and interacting with IDA programmatically really fun and easy.

Example Uses

QT Console

You can just use IPython qtconsole for a better interactive python shell for IDA.

You can also use the QT console to graph things. This is an example creating a bar chart for the occurrences of each instruction mnemonic in a function (in notepad.exe).

Notebooks

Another useful case is using IPython notebooks.

- Function Entropy - Here is an example where we compute the entropy (using scipy stats module) of each function in notepad.exe and graph the result.
- Cython and IDA - Here is an example where we use the cython cell magic to call IDA Api's that are not exposed via IDAPython.
- Sark Snapshots - Example of screen snapshots using Sark.

More examples..soon...

How the plugin works

IDA is predominantly single threaded application, so we cannot safely run the kernel in a separate thread. So instead of using another thread a hook is created on the QT process events function and the `do_one_iteration` method of the ipython kernel is executed each frame.

Installation

I suggest using the Anaconda distribution of Python as it comes with all the required python libraries pre-built and installed. To get IDA to use Anaconda, simply set the PYTHONHOME enviroment variable. Alternatively you can install IPython and the dependencies separately.

This plugin should work on all 6.X x86 QT versions of IDA on Windows.

Basic Installation and QTConsole

1. Download and extract the release
2. Copy the contents of the `plugins` and `python` directories under IDA's installation directory.
3. Launch IDA.
4. At the command line (Windows), start an IPython qtconsole with the kernel instance (outputted in the IDA console) e.g `ipython qtconsole --existing kernel-4264.json`

Using the Notebook

1. Copy `idc` directory to your IDA directory. (the `nothing.idc` script is used to pass command line parameters to the plugin)
2. Change the paths to the `idaq.exe` and `idaq64.exe` executables in the `kernel.json` under the `notebook\kernels\ida32` and `notebook\kernels\ida64` directories respectively

3. Install the kernels using `jupyter-kernelspec install` (e.g. `jupyter-kernelspec install --user notebook\kernels\ida64`)
4. When starting a notebook, choose the `IDA32` or `IDA64` kernels, depending on your desired IDA version.

How to Build

1. Install cmake
2. At the command line cd to the root directory and run the following
3. `mkdir build`
4. `cd build`
5. `cmake -G "Visual Studio 11" -DPYTHON_DIR="<YOUR_PYTHON_DIR>" -DIDA_SDK="<YOUR_IDASDK_LOCATION>" -DIDA_DIR="<YOUR_IDA_DIRECTORY>" ..` e.g. `cmake -G "Visual Studio 11" -DPYTHON_DIR="C:\Anaconda" -DIDA_SDK="C:\dev\IDA\idasdks\idasdk64" -DIDA_DIR="C:/Program Files (x86)/IDA 6.4" ..`
6. `cmake --build . --config Release`

So far only tested with "Visual Studio 11" compiler.

Changelog

0.5

- Improve python plugin load order (prevents crashes when python does not load correctly)
- Add in safe IDA process termination (Special thanks to @tmr232 for this)

0.4

- Added IDA 6.9 support

0.3

- 2015-10-04: Running the plugin from the plugins menu or view menu will now launch an IPython QTConsole. Consoles are terminated on exit.
- 2015-10-04: Only capture standard output and error when executing the kernel
- 2015-10-21: Improve error reporting

0.2

- 2015-09-30: Added support for Jupyter (replaces original support for IPython).
- 2015-10-01: Added support for ida64.

0.1

- First release

To do/Future Ideas

- More examples
- Create a library for cell/line magic functions specific to IDA

IDA Skins - by: zyantific

Plugin description

> Plugin providing advanced skinning support for IDA Pro utilizing Qt stylesheets, similar to CSS.

readme for IDA Skins

IDA Skins

Plugin providing advanced skinning support for IDA Pro utilizing Qt stylesheets , similar to CSS.

Important: This plugin is now deprecated

IDA 7.3 added theming support directly into IDA itself and ships with a forked version of the `ISASkins dark` theme. We will keep maintaining this plugin for a while for users of old IDA verions, but it will be shelved eventually. New themes should be developed for IDA directly.

Screenshot

`VSCode dark` **theme**

`ISASkins dark` **theme**

Download

Download the latest version from GitHub

Installation

Copy the contents of the `plugins` directory into the `plugins` directory of your IDA installation.

Theming

Theming IDA using IDASkins works using Qt stylesheets . For information on the most important IDA-specific UI elements, take a look in the enclosed default `stylesheet.qss` . Pull-requests for new themes are very welcome!

Idadiff - by: Adrien C.

Plugin description

> A diffing tool using Machoc Hash.

readme for Idadiff

idadiff

IDAPython script in order to auto-rename sub.

The script uses the @Heurs MACHOC algorithm (https://github.com/ANSSI-FR/polichombr) in order to build tiny CFG hashes of a source binary sample in IDA PRO. These hashes may be compared against the ones in the destination binary sample. If a 1-1 relationship is found, the sub is renamed.

TODO:

- code cleaning;
- N grams (3 and 5);
- use a %temp% or /tmp file to share the hashes;
- other methods.

IDA_EA - by: Joe Darbyshire

Plugin description

> A set of exploitation\/reversing aids for IDA.

readme for IDA_EA

IDA EA

- A set of exploitation/reversing aids for IDA

Features

Context Viewer

New context viewer for IDA, Features include:

- Recursive pointer derfereneces
- History browser
- Color coded memory
- Instruction rewind feature
- A similar interface to that of popular `GDB` plugings (eg. `PEDA/GEF`)

Instuction Emulator

- Live annotate the results if furture instructions in IDA using the `Unicorn` CPU emulator
- Can be hooked to breakpoints
- Visualise instructions before execution

Heap Explorer

Explore current heap state of glibc binaries

- Trace allocations
- Enumerate bins
- View all free and allocated chunks headers
- Useful for heap exploitation / debugging.

Trace Dumper

- Dump the results of an IDA trace into a Pandas Dataframe
- Analyze traces in Python using Pandas

CMD

- GDB bindings for IDA
- GDB style mem queries + searches

Restyle

- Restyle IDA using GUI.

Install

Dependencies

No core dependencies for the plugin. Nevertheless certain fetures will be disabled without these python libraries installed:

Trace Dumper

- `Pandas`

Instruction Emulator

- `Unicorn CPU emulator`
- `Capstone Dissasembler`

Install

- Place `ida_ea` folder in `IDA Pro` directory (`C:\Users\{name}\AppData\Roaming\Hex-Rays\IDA Pro` on Windows)
- Add line `from ida_ea import ea_main` to your `idapythonrc` file.
- Plugin is accessed via `IDA EA` tab added to the menu bar

Warning

- Only tested on Windows with `IDA 6.8`
- Only supports `x86/x86-64` binaries
- Alpha release so expect many bugs!

Enjoy!

---

idaemu - by: czl

Plugin description

Use for emulating code in IDA Pro. It is based on unicorn-engine.

readme for idaemu

idaemu

idaemu is an IDA Pro Plugin - use for emulating code in IDA Pro. It is based on unicorn-engine .

Support architecture:

- X86 (16, 32, 64-bit)
- ARM
- ARM64 (ARMv8)
- MIPS (developing)

Install

If you want to use idaemu, you have to install unicorn-engine and unicorn's python binding first. Then use the `idaemu.py` as the idapython script.

License

This project is released under the GPL license .

Example1

This is easy function for add.

```
.text:000000000040052D                public myadd
.text:000000000040052D myadd          proc near               ; CODE XREF: main+1B p
.text:000000000040052D
.text:000000000040052D var_4          = dword ptr -4
.text:000000000040052D
.text:000000000040052D                push    rbp
.text:000000000040052E                mov     rbp, rsp
.text:0000000000400531                mov     [rbp+var_4], edi
.text:0000000000400534                mov     edx, cs:magic    ; magic dd 64h
```

```
.text:000000000040053A                  mov     eax, [rbp+var_4]
.text:000000000040053D                  add     eax, edx
.text:000000000040053F                  pop     rbp
.text:0000000000400540                  retn
.text:0000000000400540 myadd            endp
```

Running the idapython scritp:

```
from idaemu import *
a = Emu(UC_ARCH_X86, UC_MODE_64)
print a.eFunc(0x040052D, None, [7])
```

Get the function result:

```
107
```

Example2

If there is a library function call inner the function, we couldn't call it directly.
We should use `alt` to hook the library function first.

```
.text:0000000000400560                  public myadd
.text:0000000000400560 myadd            proc near               ; CODE XREF: main+27 p
.text:0000000000400560
.text:0000000000400560 var_8            = dword ptr -8
.text:0000000000400560 var_4            = dword ptr -4
.text:0000000000400560
.text:0000000000400560                  push    rbp
.text:0000000000400561                  mov     rbp, rsp
.text:0000000000400564                  sub     rsp, 10h
.text:0000000000400568                  mov     [rbp+var_4], edi
.text:000000000040056B                  mov     [rbp+var_8], esi
.text:000000000040056E                  mov     eax, [rbp+var_8]
.text:0000000000400571                  mov     edx, [rbp+var_4]
.text:0000000000400574                  add     eax, edx
.text:0000000000400576                  mov     esi, eax
.text:0000000000400578                  mov     edi, offset format ; "a+b=%d\n"
.text:000000000040057D                  mov     eax, 0
.text:0000000000400582                  call    _printf
.text:0000000000400587                  leave
.text:0000000000400588                  retn
.text:0000000000400588 myadd            endp
```

Running the idapython scritp:

```
from idaemu import *

a = Emu(UC_ARCH_X86, UC_MODE_64)
```

```
def myprint(uc, out, args):
    out.append("this is hook output: %d" % args[1])
    return 0

myadd_addr = 0x00400560
printf_addr = 0x00400410
a.alt(printf_addr, myprint, 2, False)
a.eFunc(myadd_addr, None, [1, 7])
print "---- below is the trace ----"
a.showTrace()
```

Get the result:

```
---- below is the trace ----
this is hook output: 8
```

Well Done. We can alter every function in this way.

Example3

Sometimes it emulates fail with some abort:

```
Python>from idaemu import *
Python>a = Emu(UC_ARCH_ARM, UC_MODE_THUMB)
Python>print a.eFunc(here(), 0xbeae, [4])
#ERROR: Invalid instruction (UC_ERR_INSN_INVALID)
1048576
```

Then we can use `setTrace` and `showTrace` for debugging.

```
Python>from idaemu import *
Python>a = Emu(UC_ARCH_ARM, UC_MODE_THUMB)
Python>a.setTrace(TRACE_CODE)
Python>a.eFunc(here(), 0xbeae, [4])
#ERROR: Invalid instruction (UC_ERR_INSN_INVALID)
1048576
Python>a.showTrace()
### Trace Instruction at 0x13dc, size = 2
### Trace Instruction at 0x13de, size = 2
### Trace Instruction at 0x13e0, size = 2
......
### Trace Instruction at 0x19c6, size = 2
### Trace Instruction at 0x19c8, size = 2
### Trace Instruction at 0x19ca, size = 2
### Trace Instruction at 0xbeae, size = 2
```

So we found the abort reason (the RA is wrong)

LazyIDA - by: Lays

Plugin description

> Add functionalities such as function return removing, converting data, scanning for string vulnerabilities.

readme for LazyIDA

LazyIDA

Make your IDA Lazy!

Install

1. put `LazyIDA.py` into `plugins` folder under your IDA Pro installation path.

Features

- Remove function return type in Hex-Rays:
- Convert data into different formats, output will also be automatically copied to the clipboard:
- Scan for format string vulnerabilities:
- Jump to vtable functions by double clicking
- Lazy shortcuts:
  - Disasm Window:
    - `w` : Copy address of current line into clipboard
  - Hex-rays Window:
    - `w` : Copy address of current item into clipboard
    - `c` : Copy name of current item into clipboard
    - `v` : Remove return type of current item

HexRaysCodeXplorer - by: Alexander Matrosov, Eugene Rodionov, Rodrigo Branco & Gabriel Barbosa

Plugin description

> The Hex-Rays Decompiler plugin for better code navigation in RE process. CodeXplorer automates code REconstruction of C++ applications or modern malware like Stuxnet, Flame, Equation, Animal Farm …

readme for HexRaysCodeXplorer

IFL - by: Hasherezade

Plugin description

> IFL, the Interactive Functions List It's goal is to provide user-friendly way to navigate between the functions and their references.

readme for IFL

IFL - Interactive Functions List

License: CC-BY (https://creativecommons.org/licenses/by/3.0/)

A small plugin with a goal to provide user-friendly way to navigate between functions and their references.
Additionally, it allows to import reports generated by i.e. PE-sieve into IDA.
Supports:

- `.tag` format (generated by PE-sieve , Tiny Tracer , PE-bear)
- `.imports.txt` format (generated by PE-sieve)

For Python 2 version check the branch python2

Demo

Dark theme:

Light theme:

IDARef - by: Mikhail Sosonkin

Plugin description

> IDA Pro Full Instruction Reference Plugin

readme for IDARef

IdaRef

IDA Pro Full Instruction Reference Plugin - It's like auto-comments but useful.

I'm generally pretty good at figuring out what various Intel instructions do. But, once in a while I need to either know some precise detail (i.e. exact side effects of SUB) or come across a rare instruction. Then I break my train of thought and have to dig out the reference manual. Which got me thinking: *Why can't IDA just give me the full documentation?*

Enter IdaRef: The plugin will monitor the location for your cursor (ScreenEA) and display the full documentation of the instruction. At the moment it only supports x86-64, ARM and MIPS 32bit, however adding support for other architectures is relatively easy.

Usage

Simply checkout or download the repository and install it to your IDA plugins directory:

```
idaref.py -> <ida_path>/plugins/idaref.py
arm.sql -> <ida_path>/plugins/archs/arm.sql
x86-64.sql -> <ida_path>/plugins/archs/x86-64.sql
mips32.sql -> <ida_path>/plugins/archs/mips32.sql
xtensa.sql -> <ida_path>/plugins/archs/xtensa.sql
```

You can also use the installer.sh file but you'll need to open it and edit the IDA path if you're not using Mac OS and IDA 6.8.

Once loaded, the plugin can be turned ON by going to Edit/Start IdaRef menu option. To control the output right-click on the tab window to get a menu:

- Update View - Load documentation for currectly selected instruction.
- Lookup Instruction - Manual load documentation, you'll be prompted for the instruction.
- Toggle Auto-refresh - Turn on/off auto loading of documentation and rely on the first two options.

Internals

Upon loading the script will look for SQlite databases in the same directory as the itself. The naming convention for the database files is [arch name].sql. The

[arch name] will be presented to the user as choice.

The database has a table called 'instructions' and two columns called 'mnem' and 'description'. The instructions are looked up case insensitive (upper case) by the mnem value. The text from description is displayed verbatim in the view.

To add support for more architectures simply create a new database with those columns and place it in the the script directory.

```
import sqlite3 as sq
con = sq.connect("asm.sqlite")
con.text_factory = str
cur = con.cursor()
cur.execute("CREATE TABLE IF NOT EXISTS instructions (platform TEXT, mnem TEXT, description
con.commit()
```

When working with x86, I noticed that many instructions point to the same documentation. So, the plugin supports single level referencing. Just place '-R:[new instruction]' into description to redirect the loading. 'new instruction' is the target. So, when loading the script will detect the link and load the new target automatically.

```
cur.execute("INSERT INTO instructions VALUES (?, ?, ?)", ("x86", inst, "-R:%s" % first_inst)
```

Skeletons in the closet

The documentation database was created using a rather hackish screen scraping technique by the x86doc project which I forked. So, there are probably some strange characters or tags in the text. At least, it is a mechanical process so I expect that the information is correct relative to the original Intel PDF.

Ports

If you're a hopper user, there is a port called hopperref: https://github.com/zbuc/hopperref

If you're an x64dbg user, IdaRef is integrated with the `mnemonichelp xxx` command or through the context menu. Fork: https://github.com/x64dbg/idaref

Enjoy!

VMAttack - by: Anatoli Kalysch

Plugin description

an IDA PRO Plugin which enables the reverse engineer to use additional analysis features designed to counter virtualization-based obfuscation.

readme for VMAttack

VMAttack IDA PRO Plugin

IDA Pro Plugin for static and dynamic virtualization-based packed analysis and deobfuscation.

VMAttack was awarded the *second place* at the annual IDA Pro Plug-in Contest in 2016 !

Introduction

VMAttack is an IDA PRO Plug-in which enables the reverse engineer to use additional analysis features designed to counter *virtualization-based obfuscation* . For now the focus is on stack based virtual machines , but will be broadened to support more architectures in the future. The plugin supports static and dynamic analysis capabilities which use IDA API features in conjunction with the plugins own analysis capabilities to provide automatic, semi-automatic and manual analysis functionality. The main goal of this plugin is to assist the reverse engineer in undoing the *virtualization-based obfuscation* and to automate the reversing process where possible.

Installation

Prerequisites

- IDA Pro >= 6.6
- Python 2.7.10/.11
- Tested with Windows 7 and Windows 10.

Guided Install

To install the plugin simply run the setup.py:

```
python setup.py install
```

You will be prompted for the full path to your IDA PRO installation, aside from that no user interaction should be required.

Alternative manual Install:

Should the guided install fail for any reason a manual installation is also possible.

The only required python dependencies are `distorm3` and `idacute` which can be installed via pip:

```
pip install distorm3 idacute
```

Next the Windows environment variable should be set:

```
setx VMAttack X:full\path\to\plugin
```

Last you should copy the `VMAttack_plugin_stub.py` into your IDA PRO Plugins directory. That's it, now you're good to go!

Possible Issues

- Unable to find vcvarsall.bat => you need to install 'Microsoft Visual C++ Compiler for Python 2.7'

Quick start guide

The Example folder contains the obfuscated binary and source binary of an add function. The obfuscated addvmp contains the VM function which we will analyze now.

After a quick glance over the binary we see the simple structure: two arguments, `0AFFE1` and `0BABE5` are deployed on the stack and then a stub is called.

The stub starts the virtual machine function with a reference to the start of the VM byte code pushed onto the stack.

Following the address we see the virtual machine function which is basically an interpreter for the received byte code.

A solution to this obfuscation would be the reversal of the interpreter and the interpretation of the byte code by the reverse engineer. Due to the time consuming nature of this task we will try to reverse the binary with our VMAttack plugin.

VMAttacks static analysis functionality is enabled by default. The dynamic analysis capabilities however require an extra step. Since we want to use the static and dynamic capabilities for this demo, first we need to enable the dynamic functionality of VMAttack. This is done by either generating an instruction trace dynamically or loading an instruction trace from file. Trace generation is automatic and upon completion it will produce a success notification in IDA s *output window* . Traversed paths will be colored in a shade of blue, where a darker shade represents a higher number of traversals. Alternatively the loaded trace will only produce the success notification in IDA s *output window* . With the newly generated/loaded trace we now have dynamic and static capabilities enabled and can start the *grading system analysis* . Starting with the *grading analysis* is usually a good fit, since it is automated and takes several analysis

capabilities into account. This enables a cumulative result which can even tolerate analysis errors to some extent and still produce good results. At the end of the grading analysis the now graded trace will be presented in the grading viewer . The trace can now be filtered either by double clicking a grade or via context menu where the user will be prompted to input the grade threshold to display. In the case of addvmp it will be enough to select the highest grade to be presented with the deobfuscated function (since the original function is quite simple in this case). In becomes obvious, that the two values passed over the stack are added together. Additionally, should the result be not satisfiable, the user can change the importance of an analysis function (see settings) or even disable them (by setting the importance to 0), to produce better results. Simply change the importance and re-run the grading analysis.

Lets assume we have a more complicated function and the *grading analysis* did not lead us to the relevant instructions. One of the *semi-automated analysis* capabilities could present a viable alternative or even show us which analysis function failed the grading system. The *input/output analysis* could provide leads as to how the input arguments of the VM function are used and whether there is a connection between function input and function output. By checking the two input values `AFFE1` and `BABE5` and additionally the output value `16ABC6` it becomes evident which register contains the important instructions for our obfuscated function and how the `eax` return value came to be `16ABC6` (=AFFE1+BABE5).

Another powerful functionality is the *clustering analysis* . It enables the reverse engineer to quickly discern between repeating instructions and unique ones. The *clustering analysis view* additionally enables quick removal of unnecessary clusters (or instructions) in a way speeding up the work of the reverse engineer. Should a mistake be made it can be undone or alternatively the original trace can be restored. To make sense of the clustering analysis usually requires an extensive analysis of the trace and can require repeating the clustering analysis with a different cluster heuristic value set via settings .

Out of the semi-automatic analysis the *optimization analysis* requires the most user interaction. In turn it enables:

- Optimizations which make the trace easier to read or even filter as unnecessary recognized instructions.
- Filtering capabilities to remove as unnecessary recognized instructions or even whole registers from the trace.
- Undoing actions if you made a mistake.
- Restoring the initial trace if you hit a wall.

The *static analysis* in this case would enable us to analyze the byte code and optionally view the analysis as an *abstract VM graph* of the byte code. The static deobfuscation of the byte code will produce comments behind relevant bytes to describe the operation this byte produces. The commented instructions are quite intuitive and should be easy to read. The abstract VM graph in turn

will produce a control flow graph (in the case of addvmp just one basic block) filled with those abstract instructions from the byte code. This is also a good example of the accuracy of the static analysis, which without execution delivered an accurate representation of the initial deobfuscated function. After the static analysis we can clearly see that two arguments were passed to the function *(AOS = acces out of known space; indicates for example arguments passed via stack)* and that they were eventually added together and then returned.

Analysis Capabilities

The following subsection describes the analysis functions and offers additional information about the plugins inner workings. A quick start guide can be found in the next subsection.

The analysis features are subdivided into automatic, semi-automatic and manual, depending on user interaction necessary for the completion of the analysis. While the automatic analysis requires nearly none, manual analysis capabilities require interaction and will query the user if needed.

Instruction Trace

For the dynamic analysis capabilities of the plugin the use of an instruction trace is necessary. If you want to quickly check whether you already have a trace you can use the Show Trace feature. It will print your current trace in IDA s *output window* or nothing if you don't have a trace. But how can you actually get one? The two currently supported possibilities are to either generate an instruction trace or to load one from file:

- Trace generation requires a working IDA debugger(e.g. Win32 DBG or Bochs DBG) and uses the IDA debugger API to generate an instruction trace. Visited basic blocks and instructions are colored in a shade of blue (color can be removed afterwards via Remove Colors from Graph). During the execution function arguments are extracted, if not deactivated via settings .
- Alternatively an instruction trace can be loaded from file. Currently supported file types are .txt files exported from IDA s *trace window* and .json files saved via VMAttack . OllyDbg and ImmunityDbg generated .txt trace files are supported but are currently more limited in the available analysis capabilities.

A reverse engineer can decide to save an analyzed trace as a .json file which can be loaded later on to continue analysis. This provides a convenient way to experiment on traces and combine different analysis together for improved results.

What is a trace?

In the plugin context a trace is a list object consisting of trace lines. Each trace line consists of four basic values:

- The thread id
- The address of the instruction
- The instruction
- The CPU Context after execution

additionally a VMAttack trace line can contain:

- A stack comment
- A grade

These are explained later in their respective analysis sections. Each time a trace is presented in one of the plugins custom viewers, at least the four basic values will be available to ensure readability.

For more information see `dynamic/TraceRepresentation.py`

Automated Analysis

Grading System

The grading system analysis is a combination of all available analysis capabilities(static and dynamic) with additional pattern matching mechanisms. The basic principle is, that each trace line is initialized with a certain seed value, depending on the uniqueness of the trace line. This value is in turn upgraded or downgraded after each analysis run, depending on the importance an analysis assigns to this trace line. Additionally, after each analysis run a pattern matching upgrade or downgrade occurs for certain predefined patterns. This design results in high robustness, as the overall grade of a trace line consists of several analysis steps and one failing analysis will not necessarily lead to wrong results. Additionally a high level of automation can be achieved by countering the known weaknesses of certain analysis steps with pattern matching. After a successful grading analysis attempt the user is presented with the result in the grading analysis viewer. The result consists of necessary trace information and the grade for each trace line.

The grading analysis viewer provides additional interaction interfaces to the user and a custom context menu(CCM):

- Double clicking a grade declares it as threshold and will remove all lines with a lower grade
- CCM Set grades threshold...: only grades >= threshold will be displayed
- CCM Show All: display all trace lines
- CCM Export this trace: save analysis result as .json file

Semi-Automatic Analysis

Static Analysis Create Abstract VM Graph

The abstract VM graph is an abstraction of how the control flow graph for the obfuscated function might look like. It is populated with instructions from the deobfuscated VM byte code and thus will require the previous execution of the static deobfuscate function. If static deobfuscate was not executed before it will be shortly before graph creation.

Static deobfuscate

The static deobfuscate function tries to statically determine the instructions that will be executed by the byte code in the provided virtual machine function. The semi-automatic version of this analysis tries to determine all necessary values(byte code start, byte code end, jmp table base address, vm function start) automatically. In the event of a mismatch the reverse engineer will be prompted for a decision. For example in this case the plugin determined the start of the bytecode to be at `0x40489c`, while the parameter for the function was `0x40489a`. This triggered a mismatch and the user is prompted with a decision which value should be used. In this case the function parameter `0x40489a` was indeed the start and should have been used as byte code start value.

Another possibility where user interaction might become necessary, is when byte code is separated by useful executable instructions. In this example one could stop the deobfuscation at `0x4048f7` by choosing 'No' and input in the next prompt `Where should deobfuscation continue?` the address right after the instructions `0x404901`, as it is safe.

Dynamic Analysis Dynamic Trace Optimization

The trace optimizations viewer provides a way to dynamically interact with the trace. On one hand it allows the user to filter often occurring instructions or even remove whole register interactions from the trace, on the other it allows for powerful optimizations to be applied to the trace.

Constant Propagation : Constants are propagated where possible. This means registers are switched with their values and offsets which can be computed will be computed.

Stack Address Propagation : Every time a stack address is read the value on the stack address will be available as Stack comment.

Operation Standardization (folding) : A weaker version of the peephole optimization which standardizes certain operations.

Unused Operand Folding (folding) : Operands that are not used in later execution steps are purged from the trace.

Peephole (folding) : The trace is traversed for specific patterns which are then replaced or deleted if deemed unnecessary.

Propagations should be always save to use, as they do not leave out anything. Foldings should be used with care, as they leave out lines deemed unuseful and as such might leave out too much.

These optimizations are a very important foundation of this plugin, as most other analysis functions use or even require one or more of these optimizations to be executed on the trace prior to their analysis.

For easier interaction with the trace the optimization viewer provides trace line removal by double click and a custom context menu(CCM):

- CCM Undo: Undo change
- CCM Restore original trace: restore the original trace
- CCM Open in Clustering analysis: send the current file to clustering analysis
- CCM Export this trace: save analysis result as .json file

To filter whole registers (e.g. eax) from the trace lines toggle the according check box by the register name.

VM Input / Output Analysis

The Input / Output Analysis provides essentially a black box analysis. For each register, starting at the output value of the virtual machine function a backtrack of how this value came to be will extract the relevant trace lines. If you are interested in a specific output value and want to know how this value came to be this is the goto function. As an example lets assume we are interested in the value of the `eax` register. First the function will backtrack to which stack address the value in `eax` was mapped. Then the backtracking will continue, to determine, whether the value on the stack is a result of a calculation and if it is the backtracking will start recursively for the calculation values. Every trace line along the way which contains the result value or the components will be added to the registers input/output trace.

By checking the check boxes of the values the viewer adds colorization to them:

- Rust red: input values
- Violet: output values
- Olive: both

Colorization takes into account not only values used in the disasm section but also the CPU context!

If a register turns olive it is an indication, that its trace lines contain both a input and an output parameter. This means the register handles both the selected input and output values and either computations take place or the values are part of the trace lines CPU context.

Clustering Analysis

The main goal of the clustering analysis is to divide the instruction trace into clusters(=repeating instructions) and singles(=non-repeating instructions). The default declares a cluster if an address is encountered more than two times however this can be changed via settings by changing the value for `cluster heuristic` . After successful analysis the reverse engineer is presented with an instruction trace with single instructions and clusters. If basic block detection was not deactivated in the settings , the clusters themselves are additionally subdivided into basic blocks. Each basic block is shown as a one line summary. The basic block description consists of the basic block position inside the cluster, the start and end addresses and an instruction and stack change summary. While the stack change summary will contain all stack changes which took place during this basic block, instructions are only shown, if their value is used later or it is unclear whether their value is used. Instructions whose computations are simply overridden are not displayed in the basic block summary. Alternatively the basic block subdivision can be deactivated in the settings menu. After reloading the trace (or any removal action in the cluster viewer) the basic blocks will be gone and a cluster will only consist of its trace lines.

To increase interactivity the clustering viewer provides trace line and cluster removal by double click and a custom context menu(CCM):

- CCM Remove X: Remove a line / cluster / basic block.
- CCM Remove several clusters...: remove the X clusters with most occurrence. For example a threshold of 3 removes the 3 most common clusters.
- CCM Undo: Undo change
- CCM Restore original trace: restore the original trace
- CCM Export this trace: save analysis result as .json file

Aside from the clustering viewer the clustering analysis also opens a second viewer. The stack changes viewer presents the stack and shows the changes which took place during the execution. It will be opened as part of the clustering analysis along with the clustering viewer. Aside from the stack address view, the user can also see which stack addresses are mapped onto which cpu registers. Basically this provides a stack centered point of view on the execution which is especially useful for stack machine based virtual machines, which operate mainly over the stack.

Manual Analysis

VM Context The VM Context consists of four values:

- Code Start - the byte code start
- Base Addr - the base address of the jump table
- Code End - the byte code end
- VM Addr - the start address of the virtual machine function

The user has the choice whether he wants to determine it statically or dynam-

ically. An alternative to this is to manually input the values during static analysis or in the settings menu.

Static Analysis Deobfuscate from...

This provides a manual interface for the static deobfuscation function. Static deobfuscation requires four input values which are basically the VM context. This is useful if static and/or dynamic determination of the VM context fail but the reverse engineer managed to retrieve the right values. Upon selection the user will be prompted for all the values:

- Code Start - the byte code start
- Base Addr - the base address of the jump table
- Code End - the byte code end
- VM Addr - the start address of the virtual machine function

Dynamic Analysis Follow Virtual Register: This provides a manual interface to the register tracking functionality.

Find Virtual Reg to Reg mapping: This function helps to map the stack addresses of the output from the virtual machine function to the registers in which those values are returned.

Find VM Function Output Parameter: Finds function output parameter.

Find VM Function Input Parameter: Finds function input parameter.

Address Count: The address count reads in a trace and returns in IDA s output window the ratio: (Address (disasm): frequency of occurrence).

Settings

The Settings provide the necessary interface to enable the user to change values on the fly or even input own values if the ones determined by the plugin are wrong. Further changes in the default behaviour of the program can also be selected or removed.

Code Start : the byte code start

Base Addr : the base address of the jump table

Code End : the byte code end

VM Addr : the start address of the virtual machine function

Show Basic Blocks : show basic blocks during clustering analysis

Greedy Clustering : cluster until no more clusters are found

Cluster Heuristic : threshold after how many repetitions an address is considered a cluster

Input/Output Importance : Importance of input/output analysis for the grading system (set to 0 to disable)

Clustering Importance : Importance of clustering analysis for the grading system (set to 0 to disable)

Pattern Matching Importance : Importance of pattern matching analysis for the grading system (set to 0 to disable)

Memory usage Importance : Importance of memory instructions analysis for the grading system (set to 0 to disable)

Step Into System Libraries : Should system libraries be disregarded during trace generation

Extract function parameters : Should function parameters be extracted during trace generation

Current Progress

Current Version: 0.2

This plugin already has a working state. Nevertheless there are several improvements (analysis capabilities of this plugin and software engineering optimizations) that will be implemented over time. The TODOs in the code give a rough estimate about those future improvements.

Current TODO : improve usage of the static analysis in grading system

Next TODO : enhance multithreading of analysis capabilities

AutoRE - by: Aliaksandr Trafimchuk

Plugin description

IDA PRO auto-renaming plugin with tagging support.

readme for AutoRE

Features

1. Auto-renaming dummy-named functions, which have one API call or jump to the imported API

Before

After

2. Assigning TAGS to functions accordingly to called API-indicators inside

- Sets tags as repeatable function comments and displays TAG tree in the separate view

Some screenshots of TAGS view:

How TAGs look in unexplored code:

You can easily rename function using its context menu or just pressing `n` hotkey:

Installation

Just copy `auto_re.py` to the `IDA\plugins` directory and it will be available through `Edit -> Plugins -> Auto RE` menu

---

sk3wldbg - by: Chris Eagle

Plugin description

> the Sk3wlDbg plugin for IDA Pro. It's purpose is to provide a front end for using the Unicorn Engine to emulate machine code that you are viewing with IDA.

readme for sk3wldbg

WARNING: THIS CODE IS VERY RAW AND PROBABLY VERY BUGGY!

Introduction

This is the Sk3wlDbg plugin for IDA Pro. It's purpose is to provide a front end for using the Unicorn Engine to emulate machine code that you are viewing with IDA.

The plugin installs as an IDA debugger which you may select whenever you open an IDA database containing code supported by Unicorn. Currently supported architectures include:

- x86
- x86-64
- ARM
- ARM64
- MIPS
- MIPS64
- SPARC
- SPARC64
- M68K

BUILDING:

The plugin is dependent on the Unicorn engine. IDA versions 6.x and older (pre 7.0) are buit as 32-bit binaries. If you are using one of these versions of IDA you MUST have a 32-bit build of the Unicorn library for your IDA platform (Windows, Linux, OS X). If you are using IDA version 7.0 or later, you MUST have a 64-bit build of Unicorn.

On all platforms you should clone sk3wldbg into your IDA SDK plugins subdirectory so that you end up with $IDASDKDIR/plugins/sk3wldbg because the build files all use relative paths to find the IDA header files.

Building Unicorn for Linux / OSX

- If building Unicorn for IDA 6.x on Linux use: ./make.sh linux32
- If building Unicorn for IDA 7.x on Linux use: ./make.sh linux64
- If building Unicorn for OS X use: ./make.sh macos-universal

Follow `make.sh` with `make install`

Build sk3wldbg for Linux / OS X:

Use the include Makefile to build the plugin. You may need to adjust the paths that get searched to find your IDA installation ("/Applications/IDA Pro N.NN" is assumed on OSX and /opt/ida-N.NN is assumed on Linux, were N.NN is derived from the name of your IDA SDK directory eg idasdk695 and should match your IDA version number). This is required to successfully link the plugin. Note that the Makefile assumes that the Unicorn library headers have been copied into the sk3wldbg directory alongside the plugin source files (this is already done in the git repo). If you want to switch to using the actual Unicorn headers, make sure you update the Makefile.

$ cd $IDASDKDIR/plugins/sk3wldbg $ make

Compiled binaries will end up in $IDASDKDIR/plugins/sk3wldbg/bin

LINUX

	ida	ida64
IDA 6.x plugin	sk3wldbg_user.plx	sk3wldbg_user.plx64
IDA 7.x plugin	sk3wldbg_user.so	sk3wldbg_user64.so

OS/X

	ida	ida64
IDA 6.x plugin	sk3wldbg_user.pmc	sk3wldbg_user.pmc64
IDA 7.x plugin	sk3wldbg_user.dylib	sk3wldbg_user64.dylib

Copy the plugin(s) into your /plugins directory and Sk3wlDbg will be listed as an available debugger for all architectures supported by Unicorn.

Build Unicorn for Windows

Unicorn include unicorn.sln which may be used to build both 32 and 64-bit versions of Unicorn. The necessary binaires end up in unicorn/msvc/distro/Win32 and unicorn/msvc/distro/x86. You will need unicorn.lib and unicorn.dll for your version of IDA (32 or 64-bit). Copy the appropriate unicorn.lib into your sk3wldbg git tree at sk3wldbg/lib/x86 or sk3wldbg/lib/x64.

Build sk3wldbg for Windows

Build with Visual Studio C++ 2013 or later using the included solution (.sln) file (sk3wlbdg.sln). Several build targets are available depending on which version of IDA you are using:

	ida	ida64
IDA 6.x plugin	Release/Win32 sk3wldbg_user.plw	Release64/Win32 sk3wldbg_user.p64
IDA 7.x plugin	Release/x64 sk3wldbg_user.dll	Release64/x64 sk3wldbg_user64.dll

---

Note that the project configuration assumes that the Unicorn library headers have been copied into the sk3wldbg directory alongside the solution file (this is already done in the git repo). If you want to switch to using the actual Unicorn headers, make sure you update the Visual Studio project settings.

Copy the plugin(s) into your /plugins directory and Sk3wlDbg will be listed as an available debugger for all architectures supported by Unicorn.

Note that the unicorn dll needs to be found in your PATH or copied into your IDA installation directory.

INSTALLATION

Assuming you have installed IDA to $IDADIR, install the plugin by copying the compiled binaries from $IDASDKDIR/bin/plugins to $IDADIR/plugins (Linux/Windows) or $IDADIR/idabin/plugins (OS X). Windows users should also copy unicorn.dll into $IDADIR. Linux and OS X users should make sure they have installed the Unicorn shared library into an appropriate location on their respective systems (/usr/local/lib often works). This should already be taken care of if you build and install Unicorn from source.

Pre-built binaries:

As an alternative to building the plugin yourself, pre-built binaries for IDA 6.95 (Windows, Linux, OS X) are available in the bins directory. Make sure that you have a suitable Unicorn installed for your platform.

USING THE PLUGIN

With the plugin installed, open a binary of interest in IDA and select Sk3wlDbg as your debugger (Debugger/Switch debugger). If Sk3wlDbg does not appear as an available debugger, it has either not been installed correctly, the Unicorn shared library can't be found, or the current processor type is not supported by the plugin.

No options are currently recognized by the plugin. When you launch the debugger you will be asked whether you wish to begin execution at the cursor location or at the program's advertised entry point. You should probably also set some breakpoints to make sure you gain control of the debugger at some point.

The plugin contains very minimalist ELF32/64 and PE/PE32+ loaders to load the file image into the Unicorn emulator instance. Outside of these formats the plugin simply copies the contents of your IDA sections into the emulator. You currently also get a stack and that's about it.

For ELF64/x86_64, the emulator assumes Linux and sets up a minimal trampoline from ring 0 to ring 3 at debug start. Additionaly ring 0 code is

installed to handle sysenter and provide a sysexit back to ring 3. A conditional breakpoint can be installed at the tail end of the systenter code (marked by a nop) to examine the syscall arguments and, if desired, manipulate the process state before resuming execution. See linux_kernel_x64.asm and linux_x64_syscall_bpcond.py for ideas.

Future updates will provide similar ring 0 stubs for ELF32/x86/Linux and PE32+/x86_64/Windows.

THINGS THAT WORK (> 0% of the time)

- Basic debugger operations such as step and run
- Breakpoints are just implemented as a set against which the current program counter is compared. Software breakpoints (such as INT 3) are not used.
- IDA's "Take memory snapshot" feature works.
- Conditional breakpoints handled by IDA
- Installed IDC functions allow for mapping additional memory into a Unicorn process

```
int64 sk3wl_mmap(int64 base, long size, int perms) where perms are a combination of:
    #define SEGPERM_EXEC  1         ///< Execute
    #define SEGPERM_WRITE 2         ///< Write
    #define SEGPERM_READ  4         ///< Read
void sk3wl_munmap(int64 base, long size)
```

`sk3wl_mmap` may be used to map new regions of memory into an emulated unicorn process. These may be invoked from python via the `eval_idc_expr` function:

```
idaapi.eval_idc_expr(idaapi.idc_value_t(), BADADDR, "sk3wl_mmap(0x41414000, 0x1000,
```

THINGS THAT DON'T WORK (because they are not yet implemented)

- IDA Appcalls
- Exception handling (as in the debugger catching exception that happen in the emulated code like out of bounds memory accesses or illegal instructions)
- Tracing
- Stack traces
- Many other features I have not yet thought of

OTHER FUTURE WORK

- Extensible hooking interface to hook system calls and other exceptions
- Extensible hooking interface to hook library function calls
- Support for loading required shared libraries into the emulated process

- PEB/TEB and fs segment setup for PE based processes
- Many other features I have not yet thought of

Keypatch - by: Nguyen Anh Quynh & Thanh Nguyen

Plugin description

> a plugin of IDA Pro for Keystone Assembler Engine. See this introduction for the motivation behind Keypatch, and this slides for how it is implemented.

readme for Keypatch

Keypatch

Keypatch is the award winning plugin of IDA Pro for Keystone Assembler Engine .

Keypatch consists of 3 tools inside.

- Patcher & Fill Range : these allow you to type in assembly to directly patch your binary.
- Search : this interactive tool let you search for assembly instructions in binary.

See this quick tutorial for how to use Keypatch, and this slides for how it is implemented.

Keypatch is confirmed to work on IDA Pro version 6.4, 6.5, 6.6, 6.8, 6.9, 6.95, 7.0, 7.5 but should work flawlessly on older versions. If you find any issues, please report .

---

1. Why Keypatch?

Sometimes we want to patch the binary while analyzing it in IDA, but unfortunately the built-in asssembler of IDA Pro is not adequate.

- This tool is not friendly and without many options that would make the life of reverser easier.
- Only X86 assembler is available. Support for all other architectures is totally missing.
- The X86 assembler is not in a good shape, either: it cannot understand many modern Intel instructions.

Keypatch was developed to solve this problem. Thanks to the power of Keystone , our plugin offers some nice features.

- Cross-architecture: support Arm, Arm64 (AArch64/Armv8), Hexagon, Mips, PowerPC, Sparc, SystemZ & X86 (include 16/32/64bit).
- Cross-platform: work everywhere that IDA works, which is on Windows, MacOS, Linux.
- Based on Python, so it is easy to install as no compilation is needed.
- User-friendly: automatically add comments to patched code, and allow reverting (undo) modification.
- Open source under GPL v2.

Keypatch can be the missing piece in your toolset of reverse engineering.

---

2. Install

- Install Keystone core & Python binding for Python 2.7 from keystone-engine.org/download . Or follow the steps in the appendix section .
- Install Six module from pip because it is used by the keypatch.py: `pip install six` .
- Copy file `keypatch.py` to IDA Plugin folder, then restart IDA Pro to use Keypatch.
  - On Windows, the folder is at `C:\Program Files (x86)\IDA 6.9\plugins`
  - On MacOS, the folder is at `/Applications/IDA\ Pro\ 6.9/idaq.app/Contents/MacOS/plugins`
  - On Linux, the folder may be at `/opt/IDA/plugins/`

`NOTE`

- On Windows, if you get an error message from IDA about "fail to load the dynamic library", then your machine may miss the VC++ runtime library. Fix that by downloading & installing it from https://www.microsoft.com/en-gb/download/details.aspx?id=40784
- On other *nix platforms, the above error message means you do not have 32-bit Keystone installed yet. See appendix section below for more instructions to fix this.

---

3. Usage

- For a quick tutorial, see TUTORIAL.md . For a complete description of all of the features of Keypatch, keep reading.
- To patch your binary, press hotkey `CTRL+ALT+K` inside IDA to open Keypatch Patcher dialog.

  - The original assembly, encode & instruction size will be displayed in 3 controls at the top part of the form.
  - Choose the syntax, type new assembly instruction in the `Assembly` box (you can use IDA symbols).
  - Keypatch would *automatically* update the encoding in the `Encode` box while you are typing, without waiting for `ENTER` keystroke.
    * Note that you can type IDA symbols, and the raw assembly will be displayed in the `Fixup` control.
  - Press `ENTER` or click `Patch` to overwrite the current instruction with the new code, then *automatically* advance to the the next instruction.
    * Note that when size of the new code is different from the original code, Keypatch can pad until the next instruction boundary with NOPs opcode, so the code flow is intact. Uncheck the choice `NOPs padding until next instruction boundary` if this is undesired.
    * By default, Keypatch appends the modified instruction with the information of the original code (before being patched). Uncheck the choice `Save original instructions in IDA comment` to disable this feature.
  - By default, the modification you made is only recorded in the IDA database. To apply these changes to the original binary (thus overwrite it), choose menu `Edit | Patch program | Apply patches to input file`.

- To fill a range of code with an instruction, select the range, then either press hotkey `CTRL+ALT+K`, or choose menu `Edit | Keypatch | Fill Range`.
  - In the `Assembly` box, you can either enter assembly code, or raw hexcode. Some examples of acceptable raw hexcode are `90`, `aa bb`, `0xAA, 0xBB`.

- To revert (undo) the last patching, choose menu `Edit | Keypatch | Undo last patching`.

- To search for assembly instructions (without overwritting binary), open Keypatch Search from menu `Edit | Keypatch | Search`.

  - Choose the architecture, address, endian mode & syntax, then type assembly instructions in the `Assembly` box.
  - Keypatch would *automatically* update the encoding in the `Encode` box while you are typing, without waiting for `ENTER` keystroke.
  - When you click `Search` button, Keypatch would look for all the occurences of the instructions, and show the result in a new form.

- To check for new version of Keypatch, choose menu `Edit | Keypatch | Check for update`.
- At any time, you can also access to all the above Keypatch functionalities just by right-click in IDA screen, and choose from the popup menu.

---

4. Contact

Email keystone.engine@gmail.com for any questions.

For future update of Keypatch, follow our Twitter @keystone_engine for announcement.

Appendix. Install Keystone for IDA Pro

We all know that before IDA 7.0, IDA Pro's Python is 32-bit itself, so it can only loads 32-bit libraries. For this reason, we have to build & install Keystone 32-bit. However, since IDA 7.0 supports both 32-bit & 64-bit, which means we also need to install a correct version of Keystone. Simply install from Pypi, with `pip` (32-bit), like followings:

```
pip install keystone-engine
```

Done? Now go back to section 2 & install Keypatch for IDA Pro. Enjoy!

CGEN - by: Yifan Lu

Plugin description

> an extension of CGEN (which is an attempt at modeling CPUs in Scheme and then automatically generating simulators, assemblers, etc) to generate IDA Pro modules.

readme for CGEN

```
   README for GNU development tools

This directory contains various GNU compilers, assemblers, linkers,
debuggers, etc., plus their support routines, definitions, and documentation.

If you are receiving this as part of a GDB release, see the file gdb/README.
If with a binutils release, see binutils/README;  if with a libg++ release,
see libg++/README, etc.  That'll give you info about this
```

```
package -- supported targets, how to use it, how to report bugs, etc.

It is now possible to automatically configure and build a variety of
tools with one command.  To build all of the tools contained herein,
run the ``configure'' script here, e.g.:

    ./configure
    make

To install them (by default in /usr/local/bin, /usr/local/lib, etc),
then do:
    make install

(If the configure script can't determine your type of computer, give it
the name as an argument, for instance ``./configure sun4''.  You can
use the script ``config.sub'' to test whether a name is recognized; if
it is, config.sub translates it to a triplet specifying CPU, vendor,
and OS.)

If you have more than one compiler on your system, it is often best to
explicitly set CC in the environment before running configure, and to
also set CC when running make.  For example (assuming sh/bash/ksh):

    CC=gcc ./configure
    make

A similar example using csh:

    setenv CC gcc
    ./configure
    make

Much of the code and documentation enclosed is copyright by
the Free Software Foundation, Inc.  See the file COPYING or
COPYING.LIB in the various directories, for a description of the
GNU General Public License terms under which you can copy the files.

REPORTING BUGS: Again, see gdb/README, binutils/README, etc., for info
on where and how to report problems.
```

Labeless - by: Aliaksandr Trafimchuk

Plugin description

> Labeless is a plugin system for dynamic, seamless and realtime synchronization between IDA Database and debug backend. It consists of two parts: IDA plugin and debug backend's plugin.

readme for Labeless

Contributed By Check Point Software Technologies LTD.

Features

1. Seamless synchronization of labels, function names, comments and global variables (w/wo demangling)

- Synchronization modes
  - On demand
  - On rename (update on-the-fly)
- Supports image base-independent synchronization

2. Dynamic dumping of debugged process memory regions

It can be useful in the following cases:

- When debugged process has extracted/temporary/injected module which doesn't appear in modules list
- When it doesn't have a valid PE header
- When it has corrupted import table, etc.
- When it contains unpacked memory regions inside a binary, you can easily merge these new memory regions with the ones that are already present in your database
- When reconstructing chain of memory chunks which are used by malware (and not only, if you know what we mean) so that the picture of its behaviour is complete

3. Python scripting

We support the following list of debug backends for now:

- OllyDbg 1.10
- DeFixed 1.10 (FOFF's team mod)
- OllyDbg 2.01
- x64dbg (x32, x64)

Overview

Labeless is a multipurpose IDA Pro plugin system for labels/comments synchronization with a debug backend, with complex memory dumping and interactive Python scripting capabilities . It consists of two parts: IDA plugin and debug backend's plugin.

Labeless significantly reduces time that researcher spends on transferring already reversed/documented code information from IDA (static) to debugger (dynamic). It saves time, preventing from doing the same job twice. Also, you can document and add data to the IDB on the fly and your changes will be automatically propagated to debug backend, even if you will restart the virtual machine or instance of debug backend will crash. So, you will never lose your research.

This solution is highly upgradable. You can implement any helper scripts in Python on debug backend's side and then just call them from IDA with one line of code, parsing the results and automatically propagating changes to IDB.

We can take that memory region and put it in the IDB, fixing imports 'on-the-fly', using debug backend's functionality. No more need in ImpRec or BinScylla, searching for the regions in memory that contain the real IAT, because we get that information dynamically from the debugged process itself.

As a result we have a lot of memory regions that may represent even different modules (if the unpacking process if multistage) with valid references between them, which gives us a possibility to build a full control flow graph of the executable. Basically, we will end up with one big IDB, containing all the info on the specific case.

Virus Bulletin 2015

- Presentation
- Slides
- Dumping multiple injections into a single database video on YouTube
- Python scripting video on YouTube
- Basic labels sync video on YouTube

Videos

- Labeless setup on Win10 x64 (with x64dbg)
- Resolving APIs dynamically with Labeless & OllyDbg2
- Resolving APIs dynamically with Labeless & x64dbg

Installation

Usage of precompiled binaries (release version)

Debug-backend setup:

If you want to use both x86 and x86_64 targets, then you should do the following steps for each python distro.

- Set up Python 2.7 (x86/x86_64)
- Copy `deploy` directory to target machine, where you want to use a debugger backend
- Set up `protobuf 2.6.1` using the following commands:

```
cd c:\deploy
c:\Python27\python.exe setup_protobuf.py
```

- Install `labeless` python module, there are two ways to archive that, the first one is to use PyPI in case you have an Internet connection on the debug machine:

```
pip install --upgrade labeless
```

In case you don't have an Internet connection, you could install prebuilt module from release archive:

```
cd c:\deploy
c:\Python27\Scripts\pip.exe install labeless-1.1.2.65-py2.py3-none-any.whl
```

Note : If you have already used Labeless before and you want to update it, don't forget to reinstall python module each time you have new release

- Configure your debugger backend: set up `plugins` directory

Configuring of your IDA PRO:

Labeless supports Windows and Linux (starting from 6.9.5 version of) IDA PRO. Labeless handles only PE/AMD64 binaries. Labeless requires IDAPython plugin `python.[plw|p64|plx|plx64]` (it ships with IDA PRO, but make sure it works well).

There are plugins:

```
IDA[XX]\plugins\labeless_ida.plw - for IDA for Windows, handles 32-bit binaries (used with i
IDA[XX]\plugins\labeless_ida.p64 - for IDA for Windows, handles 64-bit binaries (used with i
IDA[XX]\plugins\labeless_ida.plx - for IDA for Linux, handles 32-bit binaries (used with ida
IDA[XX]\plugins\labeless_ida.plx64 - for IDA for Linux, handles 64-bit binaries (used with i
```

Copy Labeless plugins to your IDA's `plugins` directory, for example `c:\IDA68\plugins`

- In case you have IDA for Windows, please, use `.plw` / `.p64` plugins.

- If you have IDA for Linux, please, use `.plx` / `.plx64` plugins. Also, copy `IDA[XX]/libprotobuf.so.9` to your IDA home directory (for example `/home/alex/ida695/`), it's an important library.

Configuring of debug backends

1. OllyDbg 1.10

You may find prepared debugger in the following directory `OllyDbg110` . (Note!: Don't forget to set up debugger's `plugins` directory).

2. DeFixed 1.10 (FOFF's team mod)

Copy `DeFixed110\plugins\labeless_olly_foff.dll` to DeFixed `plugins` directory (Note!: Don't forget to set up debugger's `plugins` directory)

3. OllyDbg 2.01

You may find prepared debugger in the following directory `OllyDbg201` . (Note!: Don't forget to set up debugger's `plugins` directory).

4. x64dbg (x32,x64)

You may find prepared debugger in the following directory `x64dbg` .

Checking if everything works

- Start debug backend (debugger) and check for *Labeless* item presence in the *Plugins* menu. If there is any problem, then check Olly's log window for details. Open the log window and check for `LL: ok, binded at <IP>:<PORT>` message, its presence means that debug backend-side plugin is initialized successfully. Note: if you start many debuggers, you may see that the following message appears

Also, you may see the firewall alert

If you want to access the debug backend from another computer, you should allow the backend to listen by this dialog or manually.

- Start working with existing IDA database or use `Labeless -> Load stub database...` from the menu
- Open Labeless settings dialog using any of the following actions:
  - menu `Edit -> Plugins -> Labeless`
  - main menu `Labeless -> Settings...`
  - hotkey `Alt+Shift+E`

- Enter IP address and port of the guest machine (where debug backend is set up), then click on ' *Test connection'* button
- If IDA displays the message `Successfully connected!` , then configuration is done correctly.

Development

- Set up Python 2.7 (x86/x86_64)
- protobuf 2.6.1
- Visual Studio 2010 + Qt 4.8.4 (built with "QT" namespace) - required by IDA's 6.8 plugin (to proper use IDA's Qt). You can configure Qt by yourself with the following command:

```
configure -platform win32-msvc2010 -shared -release -no-webkit
-opensource -no-qt3support -no-phonon -no-phonon-backend -opengl
desktop -nomake demos -nomake examples -nomake tools -no-script
-no-scripttools -no-declarative -qtnamespace QT
```

- Visual Studio 2015 + Qt 5.4.1 to build IDA's 6.9 and debug backend plugins. You should build Qt 5.4.1 from sources. Do the following steps to do that:
  - Download sources of Qt 5.4.1 from http://download.qt.io/official_releases/qt/5.4/5.4.1/single/
  - Check this article out to set up Qt 5 requirements
  - Check the Hex blog about compiling Qt 5.4.1, grab the patch and apply it on Qt5 root dir. Then compile Qt 5
  - Set up Qt VS-Addin

How to use

- If you want to enable synchronization of labels (names) and comments from IDA to Olly you should check ' *Enable labels & comments sync* ' in Labeless settings dialog in IDA. There is one required field called ' *Remote module base* ', which should be set to the current module base of the analyzed application. You can find out that information in the debugger
- Select needed features, like *Demangle name* , *Local labels* , *Non-code names*
- Select comments synchronization type:
  - <Disabled>
  - Non-repeatable
  - Repeatable
  - All

Repeatable - are comments, which IDA shows in any referenced place.

- If you want to sync labels right now - press ' *Sync now* ' button. Labeless will sync all found names in your IDB with Olly. Settings dialog will be automatically closed, while saving all settings

- Also, you may use `Labeless -> Sync labels now` from IDA's main menu
- If you want to customize settings for IDADump engine, do it in the ' *IDADump* ' tab.
- To save changed settings, click on ' *Save & Close* '

Things automatically performed in the background

- If you enabled ' *Enable labels & comments sync* ' option, then Labeless will automatically synchronize all the data on any rename operation in IDA

Troubleshooting

Issue with Python 2.7.11 is described here , so avoid usage of this version. The latest stable supported version is 2.7.10.

- Q: Labeless for x64dbg x32 is works, but x64 doesn't. Why?
- A: Please, recheck that you have installed protobuf and 'labeless' module for Python x64

Download

- Download latest release of Labeless
- (old) Download Labeless 1.0.0.7 (include IDA 6.6 build)

Credits

- Axel Souchet aka 0vercl0k
- Duncan Ogilvie aka mrexodia

Ponce - by: Alberto Garcia Illera, Francisco Oca

Plugin description

> an IDA Pro plugin that provides users the ability to perform taint analysis and symbolic execution over binaries in an easy and intuitive fashion. With Ponce you are one click away from getting all the power from cutting edge symbolic execution. Enti

readme for Ponce

Ponce

Ponce (pronounced *['poN e]* pon-they) is an IDA Pro plugin that provides users the ability to perform taint analysis and symbolic execution over binaries in an easy and intuitive fashion. With Ponce you are one click away from getting all the power from cutting edge symbolic execution. Entirely written in C/C++.

Why?

Symbolic execution is not a new concept in the security community. It has been around for many years but it is not until around 2015 that open source projects like Triton and Angr have been created to address this need. Despite the availability of these projects, end users are often left to implement specific use cases themselves.

We addressed these needs by creating Ponce, an IDA plugin that implements symbolic execution and taint analysis within the most used disassembler/debugger for reverse engineers.

Installation

Ponce works with both x86 and x64 binaries in any IDA version >= 7.0. Installing the plugin is as simple as copying the appropiate files from the latest builds to the `plugins\` folder in your IDA installation directory.

Make sure you use the Ponce binary compiled for your IDA version to avoid any incompatibilities.

OS Support

Ponce works on Windows, Linux and OSX natively!

Use cases

- Exploit development : Ponce can help you create an exploit in a far more efficient manner as the exploit developer may easily see what parts of memory and which registers you control, as well as possible addresses which can be leveraged as ROP gadgets.
- Malware Analysis : Another use of Ponce is related to malware code. Analyzing the commands a particular family of malware supports is easily determined by symbolizing a simple known command and negating all the conditions where the command is being checked.

- Protocol Reversing : One of the most interesting Ponce uses is the possibility of recognizing required magic numbers, headers or even entire protocols for controlled user input. For instance, Ponce can help you to list all the accepted arguments for a given command line binary or extract the file format required for a specific file parser.
- CTF : Ponce speeds up the process of reverse engineer binaries during CTFs. As Ponce is totally integrated into IDA you don't need to worry about setup timing. It's ready to be used!

The plugin will automatically run, guiding you through the initial configuration the first time it is run. The configuration will be saved to a configuration file so you won't have to worry about the config window again.

Use modes

- Tainting engine : This engine is used to determine at every step of the binary's execution which parts of memory and registers are controllable by the user input.
- Symbolic engine : This engine maintains a symbolic state of registers and part of memory at each step in a binary's execution path.

Examples

Negate and inject a condition In the next gif we can see the use of automatic tainting and how we can negate a condition and inject it in memory while debugging:

- We select the symbolic engine and set the option to symbolize `argv` .
- We identify the condition that needs to be satisfied to win the crackMe.
- We negate an inject the solution everytime a byte of our input is checked against the key.
- Finally we get the key `elite` that has been injected in memory and therefore reach the `Win` code.

The crackme source code can be found here

Using the tainting engine to track user controlled input In this example we can see the use of the tainting engine with cmake. We are:

- Passing a file as argument to cmake to have him parsing it.
- We select we want to use the tainting engine
- We taint the buffer that "'fread()"" reads from the file.
- We resume the execution under the debugger control to see where the taint input is moved to.
- Ponce will rename the tainted functions. These are the functions that somehow the user has influence on, not the simply executed functions.

Use Negate, Inject & Restore In the next example we are using the snapshot engine:

- Passing a file as argument.
- We select we want to use the symbolic engine.
- We taint the buffer that “‘fread()“ reads from the file.
- We create a snapshot in the function that parses the buffer read from the file.
- When a condition is evaluated we negate it, inject the solution in memory and restore the snapshot with it.
- The solution will be ”valid” so we will satisfy the existent conditions.

The example source code can be found here

Usage

In this section we will list the different Ponce options as well as keyboard shortcuts:

- Access the configuration and taint/symbolic windows: Edit > Ponce > Show Config (Ctl+Shift+P and Ctl+Alt+T)
- Enable/Disable Ponce tracing (Ctl+Shift+E)
- Symbolize/taint a register (Ctl+Shift+R)
- Symbolize/taint memory. Can be done from the IDA View or the Hex View (Ctl+Shift+M)
- Solve formula (Ctl+Shift+S)
- Negate & Inject (Ctl+Shift+N)
- Negate, Inject & Restore Snaphot (Ctl+Shift+I)
- Create Execution Snapshot (Ctl+Shift+C)
- Restore Execution Snapshot (Ctl+Shift+S)
- Delete Execution Snapshot (Ctl+Shift+D)
- Execute Native (Ctl+Shift+F9)

Triton

Ponce relies on the Triton framework to provide semantics, taint analysis and symbolic execution. Triton is an awesome Open Source project sponsored by Quarkslab and maintained mainly by Jonathan Salwan with a rich library. We would like to thank and endorse Jonathan’s work with Triton. You rock! :)

Building

Since Ponce v0.3 we have moved the building compilation process to use `CMake` . Doing this we unify the way that configuration and building happens for Linux, Windows and OSX. We now support providing feedback on the pseudocode about symbolic or taint instructions. For this feature to work you need to add `hexrays.hpp` to your IDA SDK include folder. `hexrays.hpp` can be found on `plugins/hexrays_sdk/` on your IDA installation path. If you have not purchased the hex-rays decompiler you can still build Pnce by using `-DBUILD_HEXRAYS_SUPPORT=OFF` . We use Github actions as our CI environment. Check the action files if you want to understand how the building process happens.

FAQ

Why the name of Ponce? Juan Ponce de León (1474 – July 1521) was a Spanish explorer and conquistador. He discovered Florida in the United States. The IDA plugin will help you discover, explore and hopefully conquer the different paths in a binary.

Can Ponce be used to analyze Windows, OS X and Linux binaries? Yes, you can natively use Ponce in IDA for Windows or remotely attach to a Linux or OS X box and use it. In the next Ponce version we will natively support Ponce for Linux and OS X IDA versions.

How many instructions per second can handle Ponce? In our tests we reach to process 3000 instructions per second. We plan to use the PIN tracer IDA offers to increase the speed.

Something is not working! Open an issue , we will solve it ASAP ;)

I love your project! Can I collaborate? Sure! Please do pull requests and work in the opened issues. We will pay you in beers for help ;)

Limitations

Concolic execution and Ponce have some problems:

- Symbolic memory load/write: When the index used to read a memory value is symbolic like in `x = aray[symbolic_index]` some problems arise that could lead on the loose of track of the tainted/symbolized user controled input.
- Triton doesn't work very well with floating point instructions .
- Concolic execution only analyzed the executed instructions. That means that symbolic tracking is lost in cases like the following:

```
int check(char myinput)  // Input is symbolic/tainted
{
int flag = 0;
if (myinput == 'A')  //This condition is symbolic/tainted
  flag = 1
else
  flag =- 1;
return flag; // flag is not symbolic/tainted!
}
```

Authors

- Alberto Garcia Illera (@algillera) agarciaillera@gmail.com
- Francisco Oca (@francisco_oca) francisco.oca.gonzalez@gmail.com

---

SimplifyGraph - by: Jay Smith

Plugin description

> IDA Pro plugin to assist with complex graphs

readme for SimplifyGraph

SimplifyGraph - 3rd place winner of the 2017 Hex-Rays IDA Pro Plugin Contest

Background

My personal preference is to use IDA's Graph mode when doing the majority of my reverse engineering. It provides a graphical representation of the control flow graph and gives visual cues about the structure of the current function that helps me better understand the disassembly.

Graph mode is great until the function becomes complex. IDA is often forced to place adjacent nodes relatively far apart, or have edges in the graph cross and have complex paths. Using the overview graph becomes extremely difficult due to the density of nodes and edges, like in Figure 1.

Figure 1: An annoying function

IDA has a built-in mechanism to help simplify graphs: creating groups of nodes, which replaces all of the selected nodes with a new group node representative. This is done by selecting one or more nodes, right-clicking, and selecting "Group nodes", shown in Figure 2. Doing this manually is certainly possible, but it

becomes tedious to follow edges in complex graphs and correctly select all of the relevant nodes without missing any, and without making mistakes.

Figure 2: Manual group creation

The SimplifyGraph IDA Pro plugin we're releasing is built to automate IDA's node grouping capability. The plugin is source-compatible with the legacy IDA SDK in 6.95, and has been ported to the new SDK for IDA 7.0. Pre-built binaries for both are available on the Release tab.

The plugin has several parts, introduced below.

Create Unique-Reachable (UR) Subgraph

Unique-Reachable nodes are all nodes reachable in the graph from a given start node and that are not reachable from any nodes not currently in the UR set. For example in Figure 3, all of the unique-reachable nodes starting at the green node are highlighted in blue. The grey node is reachable from the green node, but because it is reachable from other nodes not in the current UR set it is pruned prior to group creation.

Figure 3: Example Unique Reachable selection

The plugin allows you to easily create a new group based on the UR definition. Select a node in IDA's graph view to be the start of the reachable search. Right click and select "SimplifyGraph --> Create unique-reachable group". The plugin performs a graph traversal starting at this node, identifies all reachable nodes, and prunes any nodes (and their reachable nodes) that have predecessor nodes not in the current set. It then prompts you for the node text to appear in the new group node.

If you select more than one node (by holding the Ctrl key when selecting nodes) for the UR algorithm, each additional node acts as a sentry node. Sentry nodes will not be included in the new group, and they halt the graph traversal when searching for reachable nodes. For example in Figure 4, selecting the green node first treats it as the starting node, and selecting the red node second treats it as a sentry node. Running the "Create unique-reachable group" plugin option creates a new group made of the green node and all blue nodes. This can be useful when you are done analyzing a subset of the current graph, and wish to hide the details behind a group node so you can concentrate on the rest of the graph.

Figure 4: Unique reachable with sentry

The UR algorithm operates on the currently visible graph, meaning that you can run the UR algorithm repeatedly and nest groups.

Switch case groups creation

Switch statements implemented as jump tables appear in the graph as nodes with a large fan-out, as shown in Figure 5. The SimplifyGraph plugin detects when the currently selected node has more than two successor nodes and adds a right-click menu option “SimplifyGraph --> Create switch case subgraphs”. Selecting this runs the Unique-Reachable algorithm on each separate case branch and automatically uses IDA’s branch label as the group node text.

Figure 5: Switch jumptable use

Figure 6 shows a before and after graph overview of the same function when the switch-case grouping is run.

Figure 6: Before and after of switch statement groupings

Isolated Subgraphs

Running Edit --> Plugins --> SimplifyGraph brings up a new chooser named ”SimplifyGraph - Isolated subgraphs” that begins showing what I call isolated subgraphs of the current graph. A full definition appears later in the appendix including how these are calculated, but the gist is that an isolated subgraph in a directed graph is a subset of nodes and edges such that there is a single entrance node, a single exit node, and none of the nodes (other than the subgraph entry node) are reachable by nodes not in the subgraph.

Finding isolated subgraphs was originally researched to help automatically identify inline functions. It does this, but it turns out that this graph construct occurs naturally in code without inline functions. This isn’t a bad thing as it shows a natural grouping of nodes that could be a good candidate to group to help simplify the overall graph and make analysis easier.

Once the chooser is active, you can double click (or press Enter) on a row in the chooser to highlight the nodes that make up the subgraph. You can create a group for an isolated subgraph by doing one of:

- Right-click on the chooser row and select ”Create group”, or press Insert while a row is selected.
- Right-click in a highlighted isolated subgraph node and select ”SimplifyGraph --> Create isolated subgraph”.

Doing either of these prompts you for text for the new graph node to create. If you manually create/delete groups using IDA you may need to refresh the chooser’s knowledge of the current function groups (right-click and select ”Refresh groups” in the chooser). You can right click in the chooser and select ”Clear highlights” to remove the current highlights. As you navigate to new functions the chooser updates to show isolated subgraphs in the current function. Closing the chooser removes any active highlights. Any custom colors

you applied prior to running the plugin are preserved and reapplied when the current highlights are removed.

Isolated subgraph calculations operates on the original control flow graph, so isolated subgroups can't be nested. As you create groups, rows in the chooser turn red indicating a group already exists, or can't be created because there is an overlap with an existing group.

Another note: this calculation does not currently work on functions that do not return (those with an infinite loop). See appendix for details.

Graph Complement

Creating groups to simplify the overall control flow graph is nice, but it doesn't help understand the details of a group that you create. To assist with this, the last feature of the plugin helps view groups in "isolation". Right clicking on a collapsed group node, or a node that that belongs to an uncollapsed group (as highlighted by IDA in yellow), brings up the plugin option "Complement & expand group" and "Complement group", respectively. When this runs the plugin creates a group of all nodes other than the group you're interested in. This has the effect of hiding all graph nodes that you aren't currently examining and allows you to better focus on analysis of the current group. As you can see, we're abusing group creation a bit so that we can avoid creating a custom graph viewer, and instead stay within the built-in IDA graph disassembly view which allows us to continue to markup the disassembly as you're used to.

Complementing the graph gives you view like in Figure 7, where the entire graph is grouped into a node named "Complement of group X". When you're done analyzing the current group, right click on the complement node and select IDA's "Ungroup nodes" command.

Figure 7: Group complement

Example Workflow

As an example that exercises the plugin, let's revisit the function in Figure 1. This is a large command-and-control dispatch function for a piece of malware. It contains a large if-else-if series of inlined strcmp comparisons that branch to the logic for each command when the input string matches the expected command.

- Find all of the inline strcmp's and create groups for those. Run Edit --> Plugins --> SimplifyGraph to bring up the plugin chooser. In this function nearly every isolated subgraph is a 7-node inlined strcmp implementation. Go through in the chooser to verify, and create a group. This results in a graph like in Figure 8.

Figure 8: Grouped strcmp

- When an input string matches a command string, the malware branches to code that implements the command. To further simplify the graph and make analysis easier run the Unique-Reachable algorithm on each separate command by right clicking on the first node of the command implementation and select SimplifyGraph --> Create unique-reachable group. After this we now have a graph as in Figure 9.

Figure 9: Grouped command logic

- Now perform your reverse engineering on each separate branch in the dispatch function. For each command handler group node that we created, right click that node and select "SimplifyGraph --> Complement & expand group". The result of complementing a single command handler node now looks like Figure 10 which is much easier to analyze.

Figure 10: Group complement

- When done analyzing the current command handler, delete the complement group by right clicking the "Complement of group X" node and use IDA's built-in "Ungroup nodes" command. Repeat for the remaining command handler grouped nodes.

Config

You can tweak some of the configuration by entering data in a file named %IDAUSR%/SimplifyGraph.cfg, where %IDAUSR% is typically %APPDATA%/Hex-Rays/IDA Pro/ unless explicitly set to something else. All of the config applies to the isolated subgraph component. Options:

- SUBGRAPH_HIGHLIGHT_COLOR: Default 0xb3ffb3: The color to apply to nodes when you double click/press enter in the chooser to show nodes that make up the currently selected isolated subgraph. Not everyone agrees that my IDA color scheme is best, so you can set your own highlight color here.
- MINIMUM_SUBGRAPH_NODE_COUNT: Default 3: The minimum number of nodes for a valid isolated subgraph. If a discovered subgraph has fewer nodes than this number it is not included in the shown list. This prevents trivial two-node subgraphs from being shown.
- MAXIMUM_SUBGRAPH_NODE_PERCENTAGE: Default 95: The maximum percent of group nodes (100.0 *(subgroup_node_count / total_function_node_count)) allowed. This filters out isolated subgraphs that make up (nearly) the entire function, which are typically not interesting.

Example SimplifyGraph.cfg contents:

```
"MINIMUM_SUBGRAPH_NODE_COUNT"=5
"MAXIMUM_SUBGRAPH_NODE_PERCENTAGE"=75
```

```
"SUBGRAPH_HIGHLIGHT_COLOR"=0x00aa1111
```

Prior work:

I came across semi-related work while working on this: GraphSlick from the 2014 Hex-Rays contest (https://www.hex-rays.com/contests/2014/index.shtml and https://github.com/lallousx86/GraphSlick). That plugin had different goals to automatically identifying (nearly) identical inline functions via CFG and basic block analysis, and patching the program to force mock function calls to the explicit function. It had a separate viewer to present information to the user.

SimplifyGraph is focused on automating tasks when doing manual reverse engineering (group creation) to reduce the complexity of disassembly in graph mode. Future work may incorporate the same prime-products calculations to help automatically find identical isolated subgraphs.

Installation

Prebuilt Windows binaries are available from the Releases tab of the GitHub project page. The zip files contains both IDA 32 and IDA 64 plugins for each of the new IDA 7.0 SDK and for the legacy IDA 6.95 SDK. Copy the two plugins for your version of IDA to the %IDADIR%\plugins directory.

Building

Windows

This plugin & related files were built using Visual Studio 2013 Update 5.

Environment Variables Referenced by project:

- IDASDK695: path to the extracted IDA 6.95 SDK. This should have `include` and `lib` paths beneath it.
- IDASDK: path to the extracted IDA 7.0 (or newer) SDK. This Should have `include` and `lib` paths beneath it.
- BOOSTDIR: path to the extracted Boost library. Should have `boost` and `libs` paths beneath it.

The easiest way is to use the Microsoft command-line build tools:

- For IDA7.0: Launch VS2013 x64 Native Tools Command Prompt, then run:

```
msbuild SimplifyGraph.sln /property:Configuration=ReleaseIDA70_32 /property:Platform=x64
msbuild SimplifyGraph.sln /property:Configuration=ReleaseIDA70_64 /property:Platform=x64
```

- For IDA6.95: Launch VS2013 x86 Native Tools Command Prompt, then run:

```
msbuild SimplifyGraph.sln /property:Configuration=ReleaseIDA695_32 /property:Platform=Win32
msbuild SimplifyGraph.sln /property:Configuration=ReleaseIDA695_64 /property:Platform=Win32
```

Linux

This plugin & related files have been built using GCC 6.3.0 and GCC 7.2.0. For x86_64 Linux you must install the multilib GCC packages.

Environment Variables Referenced by project:

- IDA_SDK: path to the extracted IDA SDK. This should have `include` and `lib` paths beneath it.
- IDA_DIR: path to your local IDA installation. This should have the `plugins` path beneath it.

Building and installing the plugin are done using GNU make:

- For IDA6.95: Open a terminal, then run:

```
IDA_SDK=/home/user/path/to/IdaSdk make all
IDA_DIR=/home/user/path/to/Ida6.95 make install
```

- Building against the IDA7.0 SDK is untested for Linux.

Other stuff

- The test directory contains some unit & system tests for the graph calculations. Run test.exe from the command line to check that all tests pass.
- cmd_graph_help contains a command line tool that implements the algorithms. Takes as input and produces as output GraphViz dot notation graphs. Used for testing and verification of algorithms, and generating graph pictures.

Appendix: Isolated subgraphs

Finding isolated subgraphs relies on calculating the immediate dominator and immediate post-dominator trees for a given function graph. The following is important to know:

- A node d dominates node n if every path to n must go through d.
- The immediate dominator p of node n is basically the closest dominator to n, where there is no node t where p dominates t, and t dominates n.
- A node z post-dominates a node n if every path from n to the exit node must go through z.
- The immediate post-dominator x of node n is the closest post-dominator, where there is no node t where t post-dominates n and x post-dominates t.

- The immediate dominator relationship forms a tree of nodes, where every node has an immediate dominator other than the entry node.
- The Lengauer-Tarjan algorithm can efficiently calculate the immediate dominator tree of a graph. It can also calculate the immediate post-dominator tree by reversing the direction of each edge in the same graph.

The plugin calculates the immediate dominator tree and immediate post-dominator tree of the function control flow graph and looks for the situations where the (idom[i] == j) and (ipdom[j] == i). This means all paths from the function start to node i must go through node j, and all paths from j to the function terminal must go through i. A candidate isolated subgraph thus starts at node j and ends at node i.

For each candidate isolated subgraph, the plugin further verifies only the entry node has predecessor nodes not in the candidate subgraph. The plugin also filters out candidate subgraphs by making sure they have a minimum node count and cover a maximum percentage of nodes (see MINIMUM_SUBGRAPH_NODE_COUNT and MAXIMUM_SUBGRAPH_NODE_PERCENTAGE in the config section).

One complication is that functions often have more than one terminal node – programmers can arbitrarily return from the current function at any point. The immediate post-dominator tree is calculated for every terminal node, and any inconsistencies are marked as indeterminate and are not possible candidates for use. Functions with infinite loops do not have terminal nodes, and are not currently handled.

For a simple example consider the graph in Figure 11.

Figure 11: Example graph

It has the following immediate dominator tree:

node	idom
0	None
1	0
2	1
3	1
4	3
5	3
6	3
7	6
8	0

It has the following immediate post-dominator tree:

node	ipdom
0	8
1	3
2	3
3	6
4	6
5	6
6	7
7	8
8	None

Looking for pairs of (idom[i] == j) and (ipdom[j] == i) gives the following: (0, 8) (1, 3) (3, 6) (6,7)

- (0, 8) is filtered because it makes up all of the nodes of the graph.
- (1,3) and (6, 7) are filtered out because they contain nodes reachable from nodes not in the set:
  - For (1, 3) node 2 is reachable from node 6.
  - For (6, 7) node 2 is reachable from node 1.
- This leaves (3, 6) as the only isolate subgraph in this example, shown in Figure 12.

Figure 12: Example graph with isolated subgraph

RetDec - by: Avast

Plugin description

> a RetDec plugin for IDA (provides RetDec-decompiled code views to IDA)

readme for RetDec

RetDec IDA plugin

RetDec plugin for IDA (Interactive Disassembler).

The plugin is compatible with the IDA 7.5+ versions. The plugin does NOT work with IDA 6.x, IDA 7.0-7.4, or freeware version of IDA 7.0. The plugin comes at both 32-bit and 64-bit address space variants (both are 64-bit binaries). I.e. it works in both `ida` and `ida64`. At the moment, it can decompile the following architectures:

- 32-bit: x86, arm, mips, and powerpc.
- 64-bit: x86-64, arm64.

Installation and Use

Currently, we officially support only Windows and Linux. It may be possible to build macOS version from the sources, but since we do not own a macOS version of IDA, we cannot create a pre-built package, or continually make sure the macOS build is not broken.

1. Either download and unpack a pre-built package from the latest release , or build and install the RetDec IDA plugin by yourself (the process is described below).
2. Follow the user guide (`user_guide.pdf`) that is part of the downloaded package, or use the current version from this repository.
3. Don't forget to install the required dependencies mentioned in the user guide.

Build and Installation

Requirements

Note: These are requirements to build the RetDec IDA plugin, not to run it. See our User Guide for information on plugin installation, configuration, and use.

- A compiler supporting C++17
  - On Windows, only Microsoft Visual C++ is supported (version >= Visual Studio 2017).
- CMake (version >= 3.6)
- IDA SDK (version >= 7.7)

Process

- Clone the repository:
  - `git clone https://github.com/avast/retdec-idaplugin.git`
- Linux:
  - `cd retdec-idaplugin`
  - `mkdir build && cd build`
  - `cmake .. -DIDA_SDK_DIR=<path>`
  - `make`
  - `make install` (if `IDA_DIR` was set, see below)
- Windows:
  - Open a command prompt (e.g. `C:\msys64\msys2_shell.cmd` from MSYS2)
  - `cd retdec-idaplugin`

  - `mkdir build && cd build`
  - `cmake .. -DIDA_SDK_DIR=<path> -G<generator>`
  - `cmake --build . --config Release -- -m`
  - `cmake --build . --config Release --target install` (if `IDA_DIR` was set, see below)
  - Alternatively, you can open `retdec-idaplugin.sln` generated by `cmake` in Visual Studio IDE.

You must pass the following parameters to `cmake`:

- `-DIDA_SDK_DIR=</path/to/idasdk>` to tell `cmake` where the IDA SDK directory is located.
- (Windows only) `-G<generator>` is `-G"Visual Studio 15 2017 Win64"` for 64-bit build using Visual Studio 2017. Later versions of Visual Studio may be used. Only 64-bit build is supported.

You can pass the following additional parameters to `cmake`:

- `-DIDA_DIR=</path/to/ida>` to tell `cmake` where to install the plugin. If specified, installation will copy plugin binaries into `IDA_DIR/plugins`, and content of `scripts/idc` directory into `IDA_DIR/idc`. If not set, installation step does nothing.
- `-DRETDEC_IDAPLUGIN_DOC=ON` to enable the `user-guide` target which generates the user guide document (disabled by default, the target needs to be explicitly invoked).

User Guide

The User Guide in a PDF form is located in `doc/user_guide/user_guide.pdf`.

You can build your own guide by enabling and invoking the `user-guide` target:

- `cmake .. -DRETDEC_IDAPLUGIN_DOC=ON`
- Linux: `make user-guide`
- Windows: `cmake --build . --config Release --target user-guide`
- Requires LaTeX , LaTeX packages, and related tools.
- The resulting PDF will overwrite the original `user_guide.pdf` in `doc/user_guide`.

License

RetDec IDA plugin uses third-party libraries or other resources listed, along with their licenses, in the `LICENSE-THIRD-PARTY` file.

Contributing

See RetDec contribution guidelines .

Milan's tools - by: Milan Bohacek

Plugin description

> api_palette.py: a code-searching\/completion tool, for IDA APIs member_type.py: automatically sets type to structure members, depending on their name paste_name.py: a handy plugin to rename decompiler-view names to whatever is in the clipboard by simp

readme for Milan's tools

api_palette

> Originally developed by Milan Bohacek .
>
> This plugin wins the IDA *Plug-In Contest 2017: Hall Of Fame* .

api_palette.py will be useful for those who write scripts for IDA (in the CLI or the script snippets window).

Changes

- Compatible with Python 3 and IDA 7.5
- Only show/search the first line of doc
- Only search api name and doc

Usage

The default shortcut is set to Shift + W.

Screenshot

---

lighthouse - by: Markus Gaasedelen

Plugin description

a code coverage plugin for IDA Pro. The plugin leverages IDA as a platform to map, explore, and visualize externally collected code coverage data when symbols or source may not be available for a given binary.

readme for lighthouse

Lighthouse - A Coverage Explorer for Reverse Engineers

Overview

Lighthouse is a powerful code coverage explorer for IDA Pro and Binary Ninja , providing software researchers with uniquely interactive controls to study execution maps for native applications without requiring symbols or source.

This project placed 2nd in IDA's 2017 Plug-In Contest and was later nominated in the 2021 Pwnie Awards for its contributions to the security research industry.

Special thanks to @0vercl0k for the inspiration.

Releases

- v0.9 -- Python 3 support, custom coverage formats, coverage cross-refs, theming subsystem, much more.
- v0.8 -- Binary Ninja support, HTML coverage reports, consistent styling, many tweaks, bugfixes.
- v0.7 -- Frida, C++ demangling, context menu, function prefixing, tweaks, bugfixes.
- v0.6 -- Intel pintool, cyclomatic complexity, batch load, bugfixes.
- v0.5 -- Search, IDA 7 support, many improvements, stability.
- v0.4 -- Most compute is now asynchronous, bugfixes.
- v0.3 -- Coverage composition, interactive composing shell.
- v0.2 -- Multifile support, performance improvements, bugfixes.
- v0.1 -- Initial release

Installation

Lighthouse is a cross-platform (Windows, macOS, Linux) Python 2/3 plugin. It takes zero third party dependencies, making the code both portable and easy to install.

Use the instructions below for your respective disassembler.

IDA Installation

1. From IDA's Python console, run the following command to find its plugin directory:
   - `import idaapi, os; print(os.path.join(idaapi.get_user_idadir(), "plugins"))`
2. Copy the contents of this repository's `/plugins/` folder to the listed directory.
3. Restart your disassembler.

Binary Ninja Installation

Lighthouse can be installed through the plugin manager on newer versions of Binary Ninja (>2.4.2918). The plugin will have to be installed manually on older versions.

Auto Install

1. Open Binary Ninja's plugin manager by navigating the following submenus:
   - `Edit -> Preferences -> Manage Plugins`
2. Search for Lighthouse in the plugin manager, and click the `Enable` button in the bottom right.
3. Restart your disassembler.

Manual Install

1. Open Binary Ninja's plugin folder by navigating the following submenus:
   - `Tools -> Open Plugins Folder...`
2. Copy the contents of this repository's `/plugins/` folder to the listed directory.
3. Restart your disassembler.

Usage

Once properly installed, there will be a few new menu entries available in the disassembler. These are the entry points for a user to load coverage data and start using Lighthouse.

Lighthouse is able to load a few different 'flavors' of coverage data. To generate coverage data that can be loaded into Lighthouse, please look at the README in the coverage directory of this repository.

Coverage Painting

While Lighthouse is in use, it will 'paint' the active coverage data across all of the code viewers available in the disassembler. Specifically, this will apply to your linear disassembly, graph, and decompiler windows.

In Binary Ninja, only the linear disassembly, graph, and IL views are supported. Support for painting decompiler output in Binary Ninja will be added to Lighthouse in the *near future* as the feature stabilizes.

Coverage Overview

The Coverage Overview is a dockable widget that will open up once coverage has been loaded into Lighthouse.

This interactive widget provides a function level view of the loaded coverage data. It also houses a number of tools to manage loaded data and drive more advanced forms of coverage analysis.

Context Menu

Right clicking the table in the Coverage Overview will produce a context menu with a few basic amenities to extract information from the table, or manipulate the database as part of your reverse engineering process.

If there are any other actions that you think might be useful to add to this context menu, please file an issue and they will be considered for a future release of Lighthouse.

Coverage ComboBox

Loaded coverage and user constructed compositions can be selected or deleted through the coverage combobox.

HTML Coverage Report

Lighthouse can generate rudimentary HTML coverage reports. A sample report can be seen here .

Coverage Shell

At the bottom of the coverage overview window is the coverage shell. This shell can be used to perform logic-based operations that combine or manipulate the loaded coverage sets.

This feature is extremely useful in exploring the relationships of program execution across multiple runs. In other words, the shell can be used to 'diff'

execution between coverage sets and extract a deeper meaning that is otherwise obscured within the noise of their individual parts.

Composition Syntax

Coverage composition, or *Composing* as demonstrated above is achieved through a simple expression grammar and 'shorthand' coverage symbols (A to Z) on the composing shell.

Grammar Tokens

- Logical Operators: `|, &, ^, -`
- Coverage Symbol: `A, B, C, ..., Z, *`
- Parenthesis: `(...)`

Example Compositions

1. Executed code that is *shared* between coverage `A` and coverage `B` :

```
A & B
```

2. Executed code that is *unique* only to coverage `A` :

```
A - B
```

3. Executed code that is *unique* to `A` or `B` , but not `C` :

```
(A | B) - C
```

Expressions can be of arbitrary length or complexity, but the evaluation of the composition may occur right to left. So parenthesis are suggested for potentially ambiguous expressions.

Hot Shell

Additionally, there is a 'Hot Shell' mode that asynchronously evaluates and caches user compositions in real-time.

The hot shell serves as a natural gateway into the unguided exploration of composed relationships.

Search

Using the shell, you can search and filter the functions listed in the coverage table by prefixing their query with `/` .

The head of the shell will show an updated coverage % computed only from the remaining functions. This is useful when analyzing coverage for specific function families.

Jump

Entering an address or function name into the shell can be used to jump to corresponding function entries in the table.

Coverage Cross-references (Xref)

While using Lighthouse, you can right click any basic block (or instruction) and use the 'Coverage Xref' action to see which coverage sets executed the selected block. Double clicking any of the listed entries will instantly switch to that coverage set.

This pairs well with the 'Coverage Batch' feature, which allows you to quickly load and aggregate thousands of coverage files into Lighthouse. Cross-referencing a block and selecting a 'set' will load the 'guilty' set from disk as a new coverage set for you to explore separate from the batch.

Themes

Lighthouse ships with two default themes -- a 'light' theme, and a 'dark' one. Depending on the colors currently used by your disassembler, Lighthouse will attempt to select the theme that seems most appropriate.

The theme files are stored as simple JSON on disk and are highly configurable. If you are not happy with the default themes or colors, you can create your own themes and simply drop them in the user theme directory.

Lighthouse will remember your theme preference for future loads and uses.

Future Work

Time and motivation permitting, future work may include:

- ~~Asynchronous composition, painting, metadata collection~~
- ~~Multifile/coverage support~~
- Profiling based heatmaps/painting
- Coverage & profiling treemaps
- ~~Additional coverage sources, trace formats, etc~~
- Improved pseudocode painting
- ~~Lighthouse console access~~ , headless usage
- ~~Custom themes~~
- ~~Python 3 support~~

I welcome external contributions, issues, and feature requests. Please make any pull requests to the `develop` branch of this repository if you would like them to be considered for a future release.

Authors

- Markus Gaasedelen (@gaasedelen)

IDABuddy - by: Tamir Bahar

Plugin description

> ...is a reverse-engineer's best friend. Designed to be everything Clippy the Office Assistant was, and more! IDABuddy will always be there for you. Friendly and helpful while you work. Offering tips and friendly chat. And best of all - since it is op

readme for IDABuddy

IDABuddy

What is it?

IDABuddy is a reverse-engineer's best friend. Designed to be everything Clippy the Office Assistant was, and more!

IDABuddy will always be there for you. Friendly and helpful while you work. Offering tips and friendly chat.

And best of all - since it is open-source, it will never be taken away from you!

Requirements

1. Sark
2. Sark's plugin-loader plugin
3. PyYAML

Installation

1. Clone the IDABuddy repository
2. Add `idabuddy\idabuddy_plugin.py` to your `plugins.list` (`%AppData%\HexRays\plugins.list` or `C:\Program Files (x86)\IDA 6.9\cfg\plugins.list`).
3. Launch IDA

Configuration

What can I say?

IDABuddy's messages are stored in `sayings.yml` .

Basic sayings are lists of strings, spoken out one after the other.

Address sayings are single strings, with `{}` in them to denote the location to put the address to jump to.

How do I look?

Visual settings are stored in `config.yml` , as well as documentation.

Available avatars: Or create your own!

In The Press

IDABuddy was presented in a lightning talk at 32c3. Watch the video here .

Drop - by: Thomas Rinsma

Plugin description

> an experimental IDA Pro plugin capable of detecting several types of opaque predicates in obfuscated binaries. It leverages the power of the symbolic execution engine angr and its components to reason about the opaqueness of predicates based on their

readme for Drop

Experimental opaque predicate detection for IDA Pro

Drop (D rop R emoves O paque P redicates) is an experimental IDA Pro plugin capable of detecting several types of opaque predicates in obfuscated binaries by making use of the symbolic-execution engine *angr* and its components. Specifically, Drop can detect, highlight and (primitively) remove the following types of opaque predicates:

- invariant opaque predicates (i.e. "normal" opaque predicates without context)

- contextual opaque predicates (i.e. opaque predicates that depend on one or more invariants at the predicate location)

In general, the plugin is built to be as interactive as possible, allowing the analyst to specify additional context through function hooking, symbolic global variables and additional (in-)equality constraints.

Disclaimers

- This code was written during an internship. It is not a Riscure product and Riscure does not support or maintain this code.
- This is experimental code, intended as an experiment to see what can be accomplished by combining `angr` and IDA Pro for this purpose.
  - Because of certain heuristics, the plugin will in various scenarios result in false positives, false negatives, or both.
  - In certain (often complex) functions, SMT constraints will become very large, and solving time might become unreasonably large. Drop provides a button to kill the current analysis, but because of Z3's architecture, this can occasionally kill IDA itself as well. It is therefore recommended you save your database before performing any heavy analysis.

Third-party dependencies

Because of the instable nature of the APIs provided by angr and its components, Drop requires a very specific version of each to be installed. In order to make installation easier, some of these have been provided as Python `.egg` files in the `dependencies` folder. See the *Installation* section below for instructions on how to install these dependencies.

Installation

This assumes a 64-bit Windows 7 installation with IDA 6.95. Other operating systems are not tested an will require a different installation procedure. Currently, IDA 7.0 and 64-bit Python are not supported. This might change in the future.

It is assumed that (32-bit) Python 2.7, `pip` and `easy_install` are installed, as they come with IDA 6.95.

1. Make sure the 32-bit Python 2.7 executable directories are in your PATH:
   - `;C:\Python27;C:\Python27\Scripts`
2. Run the following commands:
   - `cd path_to_drop/dependencies`
   - `pip install -r requirements.txt`

- `easy_install -Z archinfo-6.7.1.13-py2.7.egg pyvex-6.7.1.31-py2.7.egg cle-6.7.1.31-py2.7.egg simuvex-6.7.1.31-py2.7.egg capstone-3.0.4-py2.7.egg angr-6.7.1.31-py2.7.egg`

3. Copy `drop/` and `drop.py` to `plugins/` in your IDA installation folder.
4. Done!

Basic usage

The following video shows Drop in action on a simple function containing the opaque predicate `7*x*x-1 != y*y` : https://streamable.com/s7fjw . The source code of the program seen in that video can be found in the demo folder.

In general, the workflow with Drop is is follows:

1. Open a binary* in IDA Pro.
2. Make sure the cursor is located within a function.
3. Launch Drop with Alt+F6.
4. Follow the steps shown in the panel:
   i. Let angr perform a rudimentary concrete trace through the function by pressing ' *concrete trace...* ' or manually mark code-blocks to include in the analysis.
   ii. Click the ' *along trace* ' button to start opaque predicate detection on the currently marked basic-blocks.
   iii. Optionally (and very experimentally), patch any resulting unreachable code with NOP instructions, to hopefully simplify the function's graph view and decompilation output.

Optionally, one can specify context during step 2 to improve symbolic analysis, this is especially helpful in larger functions with many variables or function calls.

* only x86 binaries have been properly tested.

BinCAT - by: Sarah Zennou, Philippe Biondi, Raphaël Rigo, Xavier Mehrenberger

Plugin description

a static Binary Code Analysis Toolkit, designed to help reverse engineers, directly from IDA.

readme for BinCAT

Introduction

What is BinCAT?

BinCAT is a *static* Binary Code Analysis Toolkit, designed to help reverse engineers, directly from IDA or using Python for automation.

It features:

- value analysis (registers and memory)
- taint analysis
- type reconstruction and propagation
- backward and forward analysis
- use-after-free and double-free detection

In action

You can check (an older version of) BinCAT in action here:

- Basic analysis
- Using data tainting

Check the tutorial out to see the corresponding tasks.

Quick FAQ

Supported host platforms:

- IDA plugin: all, version 7.4 or later (Only Python 3 is supported)
- analyzer (local or remote): Linux, Windows, macOS (maybe)

Supported CPU for analysis (for now):

- x86-32
- x86-64
- ARMv7
- ARMv8
- PowerPC

Installation

Only IDA v7.4 or later is supported

Older versions may work, but we won't support them.

Binary distribution install (recommended)

The binary distribution includes everything needed:

- the analyzer
- the IDA plugin

Install steps:

- Extract the binary distribution of BinCAT (not the git repo)
- In IDA, click on "File -> Script File..." menu (or type ALT-F7)
- Select `install_plugin.py`
- BinCAT is now installed in your IDA user dir
- Restart IDA

Manual installation

Analyzer The analyzer can be used locally or through a Web service.

On Linux:

- Using Docker: Docker installation instructions
- Manual: build and installation instructions

On Windows:

- build instructions

IDA Plugin

- Windows manual install .
- Linux manual install

BinCAT should work with IDA on Wine, once pip is installed:

- download https://bootstrap.pypa.io/get-pip.py (verify it's good ;)
- `~/.wine/drive_c/Python/python.exe get-pip.py`

Using BinCAT

Quick start

- Load the plugin by using the `Ctrl-Shift-B` shortcut, or using the `Edit -> Plugins -> BinCAT` menu
- Go to the instruction where you want to start the analysis
- Select the `BinCAT Configuration` pane, click `<-- Current` to define the start address
- Launch the analysis

Configuration

Global options can be configured through the `Edit/BinCAT/Options` menu.

Default config and options are stored in `$IDAUSR/idabincat/conf` .

Options

- "Use remote bincat": select if you are running docker in a Docker container
- "Remote URL": http://localhost:5000 (or the URL of a remote BinCAT server)
- "Autostart": autoload BinCAT at IDA startup
- "Save to IDB": default state for the `save to idb` checkbox

Documentation

A manual is provided and check here for a description of the configuration file format.

A tutorial is provided to help you try BinCAT's features.

Article and presentations about BinCAT

- SSTIC 2017 , Rennes, France: article (english), slides (french), video of the presentation (french)
- REcon 2017 , Montreal, Canada: slides , video

Licenses

BinCAT is released under the GNU Affero General Public Licence .

The BinCAT OCaml code includes code from the original Ocaml runtime, released under the LGPLv2 .

The BinCAT IDA plugin includes code from python-pyqt5-hexview by Willi Ballenthin, released under the Apache License 2.0.

BinCAT includes a modified copy of newspeak .

Automated builds

Windows

Automated builds are performed automatically (see azure-pipelines.yml). The latest builds and test results can be accessed here

Linux

Automated builds are performed automatically using GitHub Actions (see here), results can be obtained on GitHub's Actions tab.

NIOS2 - by: Anton Dorfman

Plugin description

> IDA Pro processor module for Altera Nios II Classic\/Gen2 microprocessor architecture

readme for NIOS2

IDA Pro NIOS2 Processor Module

This project has been forked from ptresearch - nios2 . The original repo has been archived by the owner on Sep 22, 2022. It is now read-only.
In this repo I have updated IDAPython APIs; thus, this project can be used on IDA Pro 7.4+ and IDA Pro 8.0+.

Definition NIOS2 - this is an IDA Pro processor module for Altera Nios II Classic/Gen2 microprocessor architecture.

Requirements For IDA Pro versions before 7.3 you need Python 2.7, IDA 6.9, and IDAPython 1.7.0. Older versions should also work. IDA versions 7.0+ supported.
For IDA Pro 7.4 and above you need Python 3.8 and above. By IDA Pro 7.4 and later Hex-rays has changed API on IDAPython. But the proc module under folder ida75 works on IDA Pro versions v7.4 and above including versions 8 and above.

Installation Copy the file **nios2.py** to the procs subfolder of your IDA Pro installation.
For example in Windows:

- location for IDA Pro 6.xx: "\Program Files (x86)\IDA 6.xx\procs"
- location for IDA Pro 7.xx: "\Program Files\IDA 7.xx\procs"
- location for IDA Pro 8.xx: "\Program Files\IDA 8.xx\procs"

How to use Launch IDA Pro, select the Altera Nios II Classic/Gen2 Processor processor type, and enjoy the reverse engineering goodness!
NIOS II processor module: feature description

Key features Decodes instructions and operands, and displays them on screen. Generates comments for commands. Describes both general-purpose registers and control registers. Analyzes execution control instructions. Generates cross-references. Generates references from data also. Simplifies instructions: replaces certain combinations of commands and operands with pseudoinstructions (commands for which there are no separate opcodes). Monitors changes in stack pointer and supports stack variables. Handles situations when the stack pointer is involved in calculating offsets written to other registers by converting to the offsets for stack variables. Generates cross-references from code to separate fields of structures.

32-bit numbers and offsets The NIOS II processor does not have a machine command for directly writing a 32-bit value to a register. At maximum, a 16-bit value can be written in a single command. Therefore, writing 32-bit numbers consists of two steps: the high half of the number is written to the high part of the register, while the low part is added, subtracted, or combined with the high part with the help of bitwise OR. If the 32-bit number is an offset, the low half can also be implemented in a command as a positive or negative offset relative to the base (the high half).
In the processor module, if a 32-bit number is an offset, an operand and cross-references are generated only from the low half, using the base taken from the high half. Setting a register to an offset, as well as reading or writing relative to the base, is handled. If a 32-bit number is not an offset, its value is simply output next to the command for writing the low half.
Switch
All encountered schemes for organizing switch constructions are handled. The module recognizes switch variants when the scheme is interrupted by jumps, when instructions not part of the scheme are encountered between the main commands, and when the locations of main commands have been switched. A reverse execution path approach is used that takes into account possible jumps, with setting of internal variables that signal various states of the recognizer. In practice, the module successfully recognizes around 10 different switch organization variants.
The custom instruction
The NIOS II architecture includes ”custom”, an interesting instruction that gives access to 256 user-set instructions and can access a set of 32 custom registers. The processor module implements support for the custom instruction and outputting command names for the FloatingPointHardware 2 (FPH2) component.

Jumping by register value Recognizes jumps by register value, with pre-writing of the offset to the register. Generates cross-references and outputs the name of the procedure or label next to the name of the jump command.

Addressing via global pointer The value of the global pointer in the gp register is determined in the background as the file is initially opened and navigated. The value of gp is saved in the idb database upon closing, and restored during loading. Variables that are addressed relative to gp in loading and save instructions are converted to offsets. Offset conversions are also performed when the gp register is involved in calculating offsets written to other registers. Thus the other register is set relative to the gp register for a certain region of data.

Supported IDA Versions

Version	isSupported
6.9	
7.0 - 7.3	
7.4 - 7.9	
8.0 - 8.3	

Author Anton Dorfman ADorfman@ptsecurity.com (Positive Technologies)

Contributors Blue DeviL // SCT

IDArling - by: Alexandre Adamski and Joffrey Guilbon

Plugin description

a collaborative reverse engineering plugin for IDA Pro and Hex-Rays.

readme for IDArling

Warning

This project is no longer under active development and the more featured and up-to-date fork is probably something more interesting for new comers. Also, IDA has announced an official support for collaborative reverse engineering session and one could also wait for this.

Overview

IDArling is a collaborative reverse engineering plugin for IDA Pro and Hex-Rays . It allows to synchronize in real-time the changes made to a database by multiple users, by connecting together different instances of IDA Pro.

The main features of IDArling are:

- hooking general user events
- structure and enumeration support
- Hex-Rays decompiler syncing
- replay engine and auto-saving
- database loading and saving
- interactive status bar widget
- user cursors (instructions, functions, navbar)
- invite and following an user moves
- dedicated server using Qt5
- integrated server within IDA
- LAN servers discovery
- following an user moves in real time

If you have any questions not worthy of a bug report, feel free to ping us at #idarling on freenode and ask away.

Releases

This project is under active development. Feel free to send a PR if you would like to help! :-)

It is not really stable in its current state, please stayed tuned for a first release of the project!

Installation

Install the IDArling client into the IDA plugins folder.

- Copy `idarling_plugin.py` and the `idarling` folder to the IDA plugins folder.
  - On Windows, the folder is at `C:\Program Files\IDA 7.x\plugins`
  - On macOS, the folder is at `/Applications/IDA\ Pro\ 7.x/idabin/plugins`
  - On Linux, the folder may be at `~/ida-7.x/plugins/`
- Alternatively, you can use the "easy install" method by copying the following line into the console:

```
import urllib2; exec(urllib2.urlopen('https://raw.githubusercontent.com/IDArlingTeam/IDArlin
```

Warning: The plugin is only compatible with IDA Pro 7.x on Windows, macOS, and Linux.

The dedicated server requires PyQt5, which is integrated into IDA. If you're using an external Python installation, we recommand using Python 3, which offers a pre-built package that can be installed with a simple `pip install PyQt5` .

Usage

Open the *Settings* dialog accessible from the right-clicking the widget located in the status bar. Show the servers list by clicking on the *Network Settings* tabs and add your server to it. Connect to the server by clicking on it after right-clicking the widget again. Finally, you should be able to access the following menus to upload or download a database:

```
- File --> Open from server
- File --> Save to server
```

Thanks

This project is inspired by Sol[IDA]rity . It started after contacting its authors and asking if it was ever going to be released to the public. Lighthouse source code was also carefully studied to understand how to write better IDA plugins.

- Previous plugins, namely CollabREate , IDASynergy , YaCo , were studied during the development process;
- The icons are edited and combined versions from the sites freeiconshop.com and www.iconsplace.com .

Thanks to Quarkslab for allowing this release.

Authors

- Alexandre Adamski < neat@idarling.re >
- Joffrey Guilbon < patate@idarling.re >

IDA-Minsc - by: Ali Rizvi-Santiago

Plugin description

> IDA-minsc is a plugin that extends IDAPython with a DWIM syntax. This wraps various IDA features like names, xrefs, instructions, operands, structs, types, etc. into an api that simplifies annotating or exchanging of artifacts from IDA and Hex-Rays.

readme for IDA-Minsc

- Website: https://github.com/arizvisa/ida-minsc
- Documentation: https://arizvisa.github.io/ida-minsc
- Questions (or scripting help): https://github.com/arizvisa/ida-minsc/discussions
- Wiki: https://github.com/arizvisa/ida-minsc/wiki
- Changelog: https://github.com/arizvisa/ida-minsc/wiki/Changelog
- Complaints: https://github.com/arizvisa/ida-minsc/issues

General

IDA-minsc is a plugin for IDA Pro that assists a user with scripting the IDAPython plugin that is bundled with the disassembler. This plugin groups the different aspects of the IDAPython API into a simpler model which allows a reverse engineer to automate different aspects of their work with very little investment. Although the syntax is still Python, the required knowledge of IDA's intricacies is mostly abstracted away and reduced so that most automation can be done within a few lines of Python. This is done by prioritizing the primary types that a reverse-engineer may use (an address or a name) throughout the entire api. The concept of Python's classes are mostly done away with in favor of grouping related functionality within their own namespace. By doing this, the usage of the Python keywords "`help($whatever)`", "`dir($whatever)`" or IDAPython's "`?`" command are now useful again.

A number of concepts are introduced such as an indexed tagging system, support for multiple cases of functions (to avoid object-oriented design), tools for management of events and hooks, wrappers for interacting with structures/unions/frames, and modification of general usage artifacts within the database such as cross-references, enumerations, operands (values and access), and types. The intent of this is that most search and annotation issues can be performed with just a few lines of code and many things are now serializeable which can be used to share information between databases. This should enable a user to write quick scripts that can be used to augment their reversing endeavors avoiding the distraction of having to context-switch into "programming-mode" to perform mass-annotations or searches.

Notes

This plugin aims for backwards compatibility with IDA Pro 6.8 and should work up to the most recent release. In some instances the performance may be different due to missing features of older versions of IDA. Some examples of using the plugin to solve smaller scripting problems can be found within the wiki under the "Examples" at https://github.com/arizvisa/ida-minsc/wiki/

At the moment, the .zip file that is hosted on this page is just a snapshot of the most recent version of the plugin's "master" branch. At the moment, an ongoing refactor is occurring to integrate more specific Hex-Rays functionality (other than just types) and a better constraint system for determining which case of a function is a better candidate for the parameters you provide. Hence, the most recent version can always be found in the git repository at https://github.com/arizvisa/ida-minsc . If you are interested in these other additions, however, ongoing development and testing is currently happening within the "persistence-refactor" branch at https://github.com/arizvisa/ida-minsc/tree/persistence-refactor .

Changelog

Until further notice, the changelog can be found at https://github.com/arizvisa/ida-minsc/wiki/Changelog .

Installation

This plugin introduces a number of globally available Python modules and as a result there are two ways to install this plugin which depends on whether you want access to the plugin from other plugins, when you startup IDA, or only when you open up database. For more specific details, you can view the official installation directions via the documentation at http://arizvisa.github.io/ida-minsc/install.html .

Installation (zip)

Simply unzip the .zip file into your user plugins directory.

This is documented at https://hex-rays.com/blog/igors-tip-of-the-week-103-sharing-plugins-between-ida-installs/

Installation (plugins directory)

In summary, you will be performing the following steps.

1. Ensure that Git is installed, IDA Pro works and you can use IDAPython. 2. Using Git, clone the repository at https://github.com/arizvisa/ida-minsc in a directory of your choosing. 3. Install the Python requirements using the Python package manager (pip) with the "requirements.txt" file in the plugin directory. 4. Edit the "plugins/minsc.py" file in the plugin directory so that it points at whatever directory you cloned the repository into. 5. Deploy your modified "plugins/minsc.py" file into the "plugins" sub-directory of your IDA user directory. 6. Run IDA Pro and ensure that the plugin is listed and works.

Installation (globally)

This method utilizes the `idapythonrc.py` file to execute the plugin during IDA's startup process and consists ofcloning the repository directly into the user's IDA user directory. On the Windows platform, this is typically located at `$APPDATA/Roaming/Hex-Rays/IDA Pro`. Whereas on the Linux platform this can be found at `$HOME/.idapro`. This contents of this repository should actually replace that directory. If you have any files that presently reside there, simply move them into the repository's directory. After installation, IDA Pro should load its IDAPython plugin which should result in the `idapythonrc.py` belonging to IDA-minsc being executed which will then replace IDAPython's default namespace with the one belonging to the plugin's.

To clone the repository in a directory `$TARGET`, one can simply do:

```
$ git clone https://github.com/arizvisa/ida-minsc "$TARGET"
```

After the `idapythonrc.py` gets executed the modules that are available can be listed using "`dir()`" at the IDAPython command line.

Installation (python dependencies)

After cloning the repository, the user will need to install its required Python dependencies into their site-packages. This can be done using `pip` which is a tool that is bundled with Python. The file that contains the user's requirements is in the root of the repository as `requirements.txt`.

To install the required Python dependencies, one can run `pip` as so:

```
$ pip install -r 'requirements.txt'
```

At this point when the user starts IDA Pro, IDA-minsc will replace IDAPython's namespace with its own at which point can be used immediately. To verify that IDA-minsc was installed properly, one can simply type in the following at the IDAPython prompt:

```
> database.config.version()
```

This should then return the number `0` since no database has been loaded.

Quick Start

After installing the python dependencies, starting up IDA, and opening up a database you can do something like the following to list all the functions in your database:

```
> database.functions.list("sub_*")
> database.exports.list(ordinal=[1,2,3,4,5], like="sub_*")
> database.imports.list(module="*kernel*")
```

Or to iterate through all the functions in the database, you can try:

```
> for ea in database.functions():
    print(hex(ea))
```

```
> for ea in database.exports(like="sub_*", typed=False):
    print(hex(ea), function.name(ea))
```

To display information about the database itself you can use:

```
> print(database.path(), database.filename())
> print(database.idb())
```

If you find a function and navigate to it, you can iterate through each instruction and print out their operands with:

```
> for blk in function.blocks():        # note: if you don't give it a parameter, it uses
    print(database.disasm(blk), instruction.ops(blk))
```

You can also select a range of addresses in the database and do one of the following:

```
> print(database.read())
> print(database.address())
> print(database.address.bounds())
```

Please use "`help()`" or refer to the documentation at https://arizvisa.github.io/ida-minsc for more details on what this plugin makes available to you.

Documentation

Comprehensive documentation is available at the project page on https://arizvisa.github.io/ida-minsc , or can be built locally via the " docs " branch.

If the user wishes to build documentation for local use, they will first need to install the Sphinx package. Afterwards, the entirety of the documentation resides within in the " docs " branch. Simply checkout the branch, change the directory to "docs", and then run GNU make as:

```
$ make html
```

This will result in the build system parsing the available modules and then rendering all of the documentation into the `_build` directory relative to the `docs/Makefile` . Documentation can be generated for a number of different formats. To list all of the available formats, type in `make help` at the command prompt.

Plugins

It is recommended that this plugin is used with a notebook interface such as that provided by IPyIDA (https://github.com/eset/ipyida) and discussed in detail at https://hex-rays.com/blog/plugin-focus-ipyida/ . Although this plugin provides capabilities for exporting artifacts from a database, importing artifacts from some information source, or exchanging artifacts between databases that are part of the same project, a more complete solution for sharing information between instances of a database can be found at https://github.com/binsync/binsync .

Contributing

For any issues or feature suggestions, please visit the issue tracker over at https://github.com/arizvisa/ida-minsc/issues .

Thanks

Thanks to a number of anonymous and non-anonymous people whom have helped with the development of this plugin over all of these years.

IDAFuzzy - by: Ga-Ryo

Plugin description

a fuzzy search tool for IDA Pro. This tool helps you to find command\/function\/struct and so on. This tool is inspired by Mac's Spotlight and Intellij's Search Everywhere dialog.

readme for IDAFuzzy

Hyara - by: Yi Hyun, Kwak Kyoung-Ju

Plugin description

a plugin to create pattern-matching rules.

readme for Hyara

HeapViewer - by: Daniel García Gutiérrez

Plugin description

an IDA Pro plugin to examine the heap (glibc malloc implementation), focused on exploit development.

readme for HeapViewer

ActionScript 3 - by: Boris Larin

Plugin description

… an ActionScript 3 processor module and Flash debugger plugin.

readme for ActionScript 3

ActionScript3 IDA Pro

Hex-Rays IDA Pro Plug-In Contest 2018

Author: Boris Larin

This repository contains the SWF Loader, ActionScript3 processor module, and a debugger assist plugin named KLFDB.

Requirements

IDA Pro 7.1 (Tested with IDA Pro 7.1.180227)

Installation

Copy files into the IDA Pro directory:

- 'swf.py' to 'loaders' subfolder
- 'klfdb.py' to 'plugins' subfolder
- 'as3.py' to 'procs' subfolder

Usage

Drag and drop the SWF file to IDA Pro and select the Shockwave Flash loader.

Use 'File' -> 'Produce file' -> 'Create MAP file...' to generate a map file for use with KLFDB.

KLFDB is written to work with 32-bit versions of Stand Alone Flash and with Flash for Browsers (Internet Explorer is currently supported).

To debug the SWF file with Internet Explorer, load the Adobe Flash module (e.g. c:\Windows\System32\Macromed\Flash\Flash32_ _ * *.ocx) into IDA Pro.

Use 'Edit' -> 'Klfdb' -> 'Load new map file' to load the generated map file.

From this point, it is possible to use 'Edit' -> 'Klfdb' -> 'Set breakpoints on ...' to set breakpoints on methods.

After setting breakpoints, attach to the Internet Explorer process that is about to start the SWF file and use 'Edit' -> 'Klfdb' -> 'Run'. After that, allow the Flash file to execute.

The plugin will suspend execution of Adobe Flash after the breakpoint hit and will transparently fill just-in-time compiled native code with useful comments about the original bytecode.

Acknowledgements

- RABCDAsm
- JPEXS Free Flash Decompiler

Virtuailtor - by: Gal Zaban

Plugin description

an IDAPython tool for C++ vtables reconstruction on runtime.

readme for Virtuailtor

Virtuailor - IDAPython tool for C++ vtables reconstruction

Virtuailor is an IDAPython tool that reconstructs vtables for C++ code written for intel architecture, both 32bit and 64bit code and AArch64 (New!). The tool constructed from 2 parts, static and dynamic.

The first is the static part, contains the following capabilities:

- Detects indirect calls.
- Hooks the value assignment of the indirect calls using conditional breakpoints (the hook code).

The second is the dynamic part, contains the following capabilities:

- Creates vtable structures.
- Rename functions and vtables addresses.
- Add structure offset to the assembly indirect calls.
- Add xref from indirect calls to their virtual functions(multiple xrefs).
- For AArch64- tries to fix undefined vtables and related virtual functions (support for firmware).

How to Use?

> Virtuailor now supports IDA versions from 7.0 to the newest version (7.5), if you are using IDA versions older than 7.4 you will need to switch to branch beforeIDA-7.4, master branch supports the newest version available (7.5).

1. By default Virtuailor will look for virtual calls in ALL the addresses in the code. If you want to limit the code only for specific address range, no problem, just edit the *Main* file to add the range you want to target in the variables start_addr_range and end_addr_range:

```
if __name__ == '__main__':
```

```
start_addr_range = idc.MinEA()  # You can change the virtual calls address range
end_addr_range = idc.MaxEA()
add_bp_to_virtual_calls(start_addr_range, end_addr_range)
```

2. Optional, (but extremely recommended), create a snapshot of your idb. Just press ctrl+shift+t and create a snapshot.
3. Press File->Run script... then go to Virtuailor folder and choose to run Main.py, You can see the following gif for a more clear and visual explanation.

Now the GUI will provide you an option to choose a range to target, in case you would like to target all the binary just press OK with the default values in the start and end addresses.

Afterwards the breakpoints will be placed in your code and all you have to do is to execute your code with IDA debugger, do whatever actions you want and see how the vtables is being built! For AArch64 you can setup a remote gdb server and debug using the IDA debuggger.

In case you don't want/need the breakpoints anymore just go to the breakpoint list tab in IDA and delete the breakpoints as you like.

It is also really important for me to note that this is the second version of the tool with both 32 and 64 bit support and aarch64, probably in some cases a small amount of breakpoints will be missed, in these cases please open an issue and contact me so I will be able to improve the code and help fixing it. Thank you in advanced for that :)

Output and General Functions

vtables structures The structures Virtuailor creates from the vtable used in virtual call that were hit. The vtable functions are extracted from the memory based on the relevant register that was used in the BP opcode.

Since I wanted to create a correlation between the structure in IDA and the vtables in the data section, the BP changes the vtable address name in the data section to the name of the structure. As you can see in the following picture:

The virtual functions names are also being changed, take aside situations where the names are not the default IDA names (functions with symbols or functions that the user changed) in those cases the function names will stay the same and will also be add to the vtable structure with their current name.

The chosen names is constructed using the following pattern:

- vtable_
- vfunc_ the rest of the name is either offset from the beginning of the Segment, this is mostly because most binaries nowadays are PIE and PIC and thus ASLR is enforced, (instead of using the full address name, which is also quite long on 64bit environments). The vtable structure also has a

comment, ”Was called from offset: XXXX”, this offset is the offset from the beginning of the Segment.

Adding Structures to the Assembly After creating the vtable Virtuailor also adds a connection between the structure created and the assembly as you can see in the following images:

P.S: The structure offset used in the BP is only relevant for the last call that was made, in order to get a better understanding of all the virtual calls that were made the xref feature was added as explained in the next section

Xref to virtual functions When reversing C++ statically it is not trivial to see who called who, this is because most calls are indirect calls, however after running Virtuailor every function that was called indirectly now has an xref to those locations.

The following gif shows the added Xrefs with their indirect function call:

Former talks and lectures

The tool was presented in RECon brussels, Troopers and Warcon. The presentation could be found in the following link: https://www.youtube.com/watch?v=Xk75TM7NmtA

Thanks

REcon Brussels, Troopers, Warcon crews, Nana, @tmr232, @matalaz, @oryandp, @talkain, @shiftreduce

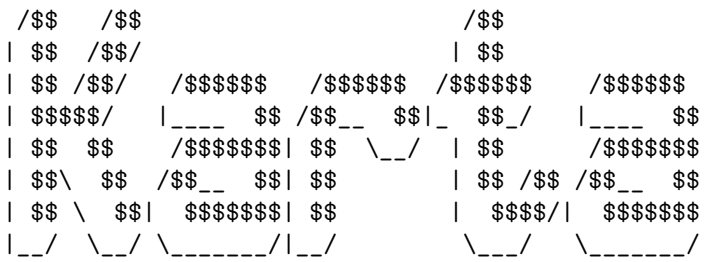
License

Karta - by: Eyal Itkin

Plugin description

> an IDA Python plugin that identifies and matches open-sourced libraries in a given binary. The plugin uses a unique technique that enables it to support huge binaries (>200,000 functions), with almost no impact on the overall performance.

readme for Karta



Purpose

"Karta" (Russian for "Map") is an IDA Python plugin that identifies and matches open-sourced libraries in a given binary. The plugin uses a unique technique that enables it to support huge binaries (>200,000 functions), with almost no impact on the overall performance.

The matching algorithm is location-driven. This means that it's main focus is to locate the different compiled files, and match each of the file's functions based on their original order within the file. This way, the matching depends on K (number of functions in the open source) instead of N (size of the binary), gaining a significant performance boost as usually N » K.

We believe that there are 3 main use cases for this IDA plugin:

1. Identifying a list of used open sources (and their versions) when searching for a useful 1-Day
2. Matching the symbols of supported open sources to help reverse engineer a malware
3. Matching the symbols of supported open sources to help reverse engineer a binary / firmware when searching for 0-Days in proprietary code

Read The Docs

https://karta.readthedocs.io/

Installation (Python 3 & IDA >= 7.4)

For the latest versions, using Python 3, simply git clone the repository and run the `setup.py install` script. Python 3 is supported since versions v2.0.0 and above.

Installation (Python 2 & IDA < 7.4)

As of the release of IDA 7.4, Karta is only actively developed for IDA 7.4 or newer, and Python 3. Python 2 and older IDA versions are still supported using the release version v1.2.0, which is most probably going to be the last supported version due to python 2.X end of life.

Identifier

Karta's identifier is a smaller plugin that identifies the existence, and fingerprints the versions, of the existing (supported) open source libraries within the binary. No more need to reverse engineer the same open-source library again-and-again, simply run the identifier plugin and get a detailed list of the used open sources. Karta currently supports more than 10 open source libraries, including:

- OpenSSL
- Libpng
- Libjpeg
- NetSNMP
- zlib
- Etc.

Matcher

After identifying the used open sources, one can compile a .JSON configuration file for a specific library (libpng version 1.2.29 for instance). Once compiled, Karta will automatically attempt to match the functions (symbols) of the open source in the loaded binary. In addition, in case your open source used external functions (memcpy, fread, or zlib_inflate), Karta will also attempt to match those external functions as well.

Folder Structure

- src: source directory for the plugin
- configs: pre-supplied *.JSON configuration files (hoping the community will contribute more)
- compilations: compilation tips for generating the configuration files, and lessons from past open sources
- docs: sphinx documentation directory

Additional Reading

- https://research.checkpoint.com/karta-matching-open-sources-in-binaries/
- https://research.checkpoint.com/thumbs-up-using-machine-learning-to-improve-idas-analysis

Credits

This project was developed by me (see contact details below) with help and support from my research group at Check Point (Check Point Research).

Contact (Updated)

This repository was developed and maintained by me, Eyal Itkin, during my years at Check Point Research. Sadly, with my departure of the research group, I will no longer be able to maintain this repository. This is mainly because of the long list of requirements for running all of the regression tests, and the IDA Pro versions that are involved in the process.

Please accept my sincere apology.

@EyalItkin

---

idapkg - by: jinmo123

Plugin description

> Packages for IDA Pro

readme for idapkg

Package manager for IDA Pro

> WARNING: This project is still between alpha and beta state. Feel free to report bugs if this is not working!

How to install

Execute the script below in IDAPython console (minified `installer.py` .)

```
import zipfile,tempfile,sys,os,threading,shutil,importlib
def install():P=os.path;tag='v0.1.4';n=tempfile.NamedTemporaryFile(delete=False,suffix='.zip
threading.Thread(target=install).start()
```

Then you can access related actions via command palette (Ctrl+Shift+P on windows/mac/linux, or Command+Shift+P on mac) after restarting IDA Pro.

Testing latest changes

To test the master branch, you can replace `tag='v...'` into `tag='master'` .

What file is created

`~(Your home directory)/idapkg`, and some lines in idapythonrc.py will be created.

```
idapkg/
  packages/
  python/
  config.json
```

packages/

When a package is installed, `packages/<name>` is created and further added to `IDAUSR` variable. This enables following folders to be loaded by IDA Pro.

```
<name>
  plugins/
  procs/
  loaders/
  til/
  sig/
  ids/
```

python/ - virtualenv

To manage PIP packages easily, this creates a virtualenv and creates `pip`, `easy_install` and other virtalenv-related files and activates the environment.

TL;DR If you run `pip install`, they are installed into `python/lib/*` (`Lib` on windows, all same.)

config.json

In fact, all paths above are configurable!

```
{
    "path": {
        "virtualenv": "...\\idapkg\\python",
        "packages": "...\\idapkg\\packages"
    },
    "repos": [
        "https://api.idapkg.com",
        "github:Jinmo/idapkg-repo/master"
    ]
}
```

And you can use your private repo for fetching packages. The api server will be uploaded soon!

Writing a package

See Writing a package (link) .

TODO

Currently finding way to reliably and generally update `IDAUSR` variable on all platforms . Currently only supporting Windows and Mac OS X.

---

ifred - by: jinmo123

Plugin description

> IDA command palette & more

readme for ifred

IDA command palette & more

How to build

Currently tested on Windows. Install Qt 5.6.3 and IDA SDK, and follow steps in azure-pipelines.yml.

You can download prebuilt plugins from azure pipelines.

Python API

You can make a custom palette in IDAPython.

```
from __palette__ import show_palette, Palette, Action
import random, string

myhandler = lambda item: sys.stdout.write('You selected: %s\n' % item.name)
random_str = lambda: "".join(random.choice(string.lowercase) for i in range(20))

entries = [Action(name=random_str(), # displayed text
    handler=myhandler, # callback
    id='action%d' % i # must be unique
    ) for i in range(20)]

show_palette(Palette('palette name here', 'placeholder here...', entries))
```

C++ API

Currently cleaning up C++ API. See `standalone/` folder.

```
#include <palette/api.h>
#define COUNT 100

QVector<Action> testItems() {
    QVector<Action> action_list;

    action_list.reserve(COUNT + 1);
    action_list.push_back(Action("std::runtime_error", "raise exception", ""));

    for (int i = 0; i < COUNT; i++) {
        auto id = QString::number(rand());
        action_list.push_back(Action(id, id, ""));
    }

    return action_list;
}

const QString TestPluginPath(const char* name) {
    // Don't worry! also packaged with bundle theme!
    // Just point a writable path
    return QString("./path_to_plugin_theme/") + name;
}

int main() {
    QApplication app(argc, argv);

    set_path_handler(TestPluginPath);

    show_palette("<test palette>", "Enter item name...", testItems(), [](const Action & acti
        if (action.id() == "std::runtime_error") {
            throw std::runtime_error("raised!");
        }
        qDebug() << action.id() << action.description() << action.shortcut();
        return false;
    });

    app.exec();
}
```

Changing theme

You can copy css, json files from `palette/res/theme/<name>/*` to `%APPDATA%/Hex-rays/IDA Pro/plugins/palette/theme/`, like the exist-

ing css, json files.

ayu white:

---

findrpc - by: Lucas Georges

Plugin description

Idapython script to carve binary for internal RPC structures

readme for findrpc

findrpc : Ida script to extract RPC interface from binaries

Usage

Just run the script `findrpc.py` .

NB : wait for the autoanalysis to finish before running the script.

The script rely on the same heuristic as James Forshaw's `FindRpcServerInterfaces` but is a bit more powerful since it can use IDA's xrefs system to uncover "non-obvious" relationships between RPC structures (e.g. in the case of a proxy definition).

Unlike `RpcView` or `NtApiDotNet` `findrpc.py` also return the RPC clients embbeded in the binary, which may or may not be of use when reversing.

Features

- View in a glance which RPC clients and servers are embedded in the binary :
- Locate dispatch tables for RPC servers:
- Quicky rename every RPC proc handlers found:
- (On Windows) Generate decompiled IDL from RPC declarations :

BB : if you want to regenerate the IDL, you need to have the `decompile` folder in the same directory as `findrpc.py` . `DecompileInterface.exe` is a custom loader for Forshaw's `NdrParser.ReadFromRpcServerInterface` which uses a json file exported from IDA instead of reading a remote process's memory.

Study Example #1 : SgrmBroker

Some RPC servers are not accessible via RpcView or NtObjectManager, although any clients can connect to it. One prime example is `SgrmBroker` which is a service running as PPL on Windows 10. Protected Processes Light (PPL) are famously protected against remote process memory reading, even if thoses processes are running as admin, thus why RpcView fail at localizing the interface.

`findrpc` can find it since it rely only on the binary :

Decompiled Interface :

```
// DllOffset: 0x2C9C0
// DllPath C:\Windows\System32\SgrmBroker.exe
// Complex Types:
/* Memory Size: 132 */
struct Struct_0 {
    /* Offset: 0 */ sbyte[128] Member0;
    /* Offset: 128 */ int Member1;
};


[uuid("7a20fcec-dec4-4c59-be57-212e8f65d3de"), version(1.0)]
interface intf_7a20fcec_dec4_4c59_be57_212e8f65d3de {

    HRESULT SgrmCreateSession(
        /* Stack Offset: 0 */ handle_t p0,
        /* Stack Offset: 8 */ [In] wchar_t[1]* p1,
        /* Stack Offset: 16 */ [In] struct Struct_0* p2,
        /* Stack Offset: 24 */ [Out] /* FC_BIND_CONTEXT */ handle_t* p3,
        /* Stack Offset: 32 */ [Out] UIntPtr* p4
    );

    HRESULT SgrmEndSession(
        /* Stack Offset: 0 */ handle_t p0,
        /* Stack Offset: 8 */ [In, Out] /* FC_BIND_CONTEXT */ handle_t* p1
    );

    HRESULT GetSessionReport(
        /* Stack Offset: 0 */ handle_t p0,
        /* Stack Offset: 8 */ [In] /* FC_BIND_CONTEXT */ handle_t* p1,
        /* Stack Offset: 16 */ [Out] /* C:(FC_TOP_LEVEL_CONFORMANCE)(24)(FC_DEREFERENCE)(FC_
        /* Stack Offset: 24 */ [Out] int* p3
    );

    HRESULT GetRuntimeReport(
```

```
        /* Stack Offset: 0 */ handle_t p0,
        /* Stack Offset: 8 */ [In] /* FC_BIND_CONTEXT */ handle_t* p1,
        /* Stack Offset: 16 */ [In] struct Struct_0* p2,
        /* Stack Offset: 24 */ [Out] /* C:(FC_TOP_LEVEL_CONFORMANCE)(32)(FC_DEREFERENCE)(FC_
        /* Stack Offset: 32 */ [Out] int* p4
    );

    HRESULT GetSessionCertificate(
        /* Stack Offset: 0 */ handle_t p0,
        /* Stack Offset: 8 */ [In] /* FC_BIND_CONTEXT */ handle_t* p1,
        /* Stack Offset: 16 */ [Out] /* C:(FC_TOP_LEVEL_CONFORMANCE)(24)(FC_DEREFERENCE)(FC_
        /* Stack Offset: 24 */ [Out] int* p3
    );
}
```

Study Example #2 : WarpJITSvc

Some Windows services are only running "on demand" and exit as soon as they are done processing the client calls. This is the case for `WarpJITSvc` : this is a Windows service which provide CPU-generated shaders when there is no graphic acceleration provided by system (e.g. on a server or in a VM).

`WarpJITSvc` is "triggered" when a client attempt to access the RPC interface `"5a0ce74d-f9cf-4dea-a4c1-2d5fe4c89d51"` . The rasterization RPC server is not present in its ServiceDLL `Windows.WARP.JITService.dll` but in one of its dependency, `d3d10warp.dll`

```
// DllOffset: 0x5D5B70
// DllPath C:\Windows\System32\d3d10warp.dll

[uuid("5a0ce74d-f9cf-4dea-a4c1-2d5fe4c89d51"), version(1.0)]
interface intf_5a0ce74d_f9cf_4dea_a4c1_2d5fe4c89d51 {

    HRESULT WARPJITOOPServerConnect(
        /* Stack Offset: 0 */ handle_t p0,
        /* Stack Offset: 8 */ [In] /* FC_SYSTEM_HANDLE Process(VmOperation, VmRead, VmWrite,
        /* Stack Offset: 16 */ [In] int p2,
        /* Stack Offset: 24 */ [In] /* C:(FC_TOP_LEVEL_CONFORMANCE)(16)(FC_ZERO)(FC_ULONG)(F
        /* Stack Offset: 32 */ [In] /* range: 0,2147483647 */ int p4,
        /* Stack Offset: 40 */ [Out] int* p5,
        /* Stack Offset: 48 */ [Out] /* C:(FC_TOP_LEVEL_CONFORMANCE)(32)(FC_ZERO)(FC_ULONG)(
    );

    HRESULT WARPJITOOPServerPassMessage(
        /* Stack Offset: 0 */ handle_t p0,
        /* Stack Offset: 8 */ [In] int p1,
        /* Stack Offset: 16 */ [In] /* C:(FC_TOP_LEVEL_CONFORMANCE)(8)(FC_ZERO)(FC_ULONG)(Ea
```

```
        /* Stack Offset: 24 */ [In] /* range: 0,2147483647 */ int p3,
        /* Stack Offset: 32 */ [Out] int* p4,
        /* Stack Offset: 40 */ [Out] /* C:(FC_TOP_LEVEL_CONFORMANCE)(24)(FC_ZERO)(FC_ULONG)(
    );

    HRESULT WARPJITOOPServerGetConnectionUUID(
        /* Stack Offset: 8 */ handle_t p0,
        /* Stack Offset: 16 */ [In] /* range: 0,2147483647 */ int p1,
        /* Stack Offset: 24 */ [Out] int* p2,
        /* Stack Offset: 32 */ [Out] /* C:(FC_TOP_LEVEL_CONFORMANCE)(16)(FC_ZERO)(FC_ULONG)(
    );

    HRESULT WARPJITOOPServerGetConnectionUUID2(
        /* Stack Offset: 8 */ handle_t p0,
        /* Stack Offset: 16 */ [In] /* range: 0,2147483647 */ int p1,
        /* Stack Offset: 24 */ [Out] int* p2,
        /* Stack Offset: 32 */ [Out] /* C:(FC_TOP_LEVEL_CONFORMANCE)(16)(FC_ZERO)(FC_ULONG)(
    );
}
```

Study Example #3 : notskrnl.exe

RPC code is fucking everywhere, even in kerneland. Apparentlty some parts of the `\\Device\\DeviceApi` device can deserialize NDR data using RPC interface's `bd84cd86-9825-4376-813d334c543f89b1` type and string formats :

This is coherent with the fact that the "Device Association" service register `bd84cd86-9825-4376-813d334c543f89b1` interface on the epr mapper :

References

- RpcView : https://github.com/silverf0x/RpcView/
- James Forshaw's NtApiDotNet : https://github.com/googleprojectzero/sandbox-attacksurface-analysis-tools/tree/master/NtApiDotNet

---

deREferencing - by: Daniel Garcia Gutierrez

Plugin description

> IDA Pro plugin that implements more user-friendly register and stack views

readme for deREferencing

deREferencing

deReferencing is an IDA Pro plugin that implements new registers and stack views. Adds dereferenced pointers, colors and other useful information, similar to some GDB plugins (e.g: PEDA, GEF, pwndbg, etc).

Supports following architectures: x86, x86-64, ARM, ARM64, MIPS32 and MIPS64

Requirements

- IDA-Pro >= 7.1

Install

Just drop the `dereferencing.py` file and the `dereferencing` folder into IDA's plugin directory.

To install just for the current user, copy the files into one of these directories:

OS	Plugin path
Linux/macOS	`~/.idapro/plugins`
Windows	`%AppData%\Hex-Rays\IDA Pro\plugins`

Usage

Both views can be opened from the menu `Debugger -> Debugger Windows` or by shortcuts:

- deREferencing - Registers (`Alt-Shift-D`)
- deREferencing - Stack (`Alt-Shift-E`)

You also can save the desktop layout using the `Windows -> Save desktop` option, so that the plugin starts automatically in other debugging sessions.

Configuration

Config options can be modified vía `deferencing/config.py` file.

Snapshots

Registers view

Stack view

Thanks

Special mention to my colleague @roman_soft for give me some ideas during the development of the plugin.

Bugs / Feedback / PRs

Any comment, issue or pull request will be highly appreciated :-)

Author

- Daniel García Gutiérrez - @danigargu

BRUTAL IDA - by: Tamir Bahar

Plugin description

> BRUTAL IDA restores your original workflow by blocking the undo and redo keyboard shortcuts.

readme for BRUTAL IDA

BRUTAL IDA

Block Redo & Undo To Achieve Legacy IDA

The Problem

As we all know, IDA has no undo. No undo - no surrender.

So naturally, IDA 7.3 adding undo functionality has ruined everyone's workflow.

In The Media

- IDA 7.3 Adds Undo
- Undo Causes Massive Outcry
- T-Shirt Line Destroyed

This will not stand.

The Solution

BRUTAL IDA restores your original workflow by blocking the `undo` and `redo` keyboard shortcuts.

It comes in 2 modes:

- : Block `undo` and `redo`
- : Crash IDA on `undo` and `redo`

Installation & Usage

It is recommended to install the plugin using the IDA Plugin Loader . Just point it to the `brutal_ida.py` script.

Once installed, the plugin will add a toolbar to IDA:

Clicking the IDA icon will toggle between and modes. The default is .

SmartJump - by: Adam Prescott (PwC)

Plugin description

> SmartJump is designed to improve the 'g' keyboard shortcut in IDA, especially when using IDA to debug binaries.

readme for SmartJump

IDA Pro plugin to enhance the JumpAsk 'g' command

Installation

Copy the contents of the plugin folder into your IDA_DIR/plugin folder
Edit `IDA_DIR\cfg\idagui.cfg` so that the line that has default text with:
`"JumpAsk" = 'g'`
Instead reads:
`"JumpAsk" = 0`
You can append the text:
`// 'g'`
To the line to give a full entry of:
`"JumpAsk" = 0 // 'g'`
If you do not want to remember what the default value was.

Usage

SmartJump is designed to improve the `g` keyboard shortcut in IDA, especially when using IDA to debug binaries. It allows a user to do basic mathematical operations `-`, `+`, `/`, `*` on values and labels in the JumpAsk window.
In addition, it allows a user to use the symbols `[` and `]` to dereference memory addresses and jump to the values contained at the address.

The supported type of values that can be used in the jumpask window are:
`here` and `here()` - these resolve to the current result of `idc.here()`
`main` and `sub_123456` - you can still jump by names/labels in the binary
`12ab34` and `0x12ab34` - all numbers are interpreted as hexadecimal numbers and can either be preceded by `0x` or not. If a global name also matches a hexadecimal number pattern then the global name will take precedence
`eax` , `ebx` , ... , `rax` , ... - 32 and 64 bit registers for x86 and x64 are supported. Using `eax` on an x64 binary will mask the lower 32 bits and return that value

All of these values can be used in combinations, a brief but inexhaustive list of examples is below:
`[eax]` - Grabs the current value in eax, goes to the memory location and attempts to read a 32 bit pointer if in IDA32 or a 64 bit pointer if in IDA64. If the resolution is a valid address then it jumps there
`[here] + rsp` - Grabs the value stored at the current address and adds it to the 64 bit stack pointer then jumps to the resulting stack location
`[[ebx]*4]+[edx]` - You are starting to get the picture, you can do any jumps that resolve to an address in IDA...

You can use the symbols `(` and `)` to explicitly group operations together rather than relying on the precedences assigned for the operators
E.g.
`0x1200 * ([ebx] + here)` - This will grab the contents of ebx, add the current address to it and then multiply it by 0x1200 and try to jump to the result

You do not have to match `[` and `]` symbols:
`[[[eax` - This will automatically have enough `]` symbols appended to the end of the query to match the opening `[` symbols
The final result would be a triple dereference of eax - `[[[eax]]]`
The matching brace completion only supports the `[` symbol currently and will not match `(` symbols. All auto added braces are put at the end of the input expression

---

Renamer - by: Sharon Brizinov

Plugin description

> For each function [Renamer] shows you what are the available strings and let you choose what is the appropriate name for the function.

readme for Renamer

Renamer

A plugin which helps in the process of naming functions based on their strings.

How does it work

The plugin walks you through all the functions in your binary. For each function it shows you what are the available strings in the function and let you choose what is the appropriate name for that function. Upon choosing name you'll jump to the next function.

The default hot-key to activate the plugin is `Ctrl-Shift-N` .

Installation

Copy `renamer.py` to IDA plugin's directory, usually at: `C:\Program Files\IDA\plugins`

Tested versions

- v7.3

Known Issues

AttributeError: 'module' object has no attribute 'MAXSTR'

Some users report they receive a traceback related to MAXSTR not being found. It'a a bug in some versions of IDAPython API, specifically in the function `SetControlValue` which seems have broken in some API migration.

> File "C:\Program Files\IDA Pro 7.3\python\ida_kernwin.py", line 6924, in SetControlValue
> tid, _ = self.ControlToFieldTypeIdAndSize(ctrl) File "C:\Program Files\IDA Pro 7.3\python\ida_kernwin.py", line 6951, in ControlToFieldTypeIdAndSize
> return (3, min(_ida_kernwin.MAXSTR, ctrl.size)) AttributeError: 'module' object has no attribute 'MAXSTR'

The solution is to manually edit `C:\Program Files\IDA Pro 7.x\python\ida_kernwin.py` and replace `_ida_kernwin.MAXSTR` with some big number (e.g. 65536).

qiling - by: ChenXu WU, ZiQiao KONG and Qiling.io Team

Plugin description

> …an advanced binary emulator with instrumentation plugin for IDA

readme for qiling

Qiling's usecase, blog and related work

Qiling is an advanced binary emulation framework, with the following features:

- Emulate multi-platforms: Windows, MacOS, Linux, Android, BSD, UEFI, DOS, MBR, Ethereum Virtual Machine
- Emulate multi-architectures: 8086, X86, X86_64, ARM, ARM64, MIPS, RISCV, PowerPC
- Support multiple file formats: PE, MachO, ELF, COM, MBR
- Support Windows Driver (.sys), Linux Kernel Module (.ko) & MacOS Kernel (.kext) via Demigod
- Emulates & sandbox code in an isolated environment
- Provides a fully configurable sandbox
- Provides in-depth memory, register, OS level and filesystem level API
- Fine-grain instrumentation: allows hooks at various levels (instruction/basic-block/memory-access/exception/syscall/IO/etc)
- Provides virtual machine level API such as save and restore current execution state
- Supports cross architecture and platform debugging capabilities
- Built-in debugger with reverse debugging capability
- Allows dynamic hotpatch on-the-fly running code, including the loaded library
- True framework in Python, making it easy to build customized security analysis tools on top

Qiling also made its way to various international conferences.

2021:

- Black Hat, USA
- Hack In The Box, Amsterdam
- Black Hat, Asia

2020:

- Black Hat, Europe
- Black Hat, USA
- Black Hat, USA (Demigod)
- Black Hat, Asia

- Hack In The Box, Lockdown 001
- Hack In The Box, Lockdown 002
- Hack In The Box, Cyberweek
- Nullcon

2019:

- Defcon, USA
- Hitcon
- Zeronights

Qiling is backed by Unicorn engine .

Visit our website https://www.qiling.io for more information.

License This project is released and distributed under free software license GPLv2 and later version.

Qiling vs other Emulators There are many open source emulators, but two projects closest to Qiling are Unicorn & Qemu usermode . This section explains the main differences of Qiling against them.

Qiling vs Unicorn engine Built on top of Unicorn, but Qiling & Unicorn are two different animals.

- Unicorn is just a CPU emulator, so it focuses on emulating CPU instructions, that can understand emulator memory. Beyond that, Unicorn is not aware of higher level concepts, such as dynamic libraries, system calls, I/O handling or executable formats like PE, MachO or ELF. As a result, Unicorn can only emulate raw machine instructions, without Operating System (OS) context
- Qiling is designed as a higher level framework, that leverages Unicorn to emulate CPU instructions, but can understand OS: it has executable format loaders (for PE, MachO & ELF at the moment), dynamic linkers (so we can load & relocate shared libraries), syscall & IO handlers. For this reason, Qiling can run executable binary without requiring its native OS

Qiling vs Qemu usermode Qemu usermode does similar thing to our emulator, that is to emulate whole executable binaries in cross-architecture way. However, Qiling offers some important differences against Qemu usermode.

- Qiling is a true analysis framework, that allows you to build your own dynamic analysis tools on top (in friendly Python language). Meanwhile, Qemu is just a tool, not a framework

- Qiling can perform dynamic instrumentation, and can even hotpatch code at runtime. Qemu does not do either
- Not only working cross-architecture, Qiling is also cross-platform, so for example you can run Linux ELF file on top of Windows. In contrast, Qemu usermode only run binary of the same OS, such as Linux ELF on Linux, due to the way it forwards syscall from emulated code to native OS
- Qiling supports more platforms, including Windows, MacOS, Linux & BSD. Qemu usermode can only handle Linux & BSD

Installation Please see setup guide file for how to install Qiling Framework.

Examples

- The example below shows how to use Qiling framework in the most striaghtforward way to emulate a Windows executable.

```
from qiling import Qiling

if __name__ == "__main__":
    # initialize Qiling instance, specifying the executable to emulate and the emulated syst
    # note that the current working directory is assumed to be Qiling home
    ql = Qiling([r'examples/rootfs/x86_windows/bin/x86_hello.exe'], r'examples/rootfs/x86_wi

    # start emulation
    ql.run()
```

- The following example shows how a Windows crackme may be patched dynamically to make it always display the ”Congratulation” dialog.

```
from qiling import Qiling

def force_call_dialog_func(ql: Qiling):
    # get DialogFunc address from current stack frame
    lpDialogFunc = ql.stack_read(-8)

    # setup stack memory for DialogFunc
    ql.stack_push(0)
    ql.stack_push(1001)     # IDS_APPNAME
    ql.stack_push(0x111)    # WM_COMMAND
    ql.stack_push(0)

    # push return address
    ql.stack_push(0x0401018)
```

```
    # resume emulation from DialogFunc address
    ql.arch.regs.eip = lpDialogFunc


if __name__ == "__main__":
    # initialize Qiling instance
    ql = Qiling([r'rootfs/x86_windows/bin/Easy_CrackMe.exe'], r'rootfs/x86_windows')

    # NOP out some code
    ql.patch(0x004010B5, b'\x90\x90')
    ql.patch(0x004010CD, b'\x90\x90')
    ql.patch(0x0040110B, b'\x90\x90')
    ql.patch(0x00401112, b'\x90\x90')

    # hook at an address with a callback
    ql.hook_address(force_call_dialog_func, 0x00401016)
    ql.run()
```

The below Youtube video shows how the above example works.

Emulating ARM router firmware on Ubuntu X64 machine

- Qiling Framework hot-patch and emulates ARM router's /usr/bin/httpd on a X86_64Bit Ubuntu

Qiling's IDAPro Plugin: Instrument and Decrypt Mirai's Secret

- This video demonstrate how Qiling's IDAPro plugin able to make IDApro run with Qiling instrumentation engine

GDBserver with IDAPro demo

- Solving a simple CTF challenge with Qiling Framework and IDAPro

Emulating MBR

- Qiling Framework emulates MBR

Qltool Qiling also provides a friendly tool named `qltool` to quickly emulate shellcode & executable binaries.

With qltool, easy execution can be performed:

With shellcode:

```
$ ./qltool code --os linux --arch arm --format hex -f examples/shellcodes/linarm32_tcp_reve
```

With binary file:

```
$ ./qltool run -f examples/rootfs/x8664_linux/bin/x8664_hello --rootfs  examples/rootfs/x866
```

With binary and GDB debugger enable:

```
$ ./qltool run -f examples/rootfs/x8664_linux/bin/x8664_hello --gdb 127.0.0.1:9999 --rootfs
```

With code coverage collection (UEFI only for now):

```
$ ./qltool run -f examples/rootfs/x8664_efi/bin/TcgPlatformSetupPolicy --rootfs examples/roo
```

With json output (Windows mainly):

```
$ ./qltool run -f examples/rootfs/x86_windows/bin/x86_hello.exe --rootfs  examples/rootfs/x8
```

Contact Get the latest info from our website https://www.qiling.io

Contact us at email info@qiling.io , or via Twitter @qiling_io or Weibo

---

Core developers, Key Contributors and etc Please refer to CREDITS.md

---

PETree - by: Tom Bonner (BlackBerry Research and Intelligence Team)

Plugin description

> PETree is a Python module for viewing Portable Executable (PE) files in a tree-view

readme for PETree

PE Tree

PE Tree is a Python module for viewing Portable Executable (PE) files in a tree-view using pefile and PyQt5 . It can also be used with IDA Pro , Ghidra , Volatility , Rekall and minidump to view and dump in-memory PE files, as well as perform import table reconstruction.

Table of contents

Features

- Standalone application with plugins for:
  - IDA Pro
  - Ghidra
  - Volatility
  - Rekall
  - Minidumps
  - Carving
- Supports Windows, Linux and Mac
- Parsing PE files and memory images from:
  - File-system
  - ZIP archives (including password protected)
  - Windows memory dumps (raw, EWF, vmem etc.)
  - Live Windows memory (using rekall)
  - Windows Minidump
  - IDA Pro database
  - Ghidra database
  - Binary file carving
- Rainbow PE map:
  - Provides a high-level overview of PE structures, size and file location
  - Allows for fast visual overview and comparison of PE samples
- Displays the following PE headers in a tree-view:
  - MZ header
  - DOS stub
  - Rich headers
  - NT/File/Optional headers
  - Data directories
  - Sections
  - Imports
  - Exports
  - Debug information
  - Load config
  - TLS
  - Resources
  - Version information
  - Certificates
  - Overlay
- Extract and save data from:
  - DOS stub

  - Sections
  - Resources
  - Certificates
  - Overlay
  - Export to CyberChef for further manipulation
- Perform VirusTotal searches of:
  - File hashes
  - PDB path
  - Timestamps
  - Section hash/name
  - Import hash/name
  - Export name
  - Resource hash
  - Certificate serial
- Dump loaded PE images from memory:
  - Fix up section pointers and sizes
  - Fix up PE headers:
    * Remove unnecessary data directory pointers
    * Recalculate PE checksum
    * Update entry-point
  - Reconstruct import address and directory tables (IAT/IDT) using several methods:
    a. Use existing IAT/IDT
    b. Rebuild IDT from existing IAT
    c. Rebuild IAT and IDT from disassembly (using IDA Pro, Ghidra or capstone)

Application

The PE Tree standalone application finds portable executables in files, folders and ZIP archives.

Requirements

- Python 3.5+

Features

- Scan files and folders for PE files
- Extract PE files from ZIP archives (including password protected with *infected*)
- Carve PE files from binary files
- Double-click VA/RVA to disassemble with capstone
- Hex-dump data

Installation

Using pip (recommended) Install directly from GitHub using a fresh virtual environment and pip:

Windows

```
> virtualenv env
> env\Scripts\activate
> pip install --upgrade pip
> pip install git+https://github.com/blackberry/pe_tree.git
```

Mac/Linux

```
$ python3 -m venv env
$ source ./env/bin/activate
$ pip install --upgrade pip
$ pip install git+https://github.com/blackberry/pe_tree.git
```

For developers Git clone the repository and setup for development:

Windows

```
> git clone https://github.com/blackberry/pe_tree.git
> cd pe_tree
> virtualenv env
> env\Scripts\activate
> pip install -e .
```

Mac/Linux

```
$ git clone https://github.com/blackberry/pe_tree.git
$ cd pe_tree
$ python3 -m venv env
$ source ./env/bin/activate
$ pip install -e .
```

Usage

Run PE Tree and scan for portable executables in files, folders and ZIP archives:

```
$ pe-tree -h
usage: pe-tree [-h] [filenames [filenames ...]]

PE-Tree

positional arguments:
  filenames   Path(s) to file/folder/zip
```

```
optional arguments:
  -h, --help  show this help message and exit
```

Run PE Tree and attempt to carve portable executable files from a binary file:

```
$ pe-tree-carve -h
usage: pe-tree-carve [-h] filename

PE-Tree (Carve)

positional arguments:
  filename    Path to file to carve

optional arguments:
  -h, --help  show this help message and exit
```

Dark-mode Dark-mode can be enabled by installing QDarkStyle :

```
$ pip install qdarkstyle
```

IDAPython

The PE Tree IDAPython plugin finds portable executables in IDA databases.

Requirements

- IDA Pro 7.0+ with Python 2.7
- IDA Pro 7.4+ with Python 2.7 or 3.5+

Features

- Easy navigation of PE file structures
- Double-click on a memory address in PE Tree to view in *IDA-view* or *hex-view*
- Search an IDB for in-memory PE images and;
  - Reconstruct imports (IAT + IDT)
  - Dump reconstructed PE files
  - Automatically comment PE file structures in IDB
  - Automatically label IAT offsets in IDB

Installation

To install and run as an IDAPython plugin you can either use setuptools or install manually.

Using setuptools

1. Download pe_tree and install for the global Python interpreter used by IDA:

   ```
   $ git clone https://github.com/blackberry/pe_tree.git
   $ cd pe_tree
   $ python setup.py develop --ida
   ```

2. Copy pe_tree_ida.py to your IDA plugins folder

Install manually

1. Download pe_tree and install requirements for the global Python interpreter used by IDA:

   ```
   $ git clone https://github.com/blackberry/pe_tree.git
   $ cd pe_tree
   $ pip install -r requirements.txt
   ```

2. Copy pe_tree_ida.py and the contents of `./pe_tree/` to your IDA plugins folder

For developers To forgo installing as a plugin, and simply run as a script under IDA, first install the pe_tree package requirements for the global Python installation:

```
$ pip install -r requirements.txt
```

Then run pe_tree_ida.py under IDA:

```
File -> Script file... -> pe_tree_ida.py -> Open
```

IDA plugins folder

OS	Plugins folder
Windows	`%ProgramFiles%\IDA Pro 7.X\plugins`
Linux	`/opt/ida-7.X/plugins`
Mac	`~/.idapro/plugins`

Usage

1. Launch IDA Pro and disassemble a PE file (always select Manual Load and Load Resources for best results!)

2. Load the PE Tree plugin:

   ```
   Edit -> Plugins -> PE Tree
   ```

Example

Dumping in-memory PE files Below are the basic steps to dump a packed PE file (for example MPRESS or UPX) and reconstruct imports (assuming the image base/entry-point is fairly standard):

1. Launch IDA Pro and disassemble an MPRESS or UPX packed PE file (select Manual Load and Load Resources)
2. Select a debugger (Windows or Bochs) and run until OEP (usually 0x00401000, but not always!)
3. At this point you could take a memory snapshot (saving all segments) and save the IDB for later
4. Ensure IDA has found all code:

   ```
   Options -> General -> Analysis -> Reanalyze program
   ```

5. Open the PE Tree IDAPython plugin, right-click in the right-hand pane and select:

   ```
   Add PE -> Search IDB
   ```

   This will scan the IDB for MZ/PE headers and display any modules it finds.
6. Right-click on HEADER-0x00400000 (or appropriate module name) and select Dump...
7. Specify the AddressOfEntryPoint (typically 0x1000, but again, not always!)
8. Ensure Rebuild IDT/IAT is selected
9. Dump!

A new executable will be created using the unpacked section data obtained from memory/IDB, whilst a new section named `.pe_tree` and containing the rebuilt IAT, hint name table and IDT will be appended to the PE file (much like an .idata section). If the entry-point memory segment has been marked writable during execution (via VirtualProtect for example) then the entry-point section's characteristics will also be marked writable. Finally, the BASERELOC, BOUND_IMPORT and SECURITY data directories are marked null, and the OPTIONAL_HEADER checksum is recalculated (if specified).

Using the above approach it is possible to dump many in-memory PE files that have either been unpacked, injected, reflectively loaded or hollowed etc.

Rekall

The PE Tree Rekall plugin finds portable executables in Windows memory dumps.

Requirements

- Python 3+

Features

- Operates against a Windows memory dump or a live system
- View, dump and reconstruct PE modules from;
  - Active processes and DLLs
  - Loaded kernel-mode drivers

Installation

1. Install Rekall from GitHub .
2. Install PE Tree standalone application (see Installation) under the same virtual environment.

Usage

Run Rekall and view active processes, DLLs and drivers on a live system:

```
$ rekall --live Memory
[1] Live (Memory) 00:00:00> run -i pe_tree_rekall.py
```

Alternatively, run Rekall/PE Tree against an existing memory dump:

```
$rekall -f memory.vmem
[1] memory.vmem 00:00:00> run -i pe_tree_rekall.py
```

Volatility

> *The PE Tree Volatility plugin finds portable executables in Windows memory dumps.*

Requirements

- Python 3.5+

Features

- Operates against a Windows memory dump
- View, dump and reconstruct PE modules from;
  - Active processes and DLLs
  - Loaded kernel-mode drivers

Installation

1. Install Volatility3 from GitHub .
2. Install PE Tree standalone application (see Installation) under the same virtual environment.

Usage

```
$ pe-tree-vol -h
usage: pe-tree-vol [-h] filename

PE-Tree (Volatility)

positional arguments:
  filename    Path to memory dump

optional arguments:
  -h, --help  show this help message and exit
```

Ghidra

The PE Tree Ghidra plugin finds portable executables in Ghidra databases.

Requirements

- Python 3.5+
- Ghidra Bridge

Features

- Easy navigation of PE file structures
- Double-click on a memory address in PE Tree to view in Ghidra disassembly/hex-view
- Reconstruct imports (IAT + IDT)
- Dump reconstructed PE files

Installation

1. Install PE Tree (see Installation)
2. Install Ghidra Bridge

Usage

1. Start the Ghidra Bridge server
2. Run the PE Tree Ghidra plugin

```
$ pe-tree-ghidra -h
usage: pe-tree-ghidra [-h] [--server SERVER] [--port PORT]

PE-Tree (Ghidra)

optional arguments:
  -h, --help       show this help message and exit
```

```
  --server SERVER  Ghidra bridge server IP (default: 127.0.0.1)
  --port PORT      Ghidra bridge server port (default: 4768)
```

Minidump

The PE Tree Minidump plugin finds portable executables in Windows Minidumps.

Requirements

- Python 3.6+
- minidump

Features

- View, dump and reconstruct PE modules from a Windows Minidump (.dmp) file

Installation

1. Install PE Tree (see Installation)
2. Install minidump (`pip install minidump`)

Usage

```
$ pe-tree-minidump -h
usage: pe-tree-minidump [-h] filename

PE-Tree (Minidump)

positional arguments:
  filename    Path to .dmp file

optional arguments:
  -h, --help  show this help message and exit
```

Configuration

Overview

The configuration is stored in an INI file and defaults to the following values:

```
[config]
debug = False
fonts = Consolas,Monospace,Courier
passwords = ,infected
virustotal_url = https://www.virustotal.com/gui/search
cyberchef_url = https://gchq.github.io/CyberChef
```

Options

Section	Option	Type	Description
config	debug	boolean	Print pefile.dump() to output
config	fonts	string	Comma-separated list of font names for UI
config	passwords	string	Comma-separated list of ZIP file passwords
config	virustotal_url	string	VirusTotal search URL
config	cyberchef_url	string	CyberChef URL

Location

Type	OS	Path
Application	Windows	`%TEMP%\pe_tree.ini`
Application	Linux/Mac	`/tmp/pe_tree.ini`
IDAPython	Windows	`%APPDATA%\HexRays\IDA Pro\pe_tree.ini`
IDAPython	Linux/Mac	`~/.idapro/pe_tree.ini`
Rekall	Windows	`%TEMP%\pe_tree_rekall.ini`
Rekall	Linux/Mac	`/tmp/pe_tree_rekall.ini`
Volatility	Windows	`%TEMP%\pe_tree_volatility.ini`
Volatility	Linux/Mac	`/tmp/pe_tree_volatility.ini`
Ghidra	Windows	`%TEMP%\pe_tree_ghidra.ini`
Ghidra	Linux/Mac	`/tmp/pe_tree_ghidra.ini`
Minidump	Windows	`%TEMP%\pe_tree_minidump.ini`
Minidump	Linux/Mac	`/tmp/pe_tree_minidump.ini`
Carve	Windows	`%TEMP%\pe_tree_carve.ini`
Carve	Linux/Mac	`/tmp/pe_tree_carve.ini`

3rd party data sharing

The following information will be shared with 3rd party web-applications (depending on configuration) under the following conditions:

VirusTotal If the VirusTotal URL is specified in the configuration then metadata such as file hashes, timestamps, etc will be sent to VirusTotal for processing when the user clicks on highlighted links or selects "VirusTotal search" from the right-click context menu.

CyberChef If the CyberChef URL is present in the configuration then any file data will be base64 encoded and sent to CyberChef for processing when the user selects "CyberChef" from the right-click context menu.

Troubleshooting

AttributeError: module 'pip' has no attribute 'main' or

PyQt5 fails to install under Linux Try to upgrade pip to version 20.0+:

```
$ pip install --upgrade pip
```

ModuleNotFoundError: No module named 'PyQt5.sip' Try uninstalling and reinstalling PyQt5 as follows:

```
pip uninstall PyQt5
pip uninstall PyQt5-sip
pip install PyQt5 PyQt5-sip
```

Missing imports after dumping Ensure IDA has found and disassembled all code:

```
Options -> General -> Analysis -> Reanalyze program
```

After this is completed try to dump/rebuild imports again.

Contributing

Please feel free to contribute! Issues and pull requests are most welcome.

Developer documentation

To build documentation from source using Sphinx :

```
$ pip install sphinx
$ sphinx-apidoc -o ./doc/source/ .
$ sphinx-build -b html ./doc/source ./doc/build -E
```

To view the documentation open `./doc/build/index.html` in a web-browser.

Authors

Tom Bonner - tombonner - @thomas_bonner

License

PE Tree is distributed under the Apache License. See LICENSE for more information.

PacXplorer - by: Ouri Lipner (Cellebrite Security Research)

Plugin description

> PacXplorer analyzes ARM64 PAC instructions to provide a new type of cross-reference: from call site to virtual function, and vice versa.

readme for PacXplorer

PacXplorer

PacXplorer is an IDA plugin that adds XREFs between virtual functions and their call sites.
This is accomplished by leveraging PAC codes in ARM64e binaries.

Installation

1. install ida-nentode somewhere IDA can import it
2. clone the repository and symlink `~/.idapro/plugins/pacxplorer.py` to `pacxplorer.py` in the cloned repo

Usage

Preliminary Analysis

1. open an IDB and make sure autoanalysis has finished
   1.1. *KernelCache only:* make sure to run ida_kernelcache(Cellebrite's fork) . This defines the `` `vtable for' `` symbols
2. from the menu select `Edit -> Plugins -> PacXplorer -> Analyse IDB`
   2.2. if asked, point PacXplorer to the original input binary file that created the IDB
   - this will only happen if the original PAC codes are not present in the IDB and the input binary can't be located automatically
3. done!

Continuous Use

TL;DR

`-x` or `Right Click -> Jump to PAC XREFs` at suitable locations will open a selection window

From call site to virtual function

1. place the cursor on a `MOVK` instruction near the call site to a virtual function (marked with a comment).
2. press the hotkey or activate the menu entry

3. a list of possible virtual methods called will open
4. if the same virtual function is called from several vtables, the `class` column will show `<multiple classes>` instead of a class name
   - `Right click -> PAC: toggle unique function names` toggles this grouping

From virtual function to call site

1. place the cursors either on a vtable entry or at the start of a virtual function
2. press the hotkey or activate the menu entry
3. a list of possible call sites will open

Principals of Operation

PAC codes sign pointers with a *key* and a *context* .
LLVM ABI specifies that the *context* of vtable entries is a mix between the *entry's address* and a *hash* of the function prototype.

Consider the following vtable at address `0x00000001 abcdef00` :

offset	method	hash
0	`foo()`	0x1234
8	`bar()`	0x9876

The formula for calculating the context is:

```
addr_part = (addr_of_vtable + offset_of_method) & 0x0000ffffffffffff
hash_part = (hash & 0xffff) << 48
context   =  addr_part | hash_part
```

Hence when calling `bar()` , the context will be: `0x98760001 abcdef08` .

Out of these factors, the *offset* of the method and the *hash* are known at compile time, but the actual *address of the vtable* is only known at runtime, through the `this` ptr (heck, this is the whole purpose of vtables to begin with).

Therefore, a typical (simplified) code snippet might look as follows:

```
LDR     X8, [X0]            ; load vtable address
LDRA    X9, [X8,#0x18]!     ; X8 = vtbl + offset
MOVK    X8, #0x68DA,LSL#48  ; set the hash
BLRAA   X9, X8              ; virtual call
```

PacXplorer looks for similar `MOVK` instructions in all of the defined functions and analyses the code leading up to them, noting the *offset* in the table and the *hash value* , and constructs *PAC tuple mappings* of `{(offset, hash): address of call}`

On the other hand, PacXplorer iterates over all of the defined vtables, which it finds using symbol names.
In the binary, each vtable contains *tagged pointers* , which will have been untagged by IDA. The pointer tags embed the hash values that are used for PAC. PacXplorer looks for the original pointer tags, which will have either been preserved in IDA's *patched bytes* window, or by opening the actual original binary. Using that, it creates the same *PAC tuple* for each virtual call, and construct a mapping of `{(offset, hash): entry in vtable}` .

At runtime, a simple matching of these two mappings is performed.

Q&A

Q: Why are there several virtual methods in the XREFs window?
A: This is the *inherent ambiguity* which is an intrinsic limitation of this method. For every class inheritance tree, when calling a virtual method that's present in the parent class and overloaded in some of the children, there is no knowing at compile time which overloaded version will actually get called.
Obviously only what's known at compile time can be statically analysed.

Q: Why use a special window and not add the XREFs to the regular XREF list?
A: Due to the inherent ambiguity of which virtual function is called, I decided not to add (potentially many) bogus XREFs to the regular list, but keep them separated.

Q: Could there be false positives?
A: Yes. Besides the inherent ambiguity, there could also be cases where two functions in unrelated vtables generate the same (offset, hash) tuple.
When trying out using the hash value alone (disregarding the offset in the vtable), I've encountered many such false positives. Using the combination of (offset, hash) I'm yet to observe any such false positives.

Q: Why is the XREF from the `MOVK` instruction and not the `BLRAA` call?
A: I've encountered instances of several virtual calls using the same BLRAA. Think of a function that selects a command handler with a switch-case, and all the cases jump to the same exit node that performs the call.

Q: Great stuff, I want to work with you!
A: Uhh, that's not really a question but thanks! click here (Remote talent welcome)

Limitations

1. Works only on ARM64e binaries that conform to the ABI
2. Vtable symbols need to be present in the binary (`` `vtable for' ``). In the case of the Kernel, ida_kernelcache needs to have been run. Note

that the official version doesn't support recent Kernels, but Cellebrite's fork is up to date.
3. If the tagged pointers haven't been preserved in the IDB, the original binary is needed for the analysis stage (but not afterwards)
4. Hexrays support WIP

Meta

Authored by Ouri Lipner of Cellebrite Security Research Labs.
Currently maintained by Omer Porzecanski of Cellebrite Security Research Labs.
Developed and tested for IDA 7.5 - 7.7 on OS X, iOS 12.x - 15.4 beta

---

Lucid - by: Markus Gaasedelen

Plugin description

> Lucid is a developer-oriented IDA Pro plugin for exploring the Hex-Rays microcode.

readme for Lucid

Lucid - An Interactive Hex-Rays Microcode Explorer

Overview

Lucid is a developer-oriented IDA Pro plugin for exploring the Hex-Rays microcode. It was designed to provide a seamless, interactive experience for studying microcode transformations in the decompiler pipeline.

This plugin is labeled only as a prototype & code resource for the community. Please note that it is a development aid, not a general purpose reverse engineering tool.

Special thanks to genmc / @pat0is et al. for the inspiration.

Releases

- v0.1 -- Initial release

Installation

Lucid is a cross-platform (Windows, macOS, Linux) Python 2/3 plugin. It takes zero third party dependencies, making the code both portable and easy to install.

1. From your disassembler's python console, run the following command to find its plugin directory:
   - IDA Pro : `os.path.join(idaapi.get_user_idadir(), "plugins")`
2. Copy the contents of this repository's `/plugins/` folder to the listed directory.
3. Restart your disassembler.

This plugin is only supported for IDA 7.5 and newer.

Usage

Lucid will automatically load for any architecture with a Hex-Rays decompiler present. Simply right click anywhere in a Pseudocode window and select `View microcode` to open the Lucid Microcode Explorer.

By default, the Microcode Explorer will synchronize with the active Hex-Rays Pseudocode window.

Lucid Layers

Lucid makes it effortless to trace microinstructions through the entire decompiler pipeline. Simply select a microinstruction, and *scroll* (or click... if you must) through the microcode maturity layer list.

Watch as the explorer stays focused on your selected instruction, while the surrounding microcode landscape melts away. It's basically magic.

Sub-instruction Granularity

Cursor tracing can operate at a sub-operand / sub-instruction level. Placing your cursor on different parts of the same microinstruction can trace subcomponents back to their respective origins.

If the instructions at the traced address get optimized away, Lucid will attempt to keep your cursor in the same approximate context. It will change the cursor color from green to red to indicate the loss of precision.

Sub-instruction Trees

As the Hex-Rays microcode increases in maturity, the decompilation pipeline begins to nest microcode as sub-instructions and sub-operands that form tree-

based structures.

You can view these individual trees by right clicking an instruction and selecting `View subtree` .

Known Bugs

As this is the initial release, there will probably a number of small quirks and bugs. Here are a few known issues at the time of release:

- While sync'd with hexrays, cursor mapping can get wonky if focused on microcode that gets optimized away
- When opening the Sub-instruction Graph, window/tab focus can change unexpectedly
- Microcode Explorer does not dock to the top-level far right compartment on Linux?
- Switching between multiple Pseudocode windows in different functions might cause problems
- Double clicking an instruction address comment can crash IDA if there is no suitable view to jump to
- Plugin has not been tested robustly on Mac / Linux
- ...?

If you encounter any crashes or bad behavior, please file an issue.

Future Work

Time and motivation permitting, future work may include:

- Clean up the code.....
- Interactive sub-instruction graph generalization (to pattern_t / rules)
- Microcode optimizer development workflow?
- Microcode optimization manager?
- Ctree explorer (and similar graph generalization stuff...)
- Microcode hint text?
- Improve layer translations
- Improve performance
- Migrate off IDA codeview?
- ...?

I welcome external contributions, issues, and feature requests. Please make any pull requests to the `develop` branch of this repository if you would like them to be considered for a future release.

Authors

- Markus Gaasedelen (@gaasedelen)

---

idapm - by: Taichi Kotake

Plugin description

idapm is IDA Plugin Manager.

readme for idapm

idapm

`idapm` is IDA Plugin Manager. It works perfectly on macOS, it probably works on Windows and Linux. This is because I only have the macOS version of IDA Pro.

Motivation

Managing the IDA Plugin is inconvenient. There is no official package manager and you have to copy files to the plugin directory manually. So I developed a plugin manager inspired by `go get` that allows you to install plugins from GitHub repositories without API server, and also allows you to import plugins from different directories on your PC with a single command.

Installation

```
$ pip install git+ssh://git@github.com/tkmru/idapm.git
```

Usage

init

`init` command creates `~/idapm.json`. It contains information about the plugins you have installed.

```
$ idapm init
~/idapm.json was created successfully!
```

If `~/idapm.json` already exists, you can install the plugins listed in `~/idapm.json`.

```
$ idapm init
~/idapm.json already exists...
Do you want to install a plugin written in ~/idapm.json? [Y/n]: y
----------------------
Try: git clone https://github.com/L4ys/LazyIDA.git
```

```
Cloning into '/Applications/IDA Pro 7.5/ida.app/Contents/MacOS/plugins/idapm/L4ys/LazyIDA'..
Symbolic link(/Applications/IDA Pro 7.5/ida.app/Contents/MacOS/plugins/LazyIDA.py) has been
Installed successfully!
```

install

`install` command install plugin from GitHub repository or local.

from GitHub You can install from the specified GitHub repository.

```
$ idapm install L4ys/LazyIDA
----------------------
Try: git clone https://github.com/L4ys/LazyIDA.git
Cloning into '/Applications/IDA Pro 7.5/ida.app/Contents/MacOS/plugins/idapm/L4ys/LazyIDA'..
Symbolic link(/Applications/IDA Pro 7.5/ida.app/Contents/MacOS/plugins/LazyIDA.py) has been
Installed successfully!
```

The installed plug-ins are marked in the config.

```
$ cat /Users/tkmru/idapm.json
{
  "plugins": [
    "keystone-engine/keypatch",
    "L4ys/LazyIDA"
  ]
}
```

from local You can copy a Python script from the specified directory.

```
$ idapm install --local ./
Copy to /Applications/IDA Pro 7.5/ida.app/Contents/MacOS/plugins/test.py from ./test.py
Installed successfully!
```

list

`list` command displays a list of installed plugins.

```
$ idapm list
List of scripts in IDA plugin directory
LazyIDA.py

List of plugins in config
L4ys/LazyIDA
```

check

```
$ idapm check
IDA plugin dir:    /Applications/IDA Pro 7.5/ida.app/Contents/MacOS/plugins
idapm config path: /Users/tkmru/idapm.json
```

License

GPLv3 - GNU General Public License, version 3

---

ida_medigate - by: Uriel Malin

Plugin description

…C++ plugin for IDA Pro

readme for ida_medigate

ida_medigate C++ plugin for IDA Pro

[TOC]

Motivation And Background

Reverse engineering of compiled C++ code is not fun. Static Reverse engineering of compiled C++ code is frustrating. The main reason which makes it so hard is virtual functions. In contrast to compiled C code, there is no clear code flow. Too many times one can spend much time trying to understand what is the next virtual function is called, rather than just see the function like in compiled C code.

When one investigates a virtual function, the amount of time he or she needs to effort in order to find its xref, is not sense.

After too many C++ RE projects, I gave up and decided I need a flexible (Python) and stable (which I can easily maintain) tool for this type of research. Most of this plugin was written in January 2018, and recently I decided to clean the dust and add support of the new IDA (7.2) classes support.

This plugin isn't intended to work always "out of the box", but to be another tool for the reverser.

About

The plugin consists of two parts:

1. Implementation of C++ classes and polymorphism over IDA
2. A RTTI parser which rebuilds the classes hierarchy

This first part is not dependent on the second part, so it possible to use the plugin to reverse engineering a binary that doesn't contain RTTI, by defining those classes manually based on the plugin's API.

What makes the plugin unique is the fact it uses the same environment the researcher is already familiar with, and doesn't add any new menu or object, and based on the known IDA building blocks (structure, union, type for structure's members, etc) - This enable the plugin to support C++ abstracting for every architecture IDA supports .

Note: The RTTI parser parses x86/x64 g++ RTTI, but its structure enables to add support for more architectures and compilers easily.

Requirements

- IDA 7.5 SP + Hex-Rays Decompiler + Python 3
  - This version we partially support disassembly with no decompiler
- Linux - There is no anything that really depends on Linux, but the plugin was tested on IDA Linux version.
- ida-referee : We use this useful plugin to save xrefs for struct's members uses in the decompiler. The original plugin doesn't support Python3 so we port it (under the directory `plugins/`)

Installation:

Copy `medigate_cpp_plugin` to the `plugins` directory and add the source code path to your `idapythonrc.py` file

Copy `plugins/ida-referee/referee.py` to the same directory.

Features:

Assuming the binary original source code is the following (`examples/a.cpp`):

```
using namespace std;

class A {
    public:
    int x_a;
    virtual int f_a()=0;
};
```

```
class B : public A{
    public:
    int x_b;
    int f_a(){x_a = 0;}
    virtual int f_b(){this->f_a();}
};

class Z {
    public:
    virtual int f_z1(){cout << "f_z1";}
    virtual int f_z2(){cout << "f_z2";}
};

class C: public B, public Z{
    public:
    int f_a(){x_a = 5;}
    int x_c;
    int f_c(){x_c = 0;}
    virtual int f_z1(){cout << "f_z3";}
};


int main()
{
    C *c = new C();
    c->f_a();
    c->f_b();
    c->f_z1();
    c->f_z2();

    return 0;
}
```

The binary is stripped but contains RTTI.

RTTI Classes Hierarchy Rebuilding

When we just load the binary, the `main` function (`sub_84D` in the 32 bit version) looks like:

Initiate the g++ RTTI parser and run it, using:

```
from ida_medigate.rtti_parser import GccRTTIParser

GccRTTIParser.init_parser()

GccRTTIParser.build_all()
```

Now refresh struct C (see Remarks section), cast `v0` to be `C *`, decompile again:

Manual Classes Hierarchy Rebuilding

For cases that there are no RTTI, our infrastructure still enables to manually define c++ class. For the same example (examples/a32_stripped) you can create manually struct B, then select it's virtual table and type

```
from ida_medigate import cpp_utils

cpp_utils.make_vtable("B")
```

`make_vtable` can also get `vtable_ea` and `vtable_ea_stop` instead of the selected area.

Then create struct C, and apply the inheritance:

```
cpp_utils.add_baseclass("C", "B")
```

Now you can rebuild class C vtable by selecting it and typing:

```
cpp_utils.make_vtable("C")
```

Add structure Z, rebuild its vtable too, and now is the cool part:

```
cpp_utils.add_baseclass("C", "Z", 0x0c, to_update=True)
```

which apply C inheritance of Z at offset 0x0c and refresh the struct too (see remarks).

The last thing remained is too update the second vtable of C, the one that implements the interface of Z. Mark this vtable and type:

```
cpp_utils.make_vtable("C", offset_in_class=0x0c)
```

ida_medigate knows that this vtable is the vtable of class Z and the result will be:

The final result is the same like in the RTTI case:

Synchronization between functions and vtable members

When new a vtable struct is created (by the RTTI parser of manually by the user) each function which hasn't changed yet is renamed, decompiled, and set its first argument to `this`.

Double click on a structure's member which corresponds to such a function will navigate to the function, so the user can read the C++ code flow in a convenient way!

Every name or type changing of a function or its corresponding function pointer member in vtables is hooked and synchronized among all of them. This means

for example, the user could change the vtable member type through the decompiler window, and this new type (prototype) will be applied on the target function too.

Convenient reading of polymorphic code

In line 15 at the previous image, there is a call to B::sub_9A8 (B::f_b in the source code). This function argument is `B *` :

But, this function also might be called by a `C` instance (up-casting). we want to see the virtual function it's instance would call. Assume there are many potential derived classes so casting `this` to `C *` not always possible. For that reason, we implement a union for each baseclass that has sons that have a different virtual table. One can choose to show a different derived virtual table of `B` 's derivatives by click alt+y (the shortcut for choosing different union member):

so ultimately we can "cast" only specific calls to different virtual function:

Virtual Functions xref

The holy-grail of frustrated C++ reverse engineers. We maintain xrefs from virtual functions to the vtable struct's members which represents them!

Combining this with `ida-referee` enables us to track all the xrefs of virtual functions calls!

A limitation: we can only track virtual calls which already been decompiled. Fortunately, the auto-analysis knows to populate an argument type between functions, so with the iterative process of casting more arguments->decompiling all the relevant functions -> reading the code again and casting more arguments (...) this ability becomes really powerful!

Remarks

- The way we mark structure members as subclasses in IDAPython isn't synchronized right away to the IDB. The hack we do is to edit the structure so a synchronization will be triggered. You also may use

  ```
  utils.refresh_struct(struct_ptr)
  ```

  which adds a dummy field at the end of the struct and then undefined it.

---

idahunt - by: Aaron Adams, Cedric Halbronn (NCC Group)

Plugin description

> idahunt is a framework to analyze binaries with IDA Pro and hunt for things in IDA Pro. It is a command-line tool to analyze all executable files recursively from a given folder.

readme for idahunt

Overview

idahunt is a framework to analyze binaries with IDA Pro and hunt for things in IDA Pro. It is command line tool to analyse all executable files recursively from a given folder. It executes IDA in the background so you don't have to open manually each file. It supports executing external IDA Python scripts.

Requirements

- Python3 only (except IDA Python scripts which can be Python2/Python3 depending on your IDA setup)
- IDA Pro
- Windows, Linux, OS X

Features

- Specify how many instances of IDA you want to run simultaneously
- Automate creation of IDBs for multiple executables
- Execute IDA Python scripts across multiple executables
- Open multiple existing IDBs
- Support any binary format (raw assembly/PE/ELF/MACH-O/etc.) supported by IDA

- (Optional) Include IDA Python helpers. You can use these to easily build your own IDA Python scripts or you can use any other IDA Python library like sark or bip to name a few

Useful examples include (non-exhaustive list):

- Analyse Microsoft Patch Tuesday updates
- Analyse malware of the same family
- Analyse multiple versions of the same software
- Analyse a bunch of binaries (UEFI, HP iLO, Cisco IOS router, Cisco ASA firewall, etc.)

Scripting

IDA Python scripts capabilities are unlimited. You can import any existing IDA Python script or build your own. Some examples:

- Rename functions based on debugging strings
- Decrypt strings (e.g. malware)
- Hunt for the same symbol across multiple versions (using heuristics)
- Hunt for ROP gadgets
- Port reversed function names / symbols from one version to another using tools like diaphora
- Etc.

Usage

- `idahunt.py` : main tool to analyse executable files
- `filters/` : contains basic filters to decide which fiels in an input dir to analyze with IDA
  - `filters/default.py` : default basic filter not filtering anything and used by default
  - `filters/ciscoasa.py` : useful for analyzing Cisco ASA Firewall images
  - `filters/hpilo.py` : useful for analyzing HP iLO images
  - `filters/names.py` : basic filter based on name, name length or extension
- `script_template.py` : contains a `hello world` IDA Python script

```
C:\idahunt> C:\Python37-x64\python.exe .\idahunt.py -h
usage: idahunt.py [-h] [--inputdir INPUTDIR] [--analyse] [--open]
                  [--ida-args IDA_ARGS] [--scripts SCRIPTS [SCRIPTS ...]]
                  [--filter FILTER] [--cleanup] [--temp-cleanup] [--verbose]
                  [--max-ida MAX_IDA] [--list-only] [--version IDA_VERSION]
```

```
optional arguments:
  -h, --help            show this help message and exit
  --inputdir INPUTDIR   Input folder to search for files
  --analyse, --analyze  analyse all files i.e. create .idb for all of them
  --open                open all files into IDA (debug only)
  --ida-args IDA_ARGS   Additional arguments to pass to IDA (e.g.
                        -p<processor> -i<entry_point> -b<load_addr>)
  --scripts SCRIPTS [SCRIPTS ...]
                        List of IDA Python scripts to execute in this order
  --filter FILTER       External python script with optional arguments
                        defining a filter for the names of the files to
                        analyse. See filters/names.py for example
  --cleanup             Cleanup i.e. remove .asm files that we don't need
  --temp-cleanup        Cleanup temporary database files i.e. remove .id0,
                        .id1, .id2, .nam, .dmp files if IDA Pro crashed and
                        did not delete them
  --verbose             be more verbose to debug script
  --max-ida MAX_IDA     Maximum number of instances of IDA to run at a time
                        (default: 10)
  --list-only           List only what files would be handled without
                        executing IDA
  --version IDA_VERSION
                        Override IDA version (e.g. "7.5"). This is used to
                        find the path of IDA on Windows.
```

Simulate without executing

You can use `--list-only` with any command line to just list what the tool would do without actually doing it.

```
C:\idahunt>idahunt.py --inputdir C:\re --analyse --filter "filters\names.py -a 32 -v" --list
[idahunt] Simulating only...
[idahunt] ANALYSING FILES
[idahunt] Analysing C:\re\cves\cve-2014-4076.dll
[idahunt] Analysing C:\re\cves\cve-2014-4076.exe
[idahunt] Analysing C:\re\DownloadExecute.exe
[idahunt] Analysing C:\re\ReverseShell.exe
```

Initial analysis

Here we start an initial analysis. It finishes after a few seconds:

```
C:\idahunt>idahunt.py --inputdir C:\re --analyse --filter "filters\names.py -a 32 -v"
[idahunt] ANALYSING FILES
[idahunt] Analysing C:\re\cves\cve-2014-4076.dll
[idahunt] Analysing C:\re\cves\cve-2014-4076.exe
[idahunt] Analysing C:\re\DownloadExecute.exe
```

```
[idahunt] Analysing C:\re\ReverseShell.exe
[idahunt] Waiting on remaining 4 IDA instances
```

Here we cleanup temporary `.asm` files created by the initial analysis:

```
C:\idahunt>idahunt.py --inputdir C:\re --cleanup
[idahunt] Deleting C:\re\cves\cve-2014-4076.asm
[idahunt] Deleting C:\re\DownloadExecute.asm
[idahunt] Deleting C:\re\ReverseShell.asm
```

We can see the generated `.idb` as well as some `.log` files that contain the IDA Pro output window.

```
C:\idahunt>tree /f C:\re
Folder PATH listing
Volume serial number is XXXX-XXXX
C:\RE
    DownloadExecute.exe
    DownloadExecute.idb
    DownloadExecute.log
    ReverseShell.exe
    ReverseShell.idb
    ReverseShell.log

  cves
        cve-2014-4076.dll
        cve-2014-4076.exe
        cve-2014-4076.idb
        cve-2014-4076.log
```

Execute IDA Python script

Here we execute a basic IDA Python script that prints `[script_template] I execute in IDA, yay!` in the IDA Pro output window.

```
C:\idahunt>idahunt.py --inputdir C:\re --filter "filters\names.py -a 32 -v" --scripts C:\ida
[idahunt] EXECUTE SCRIPTS
[idahunt] Executing script C:\idahunt\script_template.py for C:\re\cves\cve-2014-4076.dll
[idahunt] Executing script C:\idahunt\script_template.py for C:\re\cves\cve-2014-4076.exe
[idahunt] Executing script C:\idahunt\script_template.py for C:\re\DownloadExecute.exe
[idahunt] Executing script C:\idahunt\script_template.py for C:\re\ReverseShell.exe
[idahunt] Waiting on remaining 4 IDA instances
```

Since it is saved in the `.log` file, we can check it successfully executed:

```
Autoanalysis subsystem has been initialized.
Database for file 'ReverseShell.exe' has been loaded.
Compiling file 'C:\Program Files (x86)\IDA 6.95\idc\ida.idc'...
Executing function 'main'...
```

```
[script_template] I execute in IDA, yay!
```

Filters

We can filter that idahunt only analyses files with a given pattern in the name (`-n Download` below):

```
C:\idahunt>idahunt.py --inputdir C:\re --filter "filters\names.py -a 32 -v -n Download" --sc
[idahunt] Simulating only...
[idahunt] EXECUTE SCRIPTS
[names] Skipping non-matching name Download in cve-2014-4076.dll
[names] Skipping non-matching name Download in cve-2014-4076.exe
[idahunt] Executing script C:\idahunt\script_template.py for C:\re\DownloadExecute.exe
[names] Skipping non-matching name Download in ReverseShell.exe
```

We can also filter that idahunt only analyses files with a given extension (`-e dll` below):

```
C:\idahunt>idahunt.py --inputdir C:\re --filter "filters\names.py -a 32 -v -e dll" --scripts
[idahunt] Simulating only...
[idahunt] EXECUTE SCRIPTS
[idahunt] Executing script C:\idahunt\script_template.py for C:\re\cves\cve-2014-4076.dll
[names] Skipping non-matching extension .dll in cve-2014-4076.exe
[names] Skipping non-matching extension .dll in DownloadExecute.exe
[names] Skipping non-matching extension .dll in ReverseShell.exe
```

Architecture detection

The architecture is required to know in advance due to IDA Pro architecture and the fact that it contains 2 different executables `idaq.exe` and `idaq64.exe` to analyse binaries of the two architectures 32-bit and 64-bit. This is especially true if you want to use the HexRays decompiler.

idahunt will automatically detect i386, ia64 and amd64 architectures in Windows PE files. If you need to automatically detect other architectures, you can create an issue or add it to idahunt and do a PR.

If you forget to provide the architecture of the files you want to analyse, the basic `filters\names.py` will return an error:

```
C:\idahunt>idahunt.py --inputdir C:\re --filter "filters\names.py -v -e dll" --scripts C:\id
[idahunt] Simulating only...
[idahunt] EXECUTE SCRIPTS
[names] Unknown architecture: None. You need to specify it with -a
[names] Skipping non-matching extension .dll in cve-2014-4076.exe
[names] Skipping non-matching extension .dll in DownloadExecute.exe
[names] Skipping non-matching extension .dll in ReverseShell.exe
```

Known projects using idahunt

- asadbg

---

grap - by: Aurélien Thierry (QuoSec), Jonathan Thieuleux, Léonard Benedetti

Plugin description

> grap is a tool to match binaries at the assembly and control flow level (for instance: a loop on a basic block containing a xor)

readme for grap

grap: define and match graph patterns within binaries

https://github.com/QuoSecGmbH/grap

grap takes patterns and binary files, uses a Casptone-based disassembler to obtain the control flow graphs from the binaries, then matches the patterns against them.

Patterns are user-defined graphs with instruction conditions ("opcode is xor and arg1 is eax") and repetition conditions (3 identical instructions, basic blocks...).

grap is available as a standalone tool with a disassembler and python bindings, and as an IDA plugin which takes advantage of the disassembly done by IDA and the reverser.

Support:

- Files: disassembly of PE, ELF and raw binary, further files should work within IDA
- Architecture: x86 and x86_64

Match quick pattern:

Match full pattern:

Match on multiple files:

Create patterns interactively from IDA:

Installation

This document describes how to build and install grap on a Linux distribution.

You may also read:

- WINDOWS.md : installing grap on Windows
- IDA.md : installation and usage instruction of the IDA plugin

Requirements

Besides compilers (build-essential), the following dependencies must be installed:

- cmake
- bison
- flex
- libboost-regex-dev
- libboost-system-dev
- libboost-filesystem-dev
- libseccomp-dev
- python3-dev
- python3-pefile
- python3-pyelftools
- python3-capstone
- swig (version 3 or newer is mandatory)

Thus on Ubuntu / Debian, this should work :

```
sudo apt-get install build-essential cmake bison flex libboost-regex-dev libboost-system-dev
```

Please note that those were tested for the latest Ubuntu LTS (18.04.3). Packages may differ depending on your distribution.

Build and install

The following commands will build and install the project:

- `mkdir build; cd build/` as we advise you to build the project in a dedicated directory
- `cmake ../src/; make` will build with cmake and make
- `sudo make install` will install grap into /usr/local/bin/

SWIG might fail to find python3 if your default version is python2, this can be overcome by switching to python3 as default. For instance on Ubuntu:

```
sudo update-alternatives --install /usr/bin/python python /usr/bin/python3 10
```

Usage

The tool can be launched by using the following command:

```
$ grap [options] pattern test_paths
```

Below are a few examples of supported options:

- `grap -h` : describes supported options

One can let grap infer a pattern from a string. Only few options are supported but this is useful for prototyping:

- `grap "opcode is xor and arg1 contains '['" (test.exe)` : look for a xor with a memory write
- `grap -v "sub->xor->sub" (test.exe)` : -v will output the path of the inferred pattern

Choose how the binaries are disassembled:

- `grap -od (pattern.grapp) samples/*` : disassemble files in folder samples/ with no attempt at matching
- `grap -f (pattern.grapp) (test.exe)` : force re-disassembling the binary, then matches it against pattern.grapp
- `grap --raw (pattern.grapp) (test.bin)` : disassembling raw file (use --raw-64 for 64 bits binaries)

Control the verbosity of the output:

- `grap -q -sa (pattern.grapp) (samples/*.grapcfg)` : match disassembled files, show matching and non matching files, one per line
- `grap -m (pattern.grapp) (test.grapcfg)` : show all matched nodes

Choose where the disassembled file(s) (.grapcfg) are written; match multiple files against multiple patterns:

- `grap patterns/basic_block_loop.grapp -o ls.grapcfg /bin/ls` : disassemble ls into ls.grapp and looks for basic block loops
- `grap (pattern1.grapp) -p (pattern2.grapp) (test.exe)` : match against multiple pattern files
- `grap -r -q patterns/ /bin/ -o /tmp/` : disassemble all files from /bin/ into /tmp/ and matches them against all .grapp patterns from patterns/ (recursive option -r applies to /bin/, not to patterns/)

Pattern examples

The following pattern detects a decryption loop consisting of a xor followed by sub found in a Backspace sample:

```
digraph decryption_md5_4ee00c46da143ba70f7e6270960823be {
A [cond=true, repeat=3]
B [cond="opcode is xor and arg2 is 0x11"]
C [cond="opcode is sub and arg2 is 0x25"]
D [cond=true, repeat=3]
E [cond="opcode beginswith j and nchildren == 2"]

A -> B
B -> C
C -> D
D -> E
E -> A [childnumber=2]
}
```

Note that pattern files can contain multiple pattern graphs.

You may find additional pattern examples in two directories:

- patterns/ contains a few patterns that can be useful on any binary such as a pattern to detect short loops or to detect a loop on basic blocks,
- examples/ contains patterns used against the Backspace malware (see examples/backspace_samples.md to obtain the binary samples).

Tutorials & further examples

On malware samples:

- Navigating malware samples with grap (CLI, IDA): https://yaps8.github.io/blog/grap_qakbot_navigation
- Automating function parsing and decryption (python bindings): https://yaps8.github.io/blog/grap_qakbot_strings

Python bindings usage:

- Python file demonstrating how to use bindings to analyze Backspace samples: examples/analyze_backspace.py
- Examples of IDApython scripting are integrated within the IDA plugin, you can see them here: https://yaps8.github.io/grap/html/scripting_css.html

Documentation

You will find more documentation in the doc/ folder:

- doc/COMPILE_OPTIONS.md
- doc/DEBUG.md
- doc/TESTS.md
- doc/syntax_highlighting.md

The syntax of pattern and test graphs is detailed in the file grap_graphs.pdf within the release section.

License

grap is licensed under the MIT license. The full license text can be found in LICENSE .

FingerMatch - by: Jan Prochazka

Plugin description

> FingerMatch is an IDA plugin for collecting functions, data, types and comments from analyzed binaries and fuzzy matching them in another binaries.

readme for FingerMatch

FingerMatch

IDA plugin for collecting functions, data, types and comments from analysed binaries and fuzzy matching them in another binaries.

> author: Jan Prochazka
> licence: none, public domain
> home: https://github.com/jendabenda/fingermatch

Usage

- fingerprint libraries and then match them in binaries you want to anlayze, you can focus only on unseen and interesting parts
- resume analysis when new version of previously analyzed binary is out, no need to reverse engineer everything from scratch
- anything what fits

Features

- fuzzy function matching
- data, types, comments matching
- can correctly match small functions
- easy to use

Installation

- works with IDA 7.4+, Python 3
- copy `fingermatch.py` into IDA `plugins` folder

UI

- menu `View -> Collect fingerprints` - collects fingerprints and save them into fingerprint database
- menu `View -> Match fingerprints` - loads fingerprints from filename and match them against analysed binary

Python API

- available from IDA console
- `fingermatch_collect(fingerdb_path)` - collects fingerprints and save them into fingerprint database
- `fingermatch_match(fingerdb_path)` - loads fingerprints from fingerprint database and match them against analysed binary

Libraries workflow

- compile library with debugging symbols (`\Z7` or `\Zi` switch with msvc)
- autoanalyze binary with IDA and pdb symbols
- collect fingerprints with FingerMatch
- match fingerprints wherever you want
- you can download example databases (`zlib` , `libpng` , `libtiff` , `openssl` , `cryptopp`) from here

Resumption workflow

- open binary, analyze it
- collect fingerprints with FingerMatch
- when new binary version is out, open new version
- match saved fingerprints

Fingerprints

Function fingerprints are based on control flow traces allowing to match the same function with shuffled basic block starts, different register allocation or instruction scheduling. Fingerprints of data, types and comments are also matched. In addition matching considers whole reference graph, so it has high chance to pinpoint correct names. Matching is tuned to have low false positive matches.

Detailed documentation is at the begining of the `fingermatch.py` file.

---

efiXplorer - by: efiXplorer team

Plugin description

> IDA plugin for UEFI firmware analysis and reverse engineering automation

readme for efiXplorer

efiXplorer - IDA plugin for UEFI firmware analysis and reverse engineering automation

Supported versions of Hex-Rays products: everytime we focus on last versions of IDA and Decompiler because we try to use most recent features from new SDK releases. That means we tested only on recent versions of Hex-Rays products and do not guarantee stable work on previous generations.

Why not IDApython: all code developed in C++ because it's a more stable and performant way to support a complex plugin and get full power of most recent SDK's features.

Supported Platforms: Windows, Linux and OSX.

efiXplorer core features

efiXloader description

Build instructions and Installation

Publications

- efiXplorer: Hunting for UEFI Firmware Vulnerabilities at Scale with Automated Static Analysis
- Static analysis-based recovery of service function calls in UEFI firmware

- How efiXplorer helping to solve challenges in reverse engineering of UEFI firmware

References

- https://github.com/LongSoft/UEFITool
- https://github.com/yeggor/UEFI_RETool
- https://github.com/gdbinit/EFISwissKnife
- https://github.com/snare/ida-efiutils
- https://github.com/al3xtjames/ghidra-firmware-utils
- https://github.com/DSecurity/efiSeek
- https://github.com/p-state/ida-efitools2
- https://github.com/zznop/bn-uefi-helper

Thank you for support

DynDataResolver - by: Holger Unterbrink (Cisco Talos)

Plugin description

> DDR is an IDA plugin that instruments binaries using the DynamoRIO framework.

readme for DynDataResolver

Dynamic Data Resolver (DDR)

Release date Version 1.0.2 beta: 17th of December 2020 15:00 CET

New features:

Start address - The instrumentation/analysis starts at this address

Break address - The instrumentation and execution of the sample process stops here

You can find a detailed description of the 1.0.2 features here at release time https://blog.talosintelligence.com/2020/12/talos-tools-of-trade.html

---

Release date Version 1.0.1 beta: 20th of October 2020 17:00 CET

You can find an overview of the 1.0.1 features here https://blog.talosintelligence.com/2020/10/ddr101beta.html

---

Version 1.0

Author: Holger Unterbrink (hunterbr@cisco.com)

Blog

Release date Version 1.0: 28th of May 2020 17:00 CET

The documentation and installation description of the project can be found here at release time:

https://blog.talosintelligence.com/tbd

A quick feature walkthrough video is here:

https://youtu.be/miSFddzvzL8

---

capa explorer - by: Mike Hunhoff, Moritz Raabe, William Ballenthin, Ana Maria Martinez Gomez

Plugin description

> capa explorer is an IDA Pro plugin written in Python that integrates the FLARE team's open-source framework, capa, with IDA. capa is a framework that uses a well-defined collection of rules to identify capabilities in a program.

readme for capa explorer

capa explorer is an IDAPython plugin that integrates the FLARE team's open-source framework, capa, with IDA Pro. capa is a framework that uses a well-defined collection of rules to identify capabilities in a program. You can run capa against a PE file, ELF file, or shellcode and it tells you what it thinks the program can do. For example, it might suggest that the program is a backdoor, can install services, or relies on HTTP to communicate. capa explorer runs capa analysis on your IDA Pro database (IDB) without needing access to the original binary file. Once a database has been analyzed, capa explorer helps you identify interesting areas of a program and build new capa rules using features extracted from your IDB.

We love using capa explorer during malware analysis because it teaches us what parts of a program suggest a behavior. As we click on rows, capa explorer jumps directly to important addresses in the IDB and highlights key features in the Disassembly view so they stand out visually. To illustrate, we use capa explorer to analyze Lab 14-02 from Practical Malware Analysis (PMA) available here . Our goal is to understand the program's functionality.

After loading Lab 14-02 into IDA and analyzing the database with capa explorer, we see that capa detected a rule match for `self delete via COMSPEC environment variable` :

We can use capa explorer to navigate our Disassembly view directly to the suspect function and get an assembly-level breakdown of why capa matched `self delete via COMSPEC environment variable` .

Using the `Rule Information` and `Details` columns capa explorer shows us that the suspect function matched `self delete via COMSPEC environment variable` because it contains capa rule matches for `create process` , `get COMSPEC environment variable` , and `query environment variable` , references to the strings `COMSPEC` , `> nul` , and `/c del` , and calls to the Windows API functions `GetEnvironmentVariableA` and `ShellExecuteEx` .

capa explorer also helps you build and test new capa rules. To start, select the `Rule Generator` tab, navigate to a function in your Disassembly view, and click `Analyze` . capa explorer will extract features from the function and display them in the `Features` pane. You can add features listed in this pane to the `Editor` pane by either double-clicking a feature or using multi-select + right-click to add multiple features at once. The `Preview` and `Editor` panes help edit your rule. Use the `Preview` pane to modify rule text directly and the `Editor` pane to construct and rearrange your hierarchy of statements and features. When you finish a rule you can save it directly to a file by clicking `Save` .

For more information on the FLARE team's open-source framework, capa, check out the overview in our first blog or jump straight to the code on GitHub .

Getting Started

Check out capa explorer's GitHub README for information on installing and using the plugin in your environment.

---

bip - by: Bruno Pujos, Synacktiv

Plugin description

> Bip is a project which aims to simplify the usage of Python for interacting with IDA. Its primary goal is to facilitate the use of Python in the interactive console of IDA and the writing of plugins

readme for bip

Bip

Bip is a project which aims to simplify the usage of python for interacting with IDA. Its main goals are to facilitate the usage of python in the interactive console of IDA and the writing of plugins. In a more general way the goal is to automate recurrent tasks done through the python API. Bip is also developed to provide a more object oriented, a "python-like" API and a *real* documentation.

This code is not complete, and a lot of features are still missing. Development is prioritized on what people ask for and what the developers use, so do not hesitate to make PR, Feature Request and Issues (including for the documentation).

The documentation is available in the RST format (and can be compiled using sphinx) in the `docs/` directory, it is also available online .

- Current IDA version: IDA 7.5SP1 and Python 2.7 or 3.8
- Last Bip Version: 1.0

Installation

This installation has been tested only on Windows and Linux: `python install.py` .

It is possible to use an optional `--dest` argument to install in a particular folder:

```
usage: install.py [-h] [--dest DEST]

optional arguments:
  -h, --help   show this help message and exit
  --dest DEST  Destination folder where to install Bip
```

This installer does not install any plugins by default, but simply the core of Bip. By default the destination folder is the one used by IDA locally (`%APPDATA%\Hex-Rays\IDA Pro\` for Windows and `$HOME/.idapro` for Linux and MacOSX).

Overview

This overview has a goal to show how the most usual operations can be done, it is far from being complete. All functions and objects in Bip are documented using doc string so just use `help(BipClass)` and `help(obj.bipmethod)` to get the doc in your shell.

Base

The module `bip.base` contains most of the *basic* features for interfacing with IDA. In practice this is mainly the disassembler part of IDA, this includes: manipulation of instructions, functions, basic blocks, operands, data, xrefs, structures, types, ...

Instructions / Operands The classes `bip.base.BipInstr` and `bip.base.BipOperand` :

```
>>> from bip.base import *
>>> i = BipInstr() # BipInstr is the base class for representing an instruction
>>> i # by default the address on the screen is taken
BipInstr: 0x1800D324B (mov     rcx, r13)
>>> i2 = BipInstr(0x01800D3242) # pass the address in argument
>>> i2
BipInstr: 0x1800D3242 (mov     r8d, 8)
>>> i2.next # access next instruction, previous with i2.prev
BipInstr: 0x1800D3248 (mov     rdx, r14)
>>> l = [i3 for i3 in BipInstr.iter_all()] # l contains the list of all BipInstruction of th
>>> i.ea # access the address
6443315787
>>> i.mnem # mnemonic representation
mov
>>> i.ops # access to the operands
[<bip.base.operand.BipOperand object at 0x0000022B0291DA90>, <bip.base.operand.BipOperand ob
>>> i.ops[0].str # string representation of an operand
rcx
>>> i.bytes # bytes in the instruction
[73L, 139L, 205L]
>>> i.size # number of bytes of this instruction
3
>>> i.comment = "hello" # set a comment, rcomment for the repeatable comments
>>> i
```

```
BipInstr: 0x1800D324B (mov     rcx, r13; hello)
>>> i.comment # get a comment
hello
>>> i.func # access to the function
Func: RtlQueryProcessLockInformation (0x1800D2FF0)
>>> i.block # access to basic block
BipBlock: 0x1800D3242 (from Func: RtlQueryProcessLockInformation (0x1800D2FF0))
```

Function / Basic block The classes `bip.base.BipFunction` and `bip.base.BipBlock` :

```
>>> from bip.base import *
>>> f = BipFunction() # Get the function, screen address used if not provided
>>> f
Func: RtlQueryProcessLockInformation (0x1800D2FF0)
>>> f2 = BipFunction(0x0018010E975) # provide an address, not necessary the first one
>>> f2
Func: sub_18010E968 (0x18010E968)
>>> f == f2 # compare two functions
False
>>> f == BipFunction(0x001800D3021)
True
>>> hex(f.ea) # start address
0x1800d2ff0L
>>> hex(f.end) # end address
0x1800d3284L
>>> f = BipFunction.get_by_name("RtlQueryProcessLockInformation") # fetch the function from
>>> f.name # get and set the name
RtlQueryProcessLockInformation
>>> f.name = "test"
>>> f.name
test
>>> f.size # number of bytes in the function
660
>>> f.bytes # bytes of the function
[72L, ..., 255L]
>>> f.callees # list of functions called by this function
[<bip.base.func.BipFunction object at 0x0000022B0291DD30>, ..., <bip.base.func.BipFunction o
>>> f.callers # list of functions which call this function
[<bip.base.func.BipFunction object at 0x0000022B04544048>]
>>> f.instr # list of instructions in the function
[<bip.base.instr.BipInstr object at 0x0000022B0291DB00>, ..., <bip.base.instr.BipInstr objec
>>> f.comment = "welcome to bip" # comment of the function, rcomment for repeatable ones
>>> f.comment
welcome to bip
>>> f.does_return # does this function return ?
```

```
True
>>> BipFunction.iter_all() # allows to iter on all functions defined in the database
<generator object iter_all at 0x0000022B029231F8>
>>> f.nb_blocks # number of basic blocks
33
>>> f.blocks # list of blocks
[<bip.base.block.BipBlock object at 0x0000022B04544D68>, ..., <bip.base.block.BipBlock objec
>>> f.blocks[5] # access the basic block 5, could be done with BipBlock(addr)
BipBlock: 0x1800D306E (from Func: test (0x1800D2FF0))
>>> f.blocks[5].func # link back to the function
Func: test (0x1800D2FF0)
>>> f.blocks[5].instr # list of instructions in the block
[<bip.base.instr.BipInstr object at 0x0000022B04544710>, ..., <bip.base.instr.BipInstr objec
>>> f.blocks[5].pred # predecessor blocks, blocks where control flow lead to this one
[<bip.base.block.BipBlock object at 0x0000022B04544D68>]
>>> f.blocks[5].succ # successor blocks
[<bip.base.block.BipBlock object at 0x0000022B04544710>, <bip.base.block.BipBlock object at
>>> f.blocks[5].is_ret # is this block containing a return
False
```

Data The class `bip.base.BipData` :

```
>>> from bip.base import *
>>> d = BipData(0x000180110068) # .rdata:0000000180110068 bip_ex          dq offset unk_1801
>>> d
BipData at 0x180110068 = 0x180110DE0 (size=8)
>>> d.name # Name of the symbol if any
bip_ex
>>> d.is_word # is it a word
False
>>> d.is_qword # is it a qword
True
>>> hex(d.value) # value at that address, this take into account the basic type (byte, word,
0x180110de0L
>>> hex(d.ea) # address
0x180110068L
>>> d.comment = "example" # comment as before
>>> d.comment
example
>>> d.value = 0xAABBCCDD # change the value
>>> hex(d.value)
0xaabbccddL
>>> d.bytes # get the bytes, as before
[221L, 204L, 187L, 170L, 0L, 0L, 0L, 0L]
>>> hex(d.original_value) # get the original value before modification
0x180110de0L
```

```
>>> d.bytes = [0x11, 0x22, 0x33, 0x44, 0, 0, 0, 0] # patch the bytes
>>> hex(d.value) # get the value
0x44332211L
>>> BipData.iter_heads() # iter on "heads" of the IDB, heads are defined data in the IDB
<generator object iter_heads at 0x0000022B02923240>
>>> hex(BipData.get_dword(0x0180110078)) # staticmethod for reading value at an address
0x60004L
>>> BipData.set_byte(0x0180110078, 0xAA) # static method for modifying a value at an address
>>> hex(BipData.get_qword(0x0180110078))
0x600aaL
```

Element In Bip most basic objects inherit from the same classes: `BipBaseElt` which is the most basic one, `BipRefElt` which includes all the objects which can have xrefs (including structures (`BipStruct`) and structure members (`BStructMember`), see below), `BipElt` which represents all elements which have an address in the IDA DataBase (idb), including `BipData` and `BipInstr` (it is this class which implements the properties: `comment` , `name` , `bytes` , ...).

It is possible to use the functions `GetElt` and `GetEltByName` to get the right basic element from an address or a name representing a location in the binary.

```
>>> from bip.base import *
>>> GetElt() # get the element at current address, in this case return a BipData object
BipData at 0x180110068 = 0xAABBCCDD (size=8)
>>> GetElt(0x00180110078) # get the element at the address 0x00180110078
BipData at 0x180110078 = 0xAA (size=1)
>>> GetElt(0x1800D2FF0) # in this case it returns an BipInstr object because this is code
BipInstr: 0x1800D2FF0 (mov     rax, rsp)
>>> GetEltByName("bip_ex") # Get using a name and not an address
BipData at 0x180110068 = 0xAABBCCDD (size=8)
>>> isinstance(GetElt(0x1800D2FF0), BipInstr) # test if that element is an instruction ?
True
>>> GetElt(0x1800D2FF0).is_code # are we on code ? same for is_data; do not work for struct
True
>>> isinstance(GetElt(0x1800D2FF0), BipData) # or data ?
False
```

Some static functions are provided to search elements in the database:

```
>>> from bip.base import *
>>> GetElt()
BipInstr: 0x1800D3248 (mov     rdx, r14)
>>> BipElt.next_code() # find next code elt from current addres or addr passed as arg
BipInstr: 0x1800D324B (mov     rcx, r13)
```

```
>>> BipElt.next_code(down=False) # find prev code element
BipInstr: 0x1800D3242 (mov     r8d, 8)
>>> BipElt.next_data() # find next data elt from current address or addr passed as arg
BipData at 0x1800D3284 = 0xCC (size=1)
>>> BipElt.next_data(down=False) # find previous data element
BipData at 0x1800D2FE1 = 0xCC (size=1)
>>> hex(BipElt.next_data_addr(down=False)) # find address of the previous data element
0x1800d2fe1L
>>> BipElt.next_unknown() # same for unknown, which are not typed element of IDA and are cor
BipData at 0x180110000 = 0xE (size=1)
>>> BipElt.next_defined() # opposite of unknown: data or code
BipInstr: 0x1800D324B (mov     rcx, r13)
>>> BipElt.search_bytes("49 ? CD", 0x1800D3248) # search for byte sequence (ignore the curre
BipInstr: 0x1800D324B (mov     rcx, r13)
```

Xref All elements which inherit from `BipRefElt` (`BipInstr` , `BipData` , `BipStruct` , ...) and some other (in particular `BipFunction`) contain methods which allow to access xrefs. They are represented by the `BipXref` objects which have a `src` (origin of the xref) and a `dst` (destination of the xref).

```
>>> from bip.base import *
>>> i = BipInstr(0x01800D3063)
>>> i # example with instruction but works the same with BipData
BipInstr: 0x1800D3063 (cmp     r15, [rsp+98h+var_58])
>>> i.xTo # List of xref which point on this instruction
[<bip.base.xref.BipXref object at 0x0000022B04544438>, <bip.base.xref.BipXref object at 0x0(
>>> i.xTo[0].src # previous instruction
BipInstr: 0x1800D305E (mov     [rsp+98h+var_78], rsi)
>>> i.xTo[0].is_ordinaryflow # is this an ordinary flow between to instruction (not jmp or c
True
>>> i.xTo[1].src # jmp to instruction i at 0x1800D3063
BipInstr: 0x1800D3222 (jmp     loc_1800D3063)
>>> i.xTo[1].is_jmp # is this xref because of a jmp ?
True
>>> i.xEaTo # bypass the xref objects and get the address directly
[6443315294L, 6443315746L]
>>> i.xEltTo # bypass the xref objects and get the elements directly, will list BipData if a
[<bip.base.instr.BipInstr object at 0x0000022B045447F0>, <bip.base.instr.BipInstr object at
>>> i.xCodeTo # bypass the xref objects and get the instr directly, if a BipData was pointed
[<bip.base.instr.BipInstr object at 0x0000022B04544438>, <bip.base.instr.BipInstr object at
>>> i.xFrom # same but for coming from this instruction
[<bip.base.xref.BipXref object at 0x0000022B04544D68>]
>>> i.xFrom[0]
<bip.base.xref.BipXref object at 0x0000022B04544438>
>>> i.xFrom[0].dst # next instruction
```

```
BipInstr: 0x1800D3068 (jz      loc_1800D3227)
>>> i.xFrom[0].src # current instruction
BipInstr: 0x1800D3063 (cmp     r15, [rsp+98h+var_58])
>>> hex(i.xFrom[0].dst_ea) # address of the next instruction
0x1800D3068L
>>> i.xFrom[0].is_codepath # this is a normal code path (include jmp and call)
True
>>> i.xFrom[0].is_call # is this because of a call ?
False
>>> f = BipFunction()
>>> f
Func: RtlQueryProcessLockInformation (0x1800D2FF0)
>>> f.xTo # works also for functions, but only with To, not with the From
[<bip.base.xref.BipXref object at 0x000001D95529EB00>, <bip.base.xref.BipXref object at 0x0
>>> f.xEltTo # here we have 3 data references to this function
[<bip.base.instr.BipInstr object at 0x000001D95529EE48>, <bip.base.data.BipData object at 0x
>>> f.xCodeTo # but only one instruction
[<bip.base.instr.BipInstr object at 0x000001D95529EC88>]
```

Struct Manipulating struct (`BipStruct`) and members (`BStructMember`):

```
>>> from bip.base import *
>>> st = BipStruct.get("EXCEPTION_RECORD") # Structs are accessed by using get and their nam
>>> st # BipStruct object
Struct: EXCEPTION_RECORD (size=0x98)
>>> st.comment = "struct comment"
>>> st.comment
struct comment
>>> st.name
EXCEPTION_RECORD
>>> st.size
152
>>> st["ExceptionFlags"] # access to the BStructMember by their name
Member: EXCEPTION_RECORD.ExceptionFlags (offset=0x4, size=0x4)
>>> st[8] # or by their offset, this is *not* the entry number 8!!!
Member: EXCEPTION_RECORD.ExceptionRecord (offset=0x8, size=0x8)
>>> st[2] # offset does not need to be the first one
Member: EXCEPTION_RECORD.ExceptionCode (offset=0x0, size=0x4)
>>> st.members # list of members
[<bip.base.struct.BStructMember object at 0x000001D95529EEF0>, ..., <bip.base.struct.BStruct
>>> st[0].name
ExceptionCode
>>> st[0].fullname
EXCEPTION_RECORD.ExceptionCode
>>> st[0].size
```

```
4
>>> st[0].struct
Struct: EXCEPTION_RECORD (size=0x98)
>>> st[0].comment = "member comment"
>>> st[0].comment
member comment
>>> st[8].xEltTo # BStructMember et BipStruct have xrefs
[<bip.base.instr.BipInstr object at 0x000001D95536DD30>, <bip.base.instr.BipInstr object at
>>> st[8].xEltTo[0]
BipInstr: 0x1800A0720 (mov     [rsp+538h+ExceptionRecord.ExceptionRecord], r10)
```

Creating struct, adding members and nested structure:

```
>>> from bip.base import *
>>> st = BipStruct.create("NewStruct") # create a new structure
>>> st
Struct: NewStruct (size=0x0)
>>> st.add("NewField", 4) # add a new member named "NewField" of size 4
Member: NewStruct.NewField (offset=0x0, size=0x4)
>>> st.add("NewQword", 8)
Member: NewStruct.NewQword (offset=0x4, size=0x8)
>>> st
Struct: NewStruct (size=0xC)
>>> st.add("struct_nested", 1)
Member: NewStruct.struct_nested (offset=0xC, size=0x1)
>>> st["struct_nested"].type = BipType.from_c("EXCEPTION_RECORD") # changing the type of mem
>>> st["struct_nested"]
Member: NewStruct.struct_nested (offset=0xC, size=0x98)
>>> st["struct_nested"].is_nested # is this a nested structure ?
True
>>> st["struct_nested"].nested_struct # getting the nested structure
Struct: EXCEPTION_RECORD (size=0x98)
```

Types IDA uses extensively types in hexrays but also in the base API for defining types of data, variables and so on. In Bip the different types inherit from the same class `BipType`. This class offers some basic methods common to all types and subclasses (class starting by `BType`) can define more specific ones.

The types should be seen as a recursive structure: a `void *` is a `BTypePtr` containing a `BTypeVoid` structure. For a list of the different types implemented in Bip see the documentation .

```
>>> from bip.base import *
>>> pv = BipType.from_c("void *") # from_c is the easiest way to create a type
>>> pv
<bip.base.biptype.BTypePtr object at 0x000001D95536DDD8>
```

```
>>> pv.size # ptr on x64 is 8 bytes
8
>>> pv.str # C string representation
void *
>>> pv.is_named # this type is not named
False
>>> pv.pointed # type below the pointer (recursive)
<bip.base.biptype.BTypeVoid object at 0x000001D95536DF60>
>>> pv.children # list of type pointed
[<bip.base.biptype.BTypeVoid object at 0x000001D95536DEB8>]
>>> d = BipData(0x000180110068)
>>> d.type # access directly to the type at the address
<bip.base.biptype.BTypePtr object at 0x000001D95536D9E8>
>>> d.type.str
void *
>>> ps = BipType.from_c("EXCEPTION_RECORD *")
>>> ps.pointed # type for struct EXCEPTION_RECORD
<bip.base.biptype.BTypeStruct object at 0x000001D95536DD30>
>>> ps.pointed.is_named # this one is named
True
>>> ps.pointed.name
EXCEPTION_RECORD
>>> ps.set_at(d.ea) # set the type ps at address d.ea
>>> d.type.str # the type has indeed changed
EXCEPTION_RECORD *
>>> d.type = pv # rolling it back
>>> d.type.str
void *
>>> BipType.get_at(d.ea) # Possible to directly get the type with get_at(address)
<bip.base.biptype.BTypePtr object at 0x000001D95536DEB8>
```

Hexrays

The module `bip.hexrays` contains the features linked to the decompiler provided by IDA.

Functions / local variables Hexrays functions are represented by the `HxCFunc` objects and local variable by the `HxLvar` objects:

```
>>> HxCFunc.from_addr() # HxCFunc represents a decompiled function
<bip.hexrays.hx_cfunc.HxCFunc object at 0x00000278AE80C860>
>>> hf = BipFunction().hxfunc # accessible from a "normal function"
>>> hex(hf.ea) # address of the function
0x1800d2ff0L
>>> hf.args # list of the arguments as HxLvar objects
[<bip.hexrays.hx_lvar.HxLvar object at 0x00000278AFDAACF8>]
```

```
>>> hf.lvars # list of all local variables (including args)
[<bip.hexrays.hx_lvar.HxLvar object at 0x00000278AFDAAB70>, ..., <bip.hexrays.hx_lvar.HxLvar
>>> lv = hf.lvars[0] # getting the first one
>>> lv
LVAR(name=a1, size=8, type=<bip.base.biptype.BTypeInt object at 0x00000278AFDAAFD0>)
>>> lv.name # getting name of lvar
a1
>>> lv.is_arg # is this variable an argument ?
True
>>> lv.name = "thisisthefirstarg" # changing name of the lvar
>>> lv
>>> lv.type = BipType.from_c("void *") # changing the type
>>> lv.comment = "new comment" # adding a comment
>>> lv.size # getting the size
8
```

CNode / Visitors Hexrays allows to manipulate the AST it produces, this is a particularly useful feature as it allows to make static analysis at a way higher level. Bip defines `CNode` which represents a node of the AST, each type of node is represented by a subclass of `CNode` . All types of node have child nodes except `CNodeExprFinal` which are the leaf of the AST. Two *main* types of nodes exist `CNodeExpr` (expressions) and `CNodeStmt` (statements). Statements correspond to the C Statements: if, while, ... , expressions are everything else. Statements can have children statements or expressions while expressions can only have expression children.

A list of all the different types of nodes and more details on what they do and how to write a visitor is available in the documentation .

Directly accessing the nodes:

```
>>> hf = HxCFunc.from_addr() # get the HxCFunc
>>> rn = hf.root_node # accessing the root node of the function
>>> rn # root node is always a CNodeStmtBlock
CNodeStmtBlock(ea=0x1800D3006, stmt_children=[<bip.hexrays.cnode.CNodeStmtExpr object at 0x0
>>> hex(rn.ea) # address of the root node, after the function prolog
0x1800d3006L
>>> rn.has_parent # root node does not have parent
False
>>> rn.expr_children # this node does not have expression statements
[]
>>> ste = rn.stmt_children[0] # getting the first statement children
>>> ste # CNodeStmtExpr contain one child expression
CNodeStmtExpr(ea=0x1800D3006, value=CNodeExprAsg(ea=0x1800D3006, ops=[<bip.hexrays.cnode.CNo
>>> ste.parent # the parent is the root node
CNodeStmtBlock(ea=0x1800D3006, stmt_children=[<bip.hexrays.cnode.CNodeStmtExpr object at 0x0
>>> a = ste.value # getting the expression of the node
```

```
>>> a # Asg is an assignement
CNodeExprAsg(ea=0x1800D3006, ops=[<bip.hexrays.cnode.CNodeExprVar object at 0x00000278AFDAAI
>>> a.first_op # first operand of the assignement is a lvar, lvar are leaf
CNodeExprVar(ea=0xFFFFFFFFFFFFFFFF, value=1)
>>> a.first_op.lvar # get the lvar object
LVAR(name=v1, size=8, type=<bip.base.biptype.BTypeInt object at 0x00000278B16390B8>)
>>> a.ops # list all operands of the expression
[<bip.hexrays.cnode.CNodeExprVar object at 0x00000278AFDAADD8>, <bip.hexrays.cnode.CNodeExpr
>>> a.ops[1] # getting the second operand, also a lvar
CNodeExprVar(ea=0xFFFFFFFFFFFFFFFF, value=0)
>>> hex(a.ops[1].closest_ea) # lvar have no position in the ASM, but possible to take the or
0x1800d3006L
```

The previous code show how to get a value and manipulate nodes quickly. To do an analysis it is easier to use visitors on the complete function. `HxCFunc.visit_cnode` allows to visit all the nodes in a function with a callback, `HxCFunc.visit_cnode_filterlist` allows to visit only nodes of a certain type by passing a list of the node classes.

This script is an example to visit a function and get the format string passed to a `printk` function. It locates the call to `printk`, gets the address of the first argument, gets the string and adds a comment in both hexrays and the assembly:

```
from bip import *

"""
    Search for all call to printk, if possible gets the string and adds
    it in comments at the level of the call.
"""

def is_call_to_printk(cn):
    """
        Check if the node object represent a call to the function ``printk``.

        :param cn: A :class:`CNodeExprCall` object.
        :return: True if it is a call to printk, False otherwise
    """
    f = cn.caller_func
    return f is not None and f.name == "printk"

def visit_call_printk(cn):
    """
        Visitor for call node which will check if a node is a call to
        ``printk`` and add the string in comment if possible.

        :param cn: A :class:`CNodeExprCall` object.
```

```
    """
    # check if it calls to printk
    # For more perf. we would want to use xref to printk and checks of
    #   the address of the node
    if not is_call_to_printk(cn): # not a call to printk: ignore
        return
    if cn.number_args < 1: # not enough args
        print("Not enough args at 0x{:X}".format(cn.closest_ea))
        return
    cnr = cn.get_arg(0).ignore_cast # get the arg
    # if we have a ref (&global) we want the object under
    if isinstance(cnr, CNodeExprRef):
        cnr = cnr.ops[0].ignore_cast
    # if this is not a global object we ignore it
    if not isinstance(cnr, CNodeExprObj):
        print("Not an object at 0x{:X}".format(cn.closest_ea))
        return
    ea = cnr.value # get the address of the object
    s = None
    try:
        s = BipData.get_cstring(ea + 2) # get the string
    except Exception:
        pass
    if s is None or s == "":
        print("Invalid string at 0x{:X}".format(cn.closest_ea))
        return
    s = s.strip() # remove \n
    # add comment both in hexrays and in asm view
    cn.hxcfunc.add_cmt(cn.closest_ea, s)
    GetElt(cn.closest_ea).comment = s

# Final function which takes the address of a function and comments the call
#   to printk
def printk_handler(eafunc):
    hf = HxCFunc.from_addr(eafunc) # get the hexrays function
    hf.visit_cnode_filterlist(visit_call_printk, [CNodeExprCall]) # visit only the call node
```

While visitors are convenient (and "fast"), Bip also exposes methods to directly get the `CNode` objects as a list. The methods `HxCFunc.get_cnode_filter` and `HxCFunc.get_cnode_filter_type` allow to avoid having a visitor function and make it easier to manipulate the hexrays API. It is also worth noting that all visitors functions provided by `HxCFunc` objects are also available directly in `CNode` objects to visit only a sub-tree of the full AST.

Plugins

Plugins using Bip should all inherit from the class `BipPlugin`. Those plugins are different from the IDA plugins and are loaded and called by the `BipPluginManager`. Each plugin is identified by its class name and those should be unique. Bip can be used with standard plugin but most of the `bip.gui` implementation is linked to the use of `BipPlugin`. For more information about plugins and internals see the documentation .

Here is a simple plugin example:

```
from bip.gui import * # BipPlugin is defined in the bip.gui module

class ExPlugin(BipPlugin):
    # inherit from BipPlugin, all plugin should be instantiated only once
    # this should be done by the plugin manager, not "by hand"

    @classmethod
    def to_load(cls): # allow to test if the plugin apply, this MUST be a classmethod
        return True # always loading

    @shortcut("Ctrl-H") # add a shortcut as a decorator, will call the method below
    @shortcut("Ctrl-5") # add an other one
    @menu("Bip/MyPluginExample/", "ExPlugin Action!") # add a menu entry named "ExPlugin Act
    def action_with_shortcut(self):
        print(self) # this is the ExPlugin object
        print("In ExPlugin action !")# code here

bpm = get_plugin_manager() # get the BipPluginManager object
bpm.addld_plugin("ExPlugin", ExPlugin) # ask the BipPluginManager to load the plugin
# plugins in ``bipplugin`` folder will be loaded automatically and do not need those lines
```

The `menu` decorator will automatically create the `MyPluginExample` menu entry in the `Bip` top level menu entry (which is created by the `BipPluginManager`), creating an entry in the `Edit/Plugins/` directory may not work because of how the entry of this submenu are created by IDA.

A plugin can expose methods which another plugin wants to call or directly from the console. A plugin should not be directly instantiated, it is the `BipPluginManager` which is in charge of loading it. To get a `BipPlugin` object, it should be requested to the plugin manager:

```
from bip.gui import *
bpm = get_plugin_manager() # get the BipPluginManager object
bpm
# <bip.gui.pluginmanager.BipPluginManager object at 0x000001EFE42D68D0>
tp = bpm["TstPlugin"] # get the plugin object name TstPlugin
```

```
tp # can also be recuperated by passing directly the class
# <__plugins__tst_plg.TstPlugin object at 0x000001EFE42D69B0>
tp.hello() # calling a method of TstPlugin
# hello
```

For the previous example with `printk` we could write the following plugin:

```
class PrintkComs(BipPlugin):

    def printk_handler(self, eafunc):
        """
            Comment all call to printk in a function with the format string
            pass to the printk. Comments are added in both the hexrays and ASM
            view. Works only if the first argument is a global.

            :param eafunc: The addess of the function in which to add the
                comment.
        """
        try:
            hf = HxCFunc.from_addr(eafunc) # get hexray view of the func
        except Exception:
            print("Fail getting the decompile view for function at 0x{:X}".format(eafunc))
            return
        hf.visit_cnode_filterlist(visit_call_printk, [CNodeExprCall]) # visit only on the ca

    @shortcut("Ctrl-H")
    @menu("Bip/PrintkCom/", "Comment printk in current function")
    def printk_current(self):
        """
            Add comment for the current function.
        """
        self.printk_handler(Here())

    @menu("Bip/PrintkCom/", "Comment all printk")
    def printk_all(self):
        """
            Add comment for the all the functions in the IDB.
        """
        # get the function which call printk
        f = BipFunction.get_by_name("printk")
        if f is None:
            print("No function named printk")
            return
        for fu in f.callers:
            print("Renaming for {}".format(fu))
            self.printk_handler(fu.ea)
```

Similar projects

- sark : "an object-oriented scripting layer written on top of IDAPython".
- ida-minsc : "a plugin for IDA Pro that assists a user with scripting the IDAPython plugin that is bundled with the disassembler".
- FIDL : "FLARE IDA Decompiler Library"

Thanks

Some people to thanks:

- saph : for starting this project.
- hakril : for the inspiration for the project and his insights on designing it.

---

bf - by: Milan Boháček

Plugin description

> Bf_proc.py adds brainfuck language to the Hex-Rays decompiler.

readme for bf

VulFi - by: Martin Petran (Accenture)

Plugin description

> A query based function cross-reference finder for vulnerability research

readme for VulFi

VulFi v2.1

Introduction

The VulFi (Vulnerability Finder) tool is a plugin to IDA Pro which can be used to assist during bug hunting in binaries. Its main objective is to provide a single view with all cross-references to the most interesting functions (such as `strcpy`, `sprintf`, `system`, etc.). For cases where a Hexrays decompiler can be used, it will attempt to rule out calls to these functions which are not interesting from a vulnerability research perspective

(think something like `strcpy(dst,"Hello World!")`). Without the decompiler, the rules are much simpler (to not depend on architecture) and thus only rule out the most obvious cases.

Installation

Place the `vulfi.py`, `vulfi_prototypes.json` and `vulfi_rules.json` files in the IDA plugin folder (`cp vulfi* <IDA_PLUGIN_FOLDER>`).

Preparing the Database File

Before you run VulFi make sure that you have a good understanding of the binary that you work with. Try to identify all standard functions (`strcpy`, `memcpy`, etc.) and name them accordingly. The plugin is case insensitive and thus `MEMCPY`, `Memcpy` and `memcpy` are all valid names. However, note that the search for the function requires exact match. This means that `memcpy?` or `std_memcpy` (or any other variant) will not be detected as a standard function and therefore will not be considered when looking for potential vulnerabilities. If you are working with an unknown binary you need to set the compiler options first `Options > Compiler`. After that VulFi will do its best to filter all obvious false positives (such as call to `printf` with constant string as a first parameter). Please note that while the plugin is made without any ties to a specific architecture some processors do not have full support for specifying types and in such case VulFi will simply mark all cross-references to potentially dangerous standard functions to allow you to proceed with manual analysis. In these cases, you can benefit from the tracking features of the plugin.

Usage

Scanning

To initiate the scan, select `Search > VulFi` option from the top bar menu. This will either initiate a new scan, or it will read previous results stored inside the `idb / i64` file. The data are automatically saved whenever you save the database.

Once the scan is completed or once the previous results are loaded a table will be presented with a view containing following columns:

- IssueName - Used as a title for the suspected issue.
- FunctionName - Name of the function.
- FoundIn - The function that contains the potentially interesting reference.
- Address - The address of the detected call.
- Status - The review status, initial `Not Checked` is assigned to every new item. The other statuses are `False Positive`,

`Suspicious` and `Vulnerable`. Those can be set using a right-click menu on a given item and should reflect the results of the manual review of the given function call.

- Priority - An attempt to prioritize more interesting calls over the less interesting ones. Possible values are `High`, `Medium` and `Low` (also `Info` for cases where the scanner was not able to identify all parameters properly). The priorities are defined along with other rules in `vulfi_rules.json` file.
- Comment - A user defined comment for the given item.

In case that there are no data inside the `idb` / `i64` file or user decides to perform a new scan. The plugin will ask whether it should run the scan using the default included rules or whether it should use a custom rules file. Please note that running a new scan with already existing data does not overwrite the previously found items identified by the rule with the same name as the one with previously stored results. Therefore, running the scan again does not delete existing comments and status updates.

In the right-click context menu within the VulFi view, you can also remove the item from the results or remove all items. Please note that any comments or status updates will be lost after performing this operation.

Investigation

Whenever you would like to inspect the detected instance of a possible vulnerable function, just double-click anywhere in the desired row and IDA will take you to the memory location which was identified as potentially interesting. Using a right-click and option `Set Vulfi Comment` allows you to enter comment for the given instance (to justify the status for example).

Adding More Functions

The plugin also allows for creating custom rules. These rules could be defined in the IDA interface (ideal for single functions) or supplied as a custom rule file (ideal for rules that aim to cover multiple functions).

Within the Interface When you would like to trace a custom function, which was identified during the analysis, right-click anywhere within its body and select `Add <name> function to VulFi`. You could also highlight and right-click a function name within current disassembly/decompiler view to avoid switching into the function body.

Custom Set of Rules It is also possible to load a custom file with set of multiple rules. To create a custom rule file with the below structure you can use the included template file here .

```
[   // An array of rules
```

```
    {
        "name": "RULE NAME", // The name of the rule
        "function_names":[
            "function_name_to_look_for" // List of all function names that should be matched
        ],
        "wrappers":true,    // Look for wrappers of the above functions as well (note that t
        "mark_if":{
            "High":"True",  // If evaluates to True, mark with priority High (see Rules belo
            "Medium":"False", // If evaluates to True, mark with priority Medium (see Rules
            "Low": "False" // If evaluates to True, mark with priority Low (see Rules below)
        }
    }
]
```

An example rule that looks for all cross-references to function `malloc` and checks whether its paramter is not constant and whether the return value of the function is checked is shown below:

```
{
    "name": "Possible Null Pointer Dereference",
    "function_names":[
        "malloc",
        "_malloc",
        ".malloc"
    ],
    "wrappers":false,
    "mark_if":{
        "High":"not param[0].is_constant() and not function_call.return_value_checked()",
        "Medium":"False",
        "Low": "False"
    }
}
```

Rules

Available Variables

- `param[<index>]` : Used to access the parameter to a function call (index starts at `0`)
- `function_call` : Used to access the function call event
- `param_count` : Holds the count of parameters that were passed to a function

Available Functions

- Is parameter a constant: `param[<index>].is_constant()`
- Get numeric value of parameter: `param[<index>].number_value()`

- Get string value of parameter: `param[<index>].string_value()`
- Is parameter set to null after the call: `param[<index>].set_to_null_after_call()`
- Is return value of a function checked: `function_call.return_value_checked(<constant_to_che`
- Is the parameter also used as a parameter in one of the calls to a specified list of functions before/after: `param[<index>].used_in_call_<before|after>(["function1","f`
- Is the call to the selected function reachable from a specific other function: `function_call.reachable_from("<function_name>")`

Examples

- Mark all calls to a function where third parameter is > 5: `param[2].number_value() > 5`
- Mark all calls to a function where the second parameter contains "%s": `"%s" in param[1].string_value()`
- Mark all calls to a function where the second parameter is not constant: `not param[1].is_constant()`
- Mark all calls to a function where the return value is validated against the value that is equal to the number of parameters: `function_call.return_value_checked(param_count)`
- Mark all calls to a function where the return value is validated against any value: `function_call.return_value_checked()`
- Mark all calls to a function where none of the parameters starting from the third are constants: `all(not p.is_constant() for p in param[2:])`
- Mark all calls to a function where any of the parameters are constant: `any(p.is_constant() for p in param)`
- Mark all calls to a function: `True`
- Mark all calls to a function where the second paramter is not constant and is not checked with `strlen` : `not param[1].is_constant() and not param[1].used_in_call_before(["strlen"])`
- Mark all calls to a function which are reachable from `read` function: `function_call.reachable_from("read")`

Issues and Warnings

- When you request the parameter with index that is out of bounds any call to a function will be marked as `Low` priority. This is a way to avoid missing cross references where it was not possible to correctly get all parameters (this mainly applies to disassembly mode).
- When you search within the VulFi view and change context out of the view and come back, the view will not load. You can solve this either by

terminating the search operation before switching the context, moving the VulFi view to the side-view so that it is always visible or by closing and re-opening the view (no data will be lost).
- Scans for more exotic architectures end with a lot of false positives.

---

ttddbg - by: Simon Garrelou, Sylvain Peyrefitte of the Airbus CERT Team

Plugin description

> ttddbg is a debugger plugin for IDA Pro which can read Time Travel Debugging traces generated by WinDBG or Visual Studio

readme for ttddbg

ttddbg - Time Travel Debugging IDA plugin

This plugin adds a new debugger to IDA which supports loading Time Travel Debugging traces generated using WinDBG Preview.

This plugin supports both x86 and x64 traces, and by extension IDA and IDA64.

Installation

Installing the plugin can be done using the installer from the releases page . The installer will automatically install the required dependencies, provided you have a copy of WinDBG Preview installed.

Usage

Once installed, you can use the plugin by selecting the `ttddbg` debugger in the IDA interface, and specifying your `*.run` file as the "Application". For help on generating a `.run` file, see `HOWTO_TIME_TRAVEL.md` .

Icon	Action
	Go to previous breakpoint
	Single step backward (RIP - one instruction)
	Manage the timeline of interesting events (Threads Created/Terminated, Module Loaded/Unloaded, E

Building the project

Prerequisites:

- A copy of the IDA SDK (available from the download center using your IDA Pro credentials)
- A copy of `TTDReplay.dll` (usually in `C:\Program Files\WindowsApps\[WinDBG folder]\amd64\ttd\`)
- A copy of `TTDReplayCPU.dll` (usually in `C:\Program Files\WindowsApps\[WinDBG folder]\amd64\ttd\`)

And let CMAKE do its magic!

```
$ git clone git@github.com:airbus-cert/ttddbg.git --recursive
$ mkdir build
$ cd build
$ cmake ..\ttddbg -DIDA_SDK_SOURCE_DIR=[PATH_TO_IDA_SDK_ROOT_FOLDER] -DCPACK_PACKAGE_INSTALL
$ cmake --build . --target package --config release
```

Developer corner

To create a dev solution:

```
$ git clone git@github.com:airbus-cert/ttddbg.git --recursive
$ mkdir build
$ cd build
$ cmake ..\ttddbg -DIDA_SDK_SOURCE_DIR=[PATH_TO_IDA_SDK_ROOT_FOLDER] -DBUILD_TESTS=ON
```

Credits and references

Greetz to commial for his work on ttd-bindings !

Quokka - by: Alexis Challande

Plugin description

> Quokka is a binary exporter: from the disassembly of a program, it generates an export file that can be used without a disassembler.

readme for Quokka

Quokka

image generated by DALL-E

Table of Contents

Introduction

Quokka is a binary exporter: from the disassembly of a program, it generates an export file that can be used without the disassembler.

The main objective of Quokka is to enable to completely manipulate the binary without ever opening a disassembler after the initial step. Moreover, it abstracts the disassembler's API to expose a clean interface to the users.

Quokka is heavily inspired by BinExport , the binary exporter used by BinDiff.

Installation

Python plugin

The plugin is built in the CI and available in the registry .

It should be possible to install directly from PIP using this kind of commmand:

```
$ pip install quokka-project
```

IDA Plugin

Note: The IDA plugin is not needed to read a `Quokka` generated file. It is only used to generate them.

The plugin is built on the CI and available in the Release tab.

To download the plugin, get the file named `quokka_plugin**.so` .

Usage

Export a file

!!! note

`This requires a working IDA installation.`

- Either using command line:

```
$ idat64 -OQuokkaAuto:true -A /path/to/hello.i64
```

Note: We are using `idat64` and not `ida64` to increase the export speed because we don't need the graphical interface.

- Using the plugin shortcut inside IDA: (by default) Alt+A

Load an export file

```
import quokka

# Directly from the binary (requires the IDA plugin to be installed)
ls = quokka.Program.from_binary("/bin/ls")

# From the exported file
ls = quokka.Program("ls.quokka",  # the exported file
                    "/bin/ls")    # the original binary
```

Building

Build

```
user@host:~/quokka$ cmake -B build \ # Where to build
                          -S . \ # Where are the sources
                          -DIdaSdk_ROOT_DIR:STRING=path/to/ida_sdk \ # Path to IDA SDK
                          -DCMAKE_BUILD_TYPE:STRING=Release \ # Build Type

user@host:~/quokka$ cmake --build build --target quokka_plugin -- -j
```

To install the plugin:

```
user@host:~/quokka$ cmake --install build
```

In any case, the plugin will also be in `build/quokka-install`. You can copy it to IDA's user plugin directory.

```
user@host:~/quokka$ cp build/quokka-install/quokka*64.so $HOME/.idapro/plugins/
```

For more detailed information about building, see Building

Documentation

Documentation is available online at documentation

FAQ

You can see a list of questions here FAQ

---

ida_names - by: Pavel Maksyutin (Positive Technologies)

Plugin description

> IDA-names automatically renames pseudocode windows with the current function name. It can also rename ANY window with SHIFT-T hotkey.

readme for ida_names

IDA-names

IDA-names automatically renames pseudocode windows with the current function name. It can also rename *ANY* window with `SHIFT-T` hotkey.

Install

Just drop `ida_names.py` in plugins folder. Feel free to change default hotkey as well!

Showcase

---

ida_kcpp - by: Uriel Malin and Ievgen Solodovnykov

Plugin description

> An IDAPython module for way more convienent way to Reverse Engineering iOS kernelcaches.

readme for ida_kcpp

ida_kcpp

An IDAPython module for way more convienent way to Reverse Engineering iOS kernelcaches.

Big part of the iOS kernelcache is written by C++, which compiled into complicated binary code use many virtual table derefences rather than explicit direct calls. `ida_kcpp` takes the advanteage of `ida_kernelcache` classes hierarchy reconstruction. It maps the IDB and synchronizes the binary functions and the original virtual methods. It enables navigation through the

iOS kernelcache by double-clicking on c++ virtual methods call, finding their xrefs, and keeping all of it synchronized during live research.

`ida_kcpp` is inspired by ida_medigate but benefits from the unique structure of the iOS kernelcache and provides more powerful and convenient research environment.

Installation

1. Install ida_kernelcache(Cellebrite's fork)
2. Make sure `ida_kernelcache` and `ida_kcpp` are in IDA python path.

Could also be added by adding into `~/.idapro/idapythonrc.py` :

```
import sys
sys.path.append(IDA_KCPP_PATH)
sys.path.append(IDA_KERNELCACHE_PATH)
```

2. clone the repository and symlink `~/.idapro/plugins/ida_kcpp_plugin.py` to `ida_kcpp_plugin.py`
3. Install ida-netnode
4. [Optional]: Install ida-referee if decompiler structs fields xrefs tracking is required

Usage

Preliminary Analysis

1. Open analyzed kernelcache. If `ida_kernelcache` has not run yet, please run it `edit >> Plugins >> Kernel Cache CPP >> ida_kernelcache process kernel`
2. Run `Edit >> Plugins >> Kernel Cache CPP >> Perform Initial sync...` (Some warning messageboxes might appear, you can ignore them for now). This step renames Virtual functions names by setting their prefix to the class they belong. This step also decompiles all the virtual functions, sets their `this` argument, and changes the relevant `::vmethods` member to be a function pointer to the decompiled type. This step can take a while...
3. Now there is a "Synced IDB", but pay attention - the plugin isn't activated yet.

Plugin activating for a synced IDB

`ida_kcpp` automatically propagates names and prototypes, and also changes some aspects of navigation in IDA, hence is not activated automatically but at any point may be activated or deactivated.

Activate `ida_kcpp` by clicking on `edit >> Plugins >> Kernel Cache CPP >> Activate Plugin` (This will install the Hexrays hooks and also run `ida_kernelcache.collect_class_info()`)

Deactivate the plugin

clicking on `Edit >> Plugins >> Kernel Cache CPP >> Deactivate Plugin`

Features overview

Virtual functions synchronizer

Virtual function renaming

When renaming a virtual function one may edit the mangled name (for example `__ZNK10AppleA7IOP9method_86Ev_99`), or change it to human-being names (`AppleA7IOP::method_86`) which will be mangled automatic. The relevant `ANCESTOR::vmethods` member will be renamed as well, and also all of the virtual functions that implement the same virtual method will be renamed too.

Virutal function prototype updating

The same goes for virutal function prototype changing . In this case, the relevant `ANCESTOR::vmethods` member's prototype will be changed to a function pointer to the type that just was set, with setting `this` as `ANCESTOR *` . Also, the prototypes of all of the relevant virtual method implementations of the ANCESTOR's DESCENDANTS will be changed, with setting their `this` to `DESCENTANT *` .

Improved XREFS

- We add an xref from every virtual function to it's abstract `::vmethods` member.
- Using `ida_referee` we can track uses of every field, and especially `::vmethods` member (virtual calls) in the decompiler.
  - We also added `CTRL+SHIFT+Y` hotkey from virtual function, to directly opening the xrefs window to its relevant `::vmethod` member. *IDA has a bug and this only works from the disassembly window and not from the decompiler.*

Improved Navigation

- Decompiler: Double click on a virtual function call pops up a window that allowing the user to choose the implementation of the virtual method.

- Structures window:
  - Double click on virtual method pops up the same window as in the decompiler
  - Fixed an IDA bug that double-click on struct name that contains `::` won't jump.

Future improvements

1. Adding support in IDA7.2 c++ classes compatibility. Require changing ida_kernelcache structs layout which we didn't do yet for legacy reason
2. Adding a name/prototype resolving interface for conflicts.

Meta

Authored by Uriel Malin and Ievgen Solodovnykov of Cellebrite Labs. Developed and tested for IDA8.0 on macOS with python 3.10.2

---

ida_bochs_windows - by: David Reguera Garcia

Plugin description

> Helper script for Windows kernel debugging with IDA Pro on native Bochs debugger (including PDB symbols)

readme for ida_bochs_windows

ida_bochs_windows

Helper script for Windows kernel debugging with IDA Pro on native Bochs debugger (including PDB symbols)

python3 + idapython 7.4

Bochs debugger:

Bochs debugger GUI:

Usage

WARNING : BEFORE OPEN IDA your must set env var: _NT_SYMBOL_PATH to windows symbols, ex:

```
SRV*C:\winsymbols*
```

Edit ”C:\Program Files\IDA Pro 7.7\cfg\dbg_bochs.cfg”

```
BOCHSDBG = "C:\\Users\\leno\\Desktop\\Bochs-pruebas\\bochs\\bochs.exe";
BOCHSRC = "C:\\Users\\leno\\Desktop\\Bochs-pruebas\\bochs\\.bochsrc";
```

Go to IDA .....

Open IDA PRO,

Go to Debugger -> Run -> Local Bochs Debugger

Application:

```
C:\Users\leno\Desktop\Bochs-pruebas\bochs\.bochsrc
```

Cick Debug Options -> Set specific options -> Select Disk image

start a debug session and go to File -> Script File -> ida_bochs_windows.py

This idapython script ask you for bochs symbol file

Done!

Export IDA Names to file for raw Bochs debug

1. Open IDA PRO, start a debug session and go to File -> Script File -> ida_bochs_windows.py
2. Execute ida_names_to_bochs_sym.py
3. Select a file to save info

Use the generated file in Bochs debugger (ldsym global + file path), example:

```
ldsym global "C:\\Users\\Dreg\\bochs\\bochs_syms.txt"
```

Export IDA Segments to file for raw Bochs debug

1. Open IDA PRO, start a debug session and go to File -> Script File -> ida_bochs_windows.py
2. Execute ida_segs_to_bochs_sym.py
3. Select a file to save info

Use the generated file in Bochs debugger (ldsym global + file path), example:

```
ldsym global "C:\\Users\\Dreg\\bochs\\bochs_segs.txt"
```

Join bochs_segs.txt and bochs_syms.txt

It can be useful have segments + symbols together:

```
type bochs_segs.txt > bochs_segs_and_syms.txt
type bochs_syms.txt >> bochs_segs_and_syms.txt
```

Now, when a instruction its out of a known segment its easy to view:

Demo video

https://youtu.be/X8bJ421iaVA

Related

Helper script for Windows kernel debugging with IDA Pro on VMware + GDB stub (including PDB symbols):

- https://github.com/therealdreg/ida_vmware_windows_gdb

Helper scripts for windows debugging with symbols for Bochs and IDA Pro (PDB files). Very handy for user mode <--> kernel mode:

- https://github.com/therealdreg/symseghelper

Helper script for Linux kernel debugging with IDA Pro on VMware + GDB stub (including some symbols helpers):

- https://github.com/therealdreg/linux_kernel_debug_disassemble_ida_vmware

Dump PDB Symbols including support for Bochs Debugging Format (with wine support):

- https://github.com/therealdreg/pdbdump_bochs

Tools for Linux kernel debugging on Bochs (including symbols, native Bochs debugger and IDA PRO):

- https://github.com/therealdreg/bochs_linux_kernel_debugging

Credits

Based on original IDA-VMware-GDB By Oleksiuk Dmytro (aka Cr4sh) https://github.com/Cr4sh/IDA-VMware-GDB

FirmLoader - by: Martin Petran (Accenture)

Plugin description

An alternative to SVD loader that uses simpler JSON files

readme for FirmLoader

FirmLoader

The FirmLoader is an IDA plugin that allows to automatically identify parts of the memory for the firmware images extracted from microcontrollers. This simplifies the process of understanding the binary contents and thus allows you to achieve your reversing goals more efficiently. The plugin is similar to the built-in SVD loader; however, it offers a simpler data structure that could be created manually from the publicly available documentation.

Installation

Copy the `firmloader.py` and `firmloader_data` folder to the IDA plugin folder (for example: `cp -R fimrloader* <IDA_FOLDER>/plugins`).

Usage

Load the binary into IDA, make sure that the base address is in accordance with the documentation for the given processor, set the name of the main code segment to `ROM` and then use `Edit > FirmLoader` menu to select the MCU model. Note that this will trigger the auto-analysis of the binary.

Features

The main motivation for creating this plugin was to make it possible to create simple JSON structures from documentation to supplement the SVD files and add couple more features:

Add basic RAM automatically

With FirmLoader plugin you do not have to create RAM section as it will create it for you automatically (including the RAM sections that are split across multiple locations). Note that when the series of MCU offers several sizes of memory all units from that series were generated with the highest portion of RAM space.

Interrupt Vector Table

Another enhancement over the SVD loader features allows FirmLoader to set the interrupt vector table (where applicable) and mark the items as pointers to the code section to allow easy inspection of various event handlers right away.

Manually create description files

In case that the MCU is of a different architecture, the structure of the metadata used by this plugin is much simpler than SVD files and thus can be easily created from the documentation manually. Doing this will avoid the lengthy process of identifying peripherals, segments, and interrupt handlers when the same model

of MCU is encountered again in the future. In case that some of the information is not available or is not applicable for the given target just leave the irrelevant sections of the data_template.json empty (do not delete them). Example of a purely manually created data file is the SPC50B64x .

Template for the data file:

```
{   // All numbers in the file except the "bits" field use hexadecimal format to make it eas
    "brand": "MANUFACTURER", // Manufacturer of the MCU (used for menu items)
    "family": "SERIES", // Series (used for menu items)
    "name": "NAME", // Name of the MCU (used for menu items)
    "bits": 32, // Address size of the MCU in bits
    "mode": 1,  // Mode of operation (ARM only at this point) 0 - ARM, 1 - Thumb
    "segments": [
        {
            "name": "DATA_FLASH", // Name of the segment
            "start": "0x0", // Base address
            "end": "0x7ffff",   // End address
            "type": "DATA"  // Type of the segment (either CODE or DATA)
        },
        {
            "name": "SRAM",
            "start": "0x20000000",
            "end": "0x200fffff",
            "type": "DATA"
        }
    ],
    "peripherals": [ // List of peripherals
        {
            "name": "PERIPHERAL_NAME", // Name of the peripheral
            "start": "0x40000000",  // Start address
            "end": "0x40003fff",    // End address
            "comment": "",  // Any comment for the peripheral
            "registers": [  // List of registers
                {
                    "name": "REGISTER_NAME",    // Register name
                    "offset": "0x0"    // Offset from the start of the peripheral
                }
            ]
        }
    ],
    "vector_table": [ // Interrupt Vector Table
        {
            "name": "Reset", // Name of the interrupt
            "value": -1,    // Value
            "addr": "0x04", // Offset of the vector from the start of the ROM segment
            "comment": "Priority: -3"   // Any comment
```

```
        }
    ]
}
```

FindFunc - by: Felix B.

Plugin description

> [...] an IDA Pro python3 plugin to find\/filter code functions that contain a certain assembly or byte pattern, reference a certain name or string, or conform to various other constraints

readme for FindFunc

FindFunc: Advanced Filtering/Finding of Functions in IDA Pro

FindFunc is an IDA Pro plugin to find code functions that contain a certain assembly or byte pattern, reference a certain name or string, or conform to various other constraints.

FindFunc won third place in the 2022 HexRays IDA Pro Plugin Contest !

Filtering with Rules

The main functionality of FindFunc is letting the user specify a set of "Rules" or constraints that a code function in IDA Pro has to satisfy. FF will then find and list all functions that satisfy ALL rules (so currently all Rules are in an AND-conjunction). Exception: Rules can be "inverted" to be negative matches. Such rules thus conform to "AND NOT".

FF will schedule the rules in a smart order to minimize processing time. Feature overview:

- Currently 6 Rules available, see below
- Aware of function chunks
- Smart scheduling of rules for performance
- Saving/Loading rules from/to file in simple ascii format
- Several independent tabs for experimentation
- Copying rules between tabs via clipboard (same format as file format)
- Saving entire session (all tabs) to file
- Advanced copying of instruction bytes (all, opcodes only, all except immediates, ...)

- Clone tabs for quick experiments with refining results
- Code matching respects Addressing-Size-Prefix and Operand-Size-Prefix

Button "Find Functions" clears existing results and starts a fresh search, "Refine Results" considers only results of the previous search.

Advanced Binary Copying

A secondary feature of FF is the option to copy binary representation of instructions with the following options:

- copy all -> copy all bytes to the clipboard
- copy without immediates -> blank out (AA ?? BB) any immediate values in the instruction bytes
- opcode only -> will blank out everything except the actual opcode(s) of the instruction (and prefixes)
- ...

See "advanced copying" section below for details. This feature nicely complements the Byte Pattern rule!

Building and Installation

FindFunc is an IDA Pro python plugin without external package dependencies. It can be installed by downloading the repository and copying file 'findfunc-main.py' and folder 'findfunc' to your IDA Pro plugin directory. No building is required.

Requirements: IDA Pro 7.x (7.6+) with python3 environment. FindFunc is designed for x86/x64 architecture only. It has been tested with IDA 7.6/7.7, python 3.9 and IDAPython 7.4.0 on Windows 10.

Available Rules

Currently the following six rules are available. They are sorted here from heavy to light with regard to performance impact. With large databases it is a good idea to first cut down the candidate-functions with a cheap rule, before doing heavy matching via e.g. Code Rules. FF will automatically schedule rules in a smart way.

Code Pattern

Rule for filtering function based on them containing a given assembly code snippet. This is NOT a text-search for IDAs textual disassembly representation, but rather performs advanced matching of the underlying instruction. The snippet may contain many consecutive instructions, one per line. Function chunks are supported. Supports special wildcard matching, in addition to literal assembly:

- ”pass” -> matches any instruction with any operands
- ”mov* any,any” -> matches instructions with mnemonic ”mov*” (e.g. mov, movzx, ...) and any two arguments.
- ”mov eax, r32” -> matches any instruction with mnemonic ”mov”, first operand register eax and second operand any 32-bit register.
  - Analogue: r for any register, r8/r16/r32/r64 for register of a specific width, ”imm” for any immediate
- ”mov r64, imm” -> matches any move of a constant to a 64bit register
- ”any r64,r64” -> matches any operation between two 64bit registers
- mov -> matches any instruction of mov mnemonic

more examples:

```
mov r64, [r32 * 8 + 0x100]
mov r, [r * 8 - 0x100]
mov r64, [r32 * 8 + imm]
pass
mov r, word [eax + r32 * 8 - 0x100]
any r64, r64
push imm
push any
```

Gotchas: Be careful when copying over assembly from IDA. IDA mingles local variable names and other information into the instruction which leads to matching failure. Also, labels are not supported (”call sub_123456”).

Note that Code Patterns is the most expensive Rule, and if only Code Rules are present FF has no option but to disassemble the entire database. This can take up to several minutes for very large binaries. See notes on performance below.

Immediate Value (Constant)

The function must contain the given immediate at least once in any position. An immediate value is a value fixed in the binary representation of the instruction. Examples for instructions matching immediate value 0x100:

```
mov eax, 0x100
mov eax, [0x100]
and al, [eax + ebx*8 + 0x100]
push 0x100
```

Immediates can be pasted directly from clipboard into the list of rules. Note: IDA performs extensive matching of any size and any position of the immediate. If you know it to be of a specific width of 4 or 8 bytes, a byte pattern can be faster (but may produce false positives).

Byte Pattern

The function must contain the given byte pattern at least once. The pattern is of the same format as IDAs binary search, and thus supports wildcards - the perfect match for the advanced-copy feature!

Examples:

```
11 22 33 44 aa bb cc
11 22 33 ?? ?? bb cc -> ?? can be any byte
```

Byte-patterns can be pasted directly from clipboard into the list of rules. Note: Pattern matching is quiet fast and a good candidate to cut down matches quickly!

String Reference

The function must reference the given string at least once. The string is matched according to pythons 'fnmatch' module, and thus supports wildcard-like matching. Matching is performed case-insensitive. Strings of the following formats are considered: [idaapi.STRTYPE_C, idaapi.STRTYPE_C_16] (this can be changed in the Config class).

Examples:

- "TestString" -> function must reference the exact string (casing ignored) at least once
- "TestStr*" -> function must reference a string starting with 'TestStr (e.g. TestString, TestStrong) at least once (casing ignored)

Note: String matching is fast and a good choice to cut down candidates quickly!

Name Reference

The function must reference the given name/label at least once. The name/label is matched according to pythons 'fnmatch' module, and thus supports wildcard-like matching. Matching is performed case-insensitive.

Examples:

- "memset" -> function must reference a name "memset" at least once
- "mem*" -> function must reference a name starting with "mem" (memset, memcpy, memcmp) at least once

Note: Name matching is very fast and ideal to cut down candidates quickly!

Function Size

The size of the function must be within the given limit: "min <= functionsize <= max". Data is entered as a string of the form "min,max". The size of a function includes all of its chunks.

Note: Function size matching is very fast and ideal to cut down candidates quickly!

Keyboard Shortcuts & GUI

For ease of use FF can be used via the following keyboard shortcuts:

- Ctrl+Alt+F -> launch/show TabWidget (main GUI)
  - Or View->FindFunc
- Ctrl+F -> start search with currently enabled rules
- Ctrl+R -> refine existing results with currently enabled rules
- Rules
  - Ctrl+C -> copy selected rules to clipboard
  - Ctrl+V -> paste rules from clipboard into current tab (appends)
  - Ctrl+S -> save selected rules to file
  - Ctrl+L -> load selected rules from file (appends)
  - Ctrl+A -> select all rules
  - Del -> delete selected rules
- Save Session
  - Ctrl+Shift+S -> Save session to file
  - Ctrl+Shift+L -> Load session from file

Further GUI usage

- Right-click on tab: option to clone tab
- Rules can be edited by double-clicking the Data column
- Rules can be inverted (negative match) by double-clicking the invert-match column
- Rules can be enabled/disabled by double-clicking the enabled-column
- Wheel-clicking any cell in Rules or Results copies the cell to clipboard
- Byte-Patterns and immediates can be pasted directly from clipboard into the Rules table
- Tabs can be renamed by double-clicking them
- Sorting is supported both for Rule-List and Result-List
- Double-click Result item to jump to it in IDA
  - function name: jump to function start
  - any other column: jump to match of last matched rule
- Checkbox Profile: Outputs profiling information for the search
- Checkbox Debug: Dumps detailed debugging output for code rule matching - only use it if few functions make it to the code checking rule, otherwise it might take very long!

Advanced Binary Copy

Frequently we want to search for binary patterns of assembly, but without hardcoded addresses and values (immediates), or even only the actual opcodes of the instruction. FindFunc makes this easy by adding three copy options to

the disassembly popup-menu:

Copy all bytes

Copies all instruction bytes as hex-string to clipboard, for use in a Byte-Pattern-Rule (or IDAs binary search).

```
B8 44332211      mov eax,11223344
68 00000001      push 1000000
66:894424 70     mov word ptr ss:[esp+70],ax
```

will be copied as

```
b8 44 33 22 11 68 00 00 00 01 66 89 44 24 70
```

Copy only non-immediate bytes

Copies instruction bytes for given instruction, masking out any immediate values. Example:

```
B8 44332211      mov eax,11223344
68 00000001      push 1000000
66:894424 70     mov word ptr ss:[esp+70],ax
```

will be copied as

```
b8 ?? ?? ?? ?? 68 ?? ?? ?? ?? 66 89 44 24 ??
```

Copy only opcodes

Copy all instruction bytes as hex-string to clipboard, masking out any bytes that are not the actual opcode (including sib, modrm, but keeping legacy prefixes).

```
B8 44332211      mov eax,11223344
68 00000001      push 1000000
66:894424 70     mov word ptr ss:[esp+70],ax
```

will be copied as

```
b8 ?? ?? ?? ?? 68 ?? ?? ?? ?? 66 89 ?? ?? ??
```

Copy opcodes + immediates

A combination that keeps opcode and immediates, but masks e.g. mod r/m. Note that this keeps opcodes even if they mandate a specific registers, e.g.

```
B8 44332211      mov eax,11223344
```

will keep the opcode B8.

Note: This is a ”best effort” using IDAs API, thus there may be few cases where it only works partially. For a 100% correct solution we would have to ship a dedicated x86 disasm library.

Similar results can be achieved with Code Pattern Rules, but this might be faster, both for user interaction and the actual search.

Copy disasm

Copies selected disassembly to clipboard, as it appears in IDA.

Performance

A brief word on performance:

1. name, string, funcsize are almost free in all cases
2. bytepattern is almost free for patterns length > 2
3. immediate is difficult: We can use idaapi search, or we can disassemble the entire database and search ourselves - we may have to do this anyways if we are looking for code patterns. BUT: scanning for code patterns is in fact much cheaper than scanning for an immediate. An api-search for all matches is relatively costly - about 1/8 as costly as disassembling the entire database. So: If we cut down matches with cheap rules first, then we greatly profit from disassembling the remaining functions and looking for the immediate ourselves, especially if a code-rule is present anyways. However: If no cheap options exist and we have to disassemble large parts of the database anyways (due to presence of code pattern rules), then using one immediate rule as a pre-filter can greatly pay off. api-searching ONE immediate is roughly equivalent to 1/8 searching for any number of code-pattern rules - although this also depends on many different factors...
4. code pattern are the most expensive by far, however checking one pattern vs checking many is very similar.

Todo (unordered):

- jcc pseudo-mnemonic
- Allow named locations in CodeRules ('call memset')
- add rule xref in/out
- undo/redo
- progress bar
- middle-click to close tabs
- tab menu: close all but this
- 'ignore all following operands' option
- Rule for parameters to API calls inside function
- Rule for parent/callsite/child function requirements
- Rule for function parameters
- Regex-rule
- string/name: casing option
- automatically convert immediate rules to byte pattern if applicable?
- settings: case sensitivity, string types, range, ...
- Hexray rules?

- OR combination of rules
- Pythonification of code ;)
- Parallelization
- Automatic generation of rules to identify a function?

Condstanta - by: Martin Petran (Accenture)

Plugin description

Plugin to search for constants used in conditional statements

readme for Condstanta

Condstanta v1.0

Introduction

Condstanta is a plugin that allows searching for constant values that are used in conditional statements such as `if` and `switch-case` or for functions that contain multiple specific constants. The plugin allows searching for exact numbers, number ranges and list of specific constants.

Installation

Copy the `condstanta.py` file to IDA plugin folder.

Usage

The plugin can be started from `Search` menu by selecting the `Condstanta` option. The dialog window for search specification offers several fields:

- Search for : The value to look for. Possible values are:
  - Exact numbers: `100`, `0xabc` or `-2.3`
  - Ranges (both edge cases are included): `0:10`, `-10:0xf`, `3:3.9`
  - List of values: `-1, 2, 0x45, 1.3`
  - Empty field means find all numbers (only applicable for the `Search for constants used in conditions`)

- What to search for : Type of the search that is to be performed. `Search for constants used in conditions` will only look for constants that are used in `if` , `case` and `while` statements. The option `Find set of constants in single function` will take range or list of values and find all functions that contain multiple of the constants from the given list (the required amount is determined by the `Minimal number of matching constants` which must be set to 2 or higher).
- Use Hexrays : Whether to use Hexrays decompiler for the search or not (only usable when the decompiler is available).

The results of the search operation are presented in form of a new view with a table (shown below). Double-click on a row will jump to the location marked in the `Address` column.

- Address - Address where the constant was found.
- Function - Name of the function where this constant was found.
- Type - `IF` , `CASE` and `LOOP` . Only applicable to `Search for constants used in conditions` , otherwise empty.
- Decimal - Decimal value of the number.
- Hex - Hex value of the number (empty for floats).
- Comment - Extracted comment from the location where the constant was detected. In Hexrays mode shows the Hexrays comment, otherwise comments in disassembly.

Issues and Warnings

- The plugin performs the search only in body of functions and thus it will not look at any disassembly that is not marked as a function.
- Hexrays mode available only for limited set of architectures and is slower (it must decompile all functions).
- Assembly mode is attempting to be architecture agnostic; this may occasionally lead to missed constants or false positive hits with architectures such as ARM or PPC which use two instructions to load large numbers. With some more exotic architectures, this mode might miss constants completely.
- Using Hexrays mode will show Hexrays comments only (CASE comments do not work), using assembly mode shows the assembly comments only.
- Float constants are only found with Hexrays.

Yagi - by: Sylvain Peyrefitte (Airbus CERT)

Plugin description

> Yagi is a C++ plugin that includes the Ghidra decompiler into IDA 7.5 and 7.6.

readme for Yagi

---

wilhelm - by: Xinyu Zhuang

Plugin description

> wilhelm is a Python API that provides a better interface for working with Hex-Rays. In particular, I designed it with the IDAPython REPL (aka console) in mind; while it works fine in scripts, it's meant to be used interactively to quickly automate so

readme for wilhelm

wilhelm: Alternative API for IDA and Hex-Rays

wilhelm is an API for working with IDA, and in particular the Hex-Rays decompiler. It aims to wrap around the existing SDK's API, plus provide additional features and concepts that make reverse engineering easier.

While wilhelm works well in scripts, it is also designed with the REPL in mind: it is tailored to be easy to use interactively, to help answer simple questions while reversing is taking place. For example, you can use it to search a group of functions for a specific code pattern. As such, wilhelm contains an event system, allowing it to react and update itself whenever the underlying IDB is modified.

Currently, the main feature provided by wilhelm is convenient access to a decompiled function's AST. The next major feature to be added is type management.

Example Usage

Initialize:

```
>>> import wilhelm as W
>>> W.initialize(Feature.PATH, Feature.MODULE)
```

Access the AST of some function in the current module:

```
>>> func = W.current().values["sub_12345"]().func
>>> func.body[0]
```

```
<wilhelm.ast.IfStmt at 0xXXXXXXXXXXXX>

>>> func.body[0].expr.op
<OP.UGT: 32>
```

Find all call expressions in the function:

```
>>> list(func.select("*/CallExpr"))
[<wilhelm.ast.CallExpr at 0xXXXXXXXXXXXX>,
 <wilhelm.ast.CallExpr at 0xXXXXXXXXXXXX>,
 <wilhelm.ast.CallExpr at 0xXXXXXXXXXXXX>,
 <wilhelm.ast.CallExpr at 0xXXXXXXXXXXXX>,
 <wilhelm.ast.CallExpr at 0xXXXXXXXXXXXX>]
```

Get the names of the callee of the call expressions:

```
>>> [W.current().get_qname_for_addr(e.addr) for e in func.select("*/CallExpr.e_func/")]
[QName<sub_1412C0>,
 QName<sub_153DD0>,
 QName<sub_15DA70>,
 QName<sub_165120>,
 QName<sub_1664F0>]
```

Get all calls expressions that are calling function at address `0x43213` :

```
>>> calls = func.select("*/CallExpr[.e_func/GlobalVarExpr{addr = 0x43213}]")
>>> list(calls)
[<wilhelm.ast.CallExpr at 0xXXXXXXXXXXXX>,
 <wilhelm.ast.CallExpr at 0xXXXXXXXXXXXX>,
 <wilhelm.ast.CallExpr at 0xXXXXXXXXXXXX>]
```

Get string value of 2nd argument to the above calls:

```
>>> [e.params[1].value for e in calls if isinstance(e.params[1], W.ast.StrExpr)]
[b'command', b'description']
```

Dependencies

Requires IDAPython 3, no support for Python 2.

wilhelm requires a working async event loop in IDAPython. The easiest way to get this is by installing qasync , which provides a Qt-based event loop. This loop must be initialized prior to loading wilhelm.

The optional `path` feature requires pyparsing .

Not a dependency, but using ipyia makes using wilhelm a lot easier.

Installation

wilhelm has yet to be properly packaged. For now, you can use it by cloning the repository and adding the python/ subdirectory to your `sys.path` somehow.

Note that you need an async event loop setup before you load wilhelm. If you're using qasync, you can add something like this to your `idapythonrc` :

```
# Sync asyncio and Qt event loop
from PyQt5.QtWidgets import QApplication
import qasync
import asyncio
qapp = QApplication.instance()
loop = qasync.QEventLoop(qapp, already_running=True)
asyncio.set_event_loop(loop)
```

Configuration

```
import wilhelm as W
W.initialize()                              # Init with only core features
# or:
W.initialize(W.Feature.PATH, W.Feature.MODULE) # Init with optional features
```

Features

Abstract Syntax Tree Access

wilhelm provides a more object-oriented/Pythonic way of accessing a decompiled function's AST. Nodes in the AST have a different class based on the kind of nodes they are, and expose relevant values as fields. A Visitor class can be used to traverse the AST.

A NodeList represents a collection of AST nodes, and provides ways of mapping and filtering the list. This can be used to quickly locate a specific node of interest.

```
>>> list(nodelist.filter_class(W.ast.BinOpExpr).filter_test(lambda n: n.op == W.ast.OP.EQ))
[<wilhelm.ast.BinOpExpr at 0xXXXXXXXXXXXX>]
```

AST Wilpaths

The optional `path` feature provides *wilpaths* , which are a way to easily navigate and select nodes in an AST. Inspired by the XPath query language for XML, a wilpath builds upon the filtering and mapping features of NodeLists.

Some examples of wilpaths:

- `IfStmt` Returns all if statements found in the current node list.

- `/IfStmt` Returns all children that are if statements.
- `*/IfStmt` Returns any descendent that is an if statement.
- `*/IfStmt.expr/` Returns the condition expression of all if statement descendents.
- `*/IfStmt.expr/*/GlobalVarExpr` Returns all global variable expressions that are found within an if statement.
- `*/IfStmt.expr/*/GlobalVarExpr{addr = 0x1234}` The above, but only those global variable expressions which have an address of 0x1234.
- `*/IfStmt[.expr/*/GlobalVarExpr{addr = 0x1234}]` The above, but instead of returning the global variable expressions, return the parent if statement.

Please see the docstring in path.py for a complete description of the wilpath DSL.

Event System

wilhelm uses an event system that allows users to register and observe various kinds of events happening within IDA. For example, a callback can be added to trigger whenever some property of a function changes.

Events can propagate, such that one could observe all events happening the children of a parent object, and vice versa.

Currently, the event system is only integrated with the naming system (QNames), but eventually will be available in other features as well, particularly the type system.

Module Representation

The `module` feature provides a way of accessing the currently-loaded IDA database (aka module). All objects in the database have an associated *qualified name* (QName), which is kept in sync with the name used by IDA. QNames allow navigation and searching based on their structure: e.g. you can query for all names that are in a particular namespace like `foo::SomeClass`. Renaming a namespace also automatically updates the names within that namespace.

Querying a name returns a representation of the object. For functions, the AST of the function can be easily accessed via this representation:

```
wilhelm.current().values["sub_12345"]().func.body[0].expr.e_lhs
```

The module feature is currently in development, and hence optional, but it will eventually form a core part of wilhelm.

Known Bugs

- The AST representation does not update when the function gets updated. This will be fixed soon; the AST code was written before the event code, so it needs to be updated to react to events like variable renaming.

Credits

TODO

License

GNU General Public License v3.0

See LICENSE for full text.

Tenet - by: Markus Gaasedelen

Plugin description

> Tenet is an IDA Pro plugin which enables reverse engineers to explore execution traces of native code. It demonstrates how visualization can augment time-travel-debugging technologies to create more fluid controls for exploring the execution runtime.

readme for Tenet

Tenet - A Trace Explorer for Reverse Engineers

Overview

Tenet is an IDA Pro plugin for exploring execution traces. The goal of this plugin is to provide more natural, human controls for navigating execution traces against a given binary. The basis of this work stems from the desire to research new or innovative methods to examine and distill complex execution patterns in software.

For more context about this project, please read the blogpost about its initial release.

Special thanks to QIRA / geohot et al. for the inspiration.

Releases

- v0.2 -- Imagebase detection, cell visualization, breakpoint refactor, bug-fixes.
- v0.1 -- Initial release

Installation

Tenet is a cross-platform (Windows, macOS, Linux) Python 3 plugin. It takes zero third party dependencies, making the code both portable and easy to install.

1. From your disassembler's python console, run the following command to find its plugin directory:
   - IDA Pro : `import idaapi, os; os.path.join(idaapi.get_user_idadir(), "plugins")`
2. Copy the contents of this repository's `/plugins/` folder to the listed directory.
3. Restart your disassembler.

This plugin is only supported for IDA 7.5 and newer.

Usage

Once properly installed, there will be a new menu entry available in the disassembler. This can be used to load externally-collected execution traces into Tenet.

As this is the initial release, Tenet only accepts simple human-readable text traces. Please refer to the tracing readme in this repository for additional information on the trace format, limitations, and reference tracers.

Bidirectional Exploration

While using Tenet, the plugin will 'paint' trails to indicate the flow of execution forwards (blue) and backwards (red) from your present position in the active execution trace.

To `step` forwards or backwards through time, you simply *scroll while hovering over the timeline* on the right side of the disassembler. To `step over` function calls, hold `SHIFT` while scrolling.

Trace Timeline

The trace timeline will be docked on the right side of the disassembler. This widget is used to visualize different types of events along the trace timeline and perform basic navigation as described above.

By *clicking and dragging across the timeline* , it is possible to zoom in on a specific section of the execution trace. This action can be repeated any number of times to reach the desired granularity.

Execution Breakpoints

Double clicking the instruction pointer in the registers window will highlight it in red, revealing all the locations the instruction was executed across the trace timeline.

To jump between executions, *scroll up or down while hovering the highlighted instruction pointer* .

Additionally, you can *right click in the disassembly listing* and select one of the navigation-based menu entries to quickly seek to the execution of an instruction of interest.

IDA's native `F2` hotkey can also be used to set breakpoints on arbitrary instructions.

Memory Breakpoints

By double clicking a byte in either the stack or memory views, you will instantly see all reads/writes to that address visualized across the trace timeline. Yellow indicates a memory *read* , blue indicates a memory *write* .

Memory breakpoints can be navigated using the same technique described for execution breakpoints. Double click a byte, and *scroll while hovering the selected byte* to seek the trace to each of its accesses.

Right clicking a byte of interest will give you options to seek between memory read / write / access if there is a specific navigation action that you have in mind.

To navigate the memory view to an arbitrary address, click onto the memory view and hit `G` to enter either an address or database symbol to seek the view to.

Region Breakpoints

It is possible to set a memory breakpoint across a region of memory by highlighting a block of memory, and double clicking it to set an access breakpoint.

As with normal memory breakpoints, hovering the region and *scrolling* can used to traverse between the accesses made to the selected region of memory.

Register Seeking

In reverse engineering, it's pretty common to encounter situations where you ask yourself *"Which instruction set this register to its current value?"*

Using Tenet, you can seek backwards to that instruction in a single click.

Seeking backwards is by far the most common direction to navigate across register changes... but for dexterity you can also seek forward to the next register assignment using the blue arrow on the right of the register.

Timestamp Shell

A simple 'shell' is provided to navigate to specific timestamps in the trace. Pasting (or typing...) a timestamp into the shell with or without commas will suffice.

Using an exclamation point, you can also seek a specified 'percentage' into the trace. Entering `!100` will seek to the final instruction in the trace, where `!50` will seek approximately 50% of the way through the trace. `!last` will seek to the last navigable instruction that can be viewed in the disassembler.

Themes

Tenet ships with two default themes -- a 'light' theme, and a 'dark' one. Depending on the colors currently used by your disassembler, Tenet will attempt to select the theme that seems most appropriate.

The theme files are stored as simple JSON on disk and are highly configurable. If you are not happy with the default themes or colors, you can create your own themes and simply drop them in the user theme directory.

Tenet will remember your theme preference for future loads and uses.

FAQ

Q: How do I record an execution trace using Tenet?

- *A: Tenet is a trace reader, not a trace recorder. You will have to use dynamic binary instrumentation frameworks (or other related technologies) to generate a compatible execution trace. Please refer to the tracing readme for more information on existing tracers, or how to implement your own.*

Q: What trace architectures does Tenet support loading?

- *A: Only x86 and AMD64, but the codebase is almost entirely architecture agnostic.*

Q: How big of a trace file can Tenet load / navigate?

- *A: Tenet's trace reader is pure python, it was written as an MVP. There is no guarantee that traces which exceed 10 million instructions will be reasonable to navigate until a native backend replaces it.*

Q: I loaded an execution trace, now there is a '.tt' file. What is it?

- *A: When Tenet loads a given text trace, it will parse, index, and compress the trace into a more performant format. On subsequent loads, Tenet will attempt to load the '.tt' file which should load in fraction of the time that it would take to load the original text trace.*

Q: The plugin crashed / threw an error / is showing bad trace information, what should I do?

- *A: If you encounter an issue or inaccuracy that can be reproduced, please file an issue against this repository and upload a sample trace + executable.*

Q: Memory in my trace is changing, but there are no writes to the region. Is this a bug!?

- *A: Your log file may not have captured all memory writes. For example, usermode DBI generally do not get a memory callback for external writes to process memory. This is most common when reading from a file, or from socket -- it is the kernel that writes memory into your designated usermode buffer, making the event invisible to traditional instrumentation.*
  - Microsoft TTD generally exhibits the same behavior, it's tricky to solve without modeling syscalls.

Q: Will this be ported to Binary Ninja / Ghidra / ... ?

- *A: Possibly, but not anytime soon (unless there is significant incentive). As a research oriented project, the driving motivation is on developing novel strategies to organize and explore program execution -- not porting them.*

Q: My organization would like to support this project, how can we help?

- *A: Without funding, the time I can devote to this project is limited. If your organization is excited by the ideas put forth here and capable of providing capital to sponsor dedicated R&D, please contact us .*

Future Work

Time and ~~motivation~~ funding permitting, future work may include:

- Filtering / coagulating library calls from traces
- Pointer analysis (e.g. annotations) for the register / stack views
- Native TraceFile & TraceReader implementations (e.g. bigger and faster traces)
- Navigation history + bookmarks view (maybe 2-in-1?)

- Richer trace informatics, more aggressive indexing of relevant events (e.g. function calls)
- Trace cartography, improved summarization and representation of trace geography
- Make the 'cpu architecture' selection/detection slightly less hardcoded
- More out-of-the-box tracing bridges, DynamoRIO, TTD, RR, QEMU, Bochs, ...
- Support for Hex-Rays / decompiled views (besides basic view sync)
- Improved workflow for automatically loading or iterating on traces
- Differential analysis, high level 'trace diffing'
- Better navigation and breakdown of threads, quantum's
- Better support for navigating 'multi module' traces (e.g. full system traces)
- Binary Ninja support
- ... ?

I welcome external contributions, issues, and feature requests. Please make any pull requests to the `develop` branch of this repository if you would like them to be considered for a future release.

Authors

- Markus Gaasedelen (@gaasedelen)

SyncReven - by: Cyrille Bagard

Plugin description

> Axion is the main application of the Reven platform, which captures a time slice of a full system execution. It can then be connected to many tools, including IDA Pro, for the analysis. A plugin to synchronize the IDA view from inside Reven already e

readme for SyncReven

Introduction

Axion is the main application of the Reven platform , which captures a time slice of a full system execution.

It can then be connected to many tools, including IDA Pro , for the analysis. A plugin to synchronize the IDA view from inside Reven already exists, but there is no plugin for the reverse direction yet.

Thus, the SyncReven plugin for IDA has been developed in order to reach two goals:

- synchronize the Axion current analysis window with some code opened in IDA;
- discover the Python API available for extending the IDA GUI features.

The aim of the plugin is to quickly target the currently running code for a given position (dynamic view), which can at first hold encrypted data for instance (static view).

As a side effect, browsing code inside IDA can also provide some kind of code coverage because the number of captures is displayed for each of the browsed instructions.

Installation process

The Reven Python bindings can be transposed from an original Reven system (Debian Buster) to a more recent user environment (Debian Bullseye).

The packages are stored in the installation directory. Some renaming tricks are mandatory to get wheels with compatible ABI, considering that:

- the `m` flag for pymalloc became useless since Python 3.8;
- the command pip3 debug --verbose provides all compatible tags.

```
scp -r user@server:/opt/reven-2.8.1-professional-mecha-buster/packages .

cp packages/reven2-2.8.1-py2.py3-none-any.whl .
cp packages/reven_api-2.8.1-cp37-cp37m-linux_x86_64.whl reven_api-2.8.1-cp39-none-linux_x86_
```

A virtual environment is not mandatory, but it offers a clean isolated setup:

```
apt-get install python3-venv

python3 -m venv py4reven

source py4reven/bin/activate

pip3 install reven2-2.8.1-py2.py3-none-any.whl reven_api-2.8.1-cp39-none-linux_x86_64.whl
```

Some extra dependencies and tricks may be needed:

```
apt-get install libboost-python1.74.0

ln -s /usr/lib/x86_64-linux-gnu/libboost_python39.so.1.74.0 /usr/lib/x86_64-linux-gnu/libboo

apt-get install libboost-date-time1.74.0

ln -s /usr/lib/x86_64-linux-gnu/libboost_date_time.so.{1.74.0,1.67.0}
```

Finally, the `syncreven.py` script has to be put inside IDA's `plugins` directory.

Workflow

The SyncReven plugin is composed of three parts:

- the `TransitionsPanel` class inherits `idaapi.PluginForm` : it extends IDA with a graphical panel and sends a transition to be displayed by Reven when needed;
- the `NavTracker` inherits `idaapi.UI_Hooks` and changes according to the current effective address;
- the `SyncWithReven` inherits `idaapi.plugin_t` : it is built from the `PLUGIN_ENTRY` entrypoint and leads all the previous classes.

At the beginning, the cursor inside IDA changes. A request is sent to REVEN to get an iterator over all the transitions for the current position:

```
found = self._server.trace.search.pc(new_ea)
```

On user request, a transition ID is later used to force the relative view inside Axion:

```
trans = self._server.trace.transition(tid)

ret = self._server.sessions.publish_transition(trans)
```

Simple example

The virtual environment providing the Python bindings has to be activated:

```
$ source py4reven/bin/activate
(py4reven) $
```

A project can then be opened in IDA from the same shell prompt. The message `Starting syncreven` should appear in the Output window.

A empty panel expects configuration parameters before displaying available Reven transition identifiers for the current IDA cursor:

IP address and listening port can be retrieved from the Reven target scenario (analysis tab) :

Activating the `Connect` button should print the following message in the Output window :

```
Reven server on ZZZZZZ (xx.xx.xx.xx:yyyy) [connected]
```

The plugin is now connected to the server, but Axion also has to be configured for a running session, which can be assigned using a combo box at the top of the Axion window:

Axion can then get fully notified by IDA.

Back to the IDA window, each time the current effective address changes, the transitions panel is updated.

Let's try with a small simple loop, which is run four times:

Once an instruction from the loop is selected, its four relative transition identifiers are displayed:

Double-clicking on one transition moves the Axion view to the selected captured transition relative to the current IDA view.

RefHunter - by: Jiwon Choi

Plugin description

> RefHunter find all references in simple and lightweighted manner. - User-friendly view - Runs without any 3rd-party application - Runs without installing itself, it's just portable. - Analyze the function and show tiny little report for you!

readme for RefHunter

IDARefHunter

Updated : This project's been introduced on IDA Plugin Contest 2021 !

Why do we need RefHunter?

Getting reference information in *one specific function* is the secret to find out the connection between lines.

- Comparing two function's *subroutine lists* is one of the simplest diffing tequnique.
- By just seeing the *string list* referenced in the function, we can infer the sketchy role of the function.

However, among the all of fancy features in IDA, getting *all reference information* in *one specific function* hasn't available so far.

That's where " *RefHunter: User-friendly function reference finder* " comes in.

RefHunter

RefHunter find all references in simple and lightweighted manner.

- User-friendly view
- Runs without any 3rd-party application
- Runs without installing itself, it's just portable.
- Analyze the function and show tiny little *report* for you!

Usage

Key	Description
Ctrl + H	Open the Ref H unter view
ESC	Close the RefHunter view
C or c	C olor the selected reference in assembly line
R or r	R efresh view

Installation

1. Download this repository
2. Open IDA > [File] > [Script File] > Select *RefHunter.py*

Requirement

- IDA version 7 >= with IDAPython
- Tested on 7.6 with Python 2.7.16

Video guide

Short video guide (<2min)

---

qscripts - by: Elias Bachaalany

Plugin description

> QScripts is productivity tool and an alternative to IDA's "Recent scripts" (Alt-F9) and "Execute Scripts" (Shift-F2) facilities.

readme for qscripts

What is QScripts?

QScripts is productivity tool and an alternative to IDA's "Recent scripts" (Alt-F9) and "Execute Scripts" (Shift-F2) facilities. QScripts allows you to develop and run any supported scripting language (*.py; *.idc, etc.) from the comfort of your own favorite text editor as soon as you save the active script, the trigger file or any of its dependencies.

Usage

Invoke QScripts from the plugins menu, press Ctrl-3 or its default hotkey Alt-Shift-F9. When it runs, the scripts list might be empty. Just press `Ins` and select a script to add, or press `Del` to delete a script from the list. QScripts shares the same scripts list as IDA's `Recent Scripts` window.

To execute a script, just press `ENTER` or double-click it. After running a script once, it will become the active script (shown in **bold**).

An active script will then be monitored for changes. If you modify the script in your favorite text editor and save it, then QScripts will execute the script for you automatically in IDA.

To deactivate a script, just press `Ctrl-D` or right-click and choose `Deactivate script monitor` from the QScripts window. When an active script becomes inactive, it will be shown in *italics*.

There are few options that can be configured in QScripts. Just press `Ctrl+E` or right-click and select `Options`:

- Clear message window before execution: clear the message log before re-running the script. Very handy if you to have a fresh output log each time.

- Show file name when execution: display the name of the file that is automatically executed
- Execute the unload script function: A special function, if defined, called `__quick_unload_script` will be invoked before reloading the script. This gives your script a chance to do some cleanup (for example to unregister some hotkeys)
- Script monitor interval: controls the refresh rate of the script change monitor. Ideally 500ms is a good amount of time to pick up script changes.
- Allow QScripts execution to be undo-able: The executed script's side effects can be reverted with IDA's Undo

Executing a script without activating it

It is possible to execute a script from QScripts without having to activate it. Just press `Shift-ENTER` on a script and it will be executed.

Working with dependencies

It is possible to instruct QScripts to re-execute the active script if any of its dependent scripts are also modified. To use the automatic dependency system, please create a file named exactly like your active script but with the additional `.deps.qscripts` extension. In that file you put your dependent scripts path.

When using Python, it would be helpful if we can also reload the changed dependent script from the active script automatically. To do that, simply add the directive line `/reload` along with the desired reload syntax. For example, here's a complete `.deps.qscripts` file with a `reload` directive (for Python 2.x):

```
/reload reload($basename$)
t2.py
// This is a comment
t3.py
```

And for Python 3.x:

```
/reload import importlib;importlib.reload($basename$);
t2.py
# This is a comment
t3.py
```

So what happens now if we have an active file `t1.py` with the dependency file above?

1. Any time `t1.py` changes, it will be automatically re-executed in IDA.
2. If the dependency index file `t1.py.deps.qscripts` is changed, then your new dependencies will be reloaded and the active script will be executed again.

3. If any dependency script file has changed, then the active script will re-execute. If you had a `reload` directive set up, then the modified dependency files will also be reloaded.

Please note that if each dependent script file has its own dependency index file, then QScripts will recursively make all the linked dependencies as part of the active script dependencies. In this case, the directives (such as `reload`) are ignored.

See also:

- Simple dependency example
- Package dependency example

Special variables in the dependency index file

- `$basename$` : This variable is expanded to the base name of the current dependency line
- `$env:EnvVariableName$` : `EnvVariableName` is expanded to its environment variable value if it exists or left unexpanded otherwise
- `$pkgbase$` : Specify a package base directory. Can be used as part of a dependency file path.
- `$pkgmodname$` : This is mainly used inside the `reload` directive. It replaces the path seperators with dots (which works nicely as a fully qualified module name). Please see the package dependency example file.

Using QScripts with trigger files

Sometimes you don't want to trigger QScripts when your scripts are saved, instead you want your own trigger condition. One way to achieve a custom trigger is by using the `/triggerfile` directive:

```
/triggerfile createme.tmp

; Dependencies...
dep1.py
```

This tells QScripts to wait until the trigger file `createme.tmp` is created (or modified) before executing your script. Now, any time you want to execute the active script, just create (or modify) the trigger file.

You may pass the `/keep` option so QScripts does not delete your trigger file, for example:

```
/triggerfile /keep dont_del_me.info
```

Using QScripts programmatically

It is possible to invoke QScripts from a script. For instance, in IDAPython, you can execute the last selected script with:

```
load_and_run_plugin("qscripts", 1);
```

(note the run argument `1`)

If the script monitor is deactivated, you can programmatically activate it by running the plugin with argument `2`. To deactivate again, use run argument `3`.

Using QScripts with compiled code

QScripts is not designed to work with compiled code, however using a combination of tricks, we can use QScripts for such cases:

What you just saw was the `hello` sample from the IDA SDK. This plugin has the `PLUGIN_UNL` flag. This flag tells IDA to unload the plugin after each invocation. We can then use the trigger files option and specify the compiled binary path as the trigger file. Additionally, we need to write a simple script that loads and runs that newly compiled plugin in IDA.

First, let's start with the script that we need to activate and run:

```
# Optionally clear the screen:
idaapi.msg_clear()

# Load your plugin and pass any arg value you want
idaapi.load_and_run_plugin('hello', 0)

# Optionally, do post work, etc.
# ...
```

Then let's create the dependency file with the proper trigger file configuration:

```
/triggerfile /keep C:\<ida_dir>\plugins\hello.dll
```

Now, simply use your favorite IDE (or terminal) and build (or rebuild) the `hello` sample plugin.

The moment the compilation succeeds, the new binary will be detected (since it is the trigger file) then your active script will use IDA's `load_and_run_plugin()` to run the plugin again.

Please check the trigger-native example.

Building

QScripts uses idax and is built using ida-cmake .

If you don't want to build from sources, then there are release pre-built for MS Windows.

Installation

QScripts is written in C++ with IDA's SDK and therefore it should be deployed like a regular plugin. Copy the plugin binaries to either of those locations:

- `<IDA_install_folder>/plugins`
- `%APPDATA%\Hex-Rays/plugins`

Since the plugin uses IDA's SDK and no other OS specific functions, the plugin should be compilable for macOS and Linux just fine. I only provide MS Windows binaries. Please check the releases page .

---

nmips - by: Leonardo Galli

Plugin description

> IDA plugin to enable nanoMIPS processor support. This is not limited to simple disassembly, but fully supports decompilation and even fixes up the stack in certain functions using custom microcode optimizers. It also supports relocations and automati

readme for nmips

IDA plugin to enable nanoMIPS processor support. This is not limited to simple disassembly, but fully supports decompilation and even fixes up the stack in certain functions using custom microcode optimizers. It also supports relocations and automatic ELF detection (even though the UI might not show it, it kinda works). Debugging also works thanks to GDB and it also does some other stuff, such as automatic switch detections.

Tested on IDA 7.6.

To see how well it works, with the mipscoder binary from 0CTF, see below :) You can disassemble, decompile and even debug it!

Installation

OS	Download
Linux	Download
macOS (ARM might be broken)	Download

OS	Download
Windows	Download

Download the corresponding version for your OS and put the plugin inside `~/.idapro/plugins` . Done! If you open a nanoMIPS ELF file, you should be able to just mash through some of the dialogs and get it working (yes metaPC should work fine if selected and yes it will show unknown arch, that's an IDA limitation unfortunately :/. Just keep mashing enter and you should be good ;)).

If you want to e.g. apply this to a flat binary file, you can instead just load it as a little endian MIPS file. Then, select this plugin from `Edit > Plugins > nanoMIPS Processor Support` . This will force it on, and it should start to disassemble stuff!

Functionality

Currently, the following works:

- debugging
- creating relocations for libraries (e.g. libc) "
- decompiling and disassembling (not all instructions are currently implemented)
- custom hexrays optimizer to fix stack variables being messed up
- automatic switch statement detection
- more stuff I probably forgot

NOTE: For debugging to work, you need to modify the gdb config file. Fortunately, this plugin can automatically do this for you. Unfortunately, due to a limitation of the gdb plugin, this will override the normal mips configuration. The plugin can automatically remove the changes again. To automatically change the configuration, either use `Edit > Configure GDB for nanoMIPS` or `Ctrl+Shift+Meta+G` .

Implementation

The basic idea behind the plugin is, to still load the binary with the MIPS processor module. The plugin registers a bunch of plugin hooks, so that it can then give IDA the illusion of working on a "normal" MIPS binary. To that end, the binary translates any nanoMIPS instruction into the equivalent MIPS version, or - if it does not exist - implement it itself.

In case the instruction is translated to MIPS, it will be decompiled automatically without any issues (well that is if the operands are correctly set. Quite some instructions have a complex operand encoding inside IDA and don't work out of the box.). Otherwise, decompiler hooks emit the correct hexrays microcode, so that these instructions can also be decompiled correctly.

If you are wondering how most of this was made possible, the answer is simple: A lot of reversing of IDA itself ;). Mostly the GDB and mips plugin, but also libida.

Building

Make sure you have meson installed. Then inside the `plugin` directory, just run:

```
meson setup builddir -Didasdk=$IDA_SDK -Dhexrays_sdk=$IDA_BIN/plugins/hexrays_sdk
meson compile -C builddir
```

TODOs

- implement assembler -> actually not possible atm :/
- fix debugging to be nicer
- rework plugin to be nicer

---

jside - by: David Zimmer

Plugin description

> This plugin comes in two parts. There is a small IPC based server which sits in IDA and allows remote automation, and then there is an external Javascript IDE which supports intellisense, syntax highlighting, and a full on javascript debugger.

readme for jside

- So I finally found some free time this weekend to integrate the duktape debugger activex control I created with the original IDA JScript interface.Only took about 5hrs to get it swapped out and working.

  Select your Version of IDA: 2015: **IDA <= 6.9 Installer** (git repo , IdaSrvr)
- 2020: **IDA 7+ Installer** (git repo , IdaSrvr2)
-

  The old version only works on 32bit disassemblies. The newer version now supports 32 and 64bit disassemblies.

quick overview: script ida using javascript

- full syntax highlighting, brace matching, auto indent, tool tips
- intellisense
- full debugger support, step in/out/over, breakpoints, mouse over variable vals
- debugger cmdline to query/set vals, run functions etc
- small IDA plugin as an IPC server with main UI as running out of process to simplify development.
- currently supports following api at time of writing:
  - **list object:** AddItem Clear ListCount Enabled
  - **fso object:** ReadFile WriteFile AppendFile FileExists DeleteFile OpenFileDialog SaveFileDialog
  - **app object:** die intToHex t ClearLog Caption alert getClipboard setClipboard BenchMark AskValue Exec EnableIDADebugMessages
  - **ida object:** isUp Message MakeStr MakeUnk LoadedFile() PatchString PatchByte GetAsm InstSize XRefsTo XRefsFrom GetName FunctionName HideBlock ShowBlock Setname AddComment GetComment AddCodeXRef AddDataXRef DelCodeXRef DelDataXRef FuncVAByName RenameFunc Find Decompile Jump JumpRVA refresh Undefine ShowEA HideEA RemoveName MakeCode FuncIndexFromVA NextEA PrevEA funcCount() NumFuncs() FunctionStart FunctionEnd ReadByte OriginalByte ImageBase() ScreenEA() QuickCall
  - **remote object:** ScanProcess ResolveExport ip response
  - **x64 object:** add subtract toHex
-

Credits: Duktape - http://duktape.org

- Scintilla by Neil Hodgson [neilh@scintilla.org] - site: http://www.scintilla.org/
- ScintillaVB by Stu Collier - site: http://www.ceditmx.com/software/scintilla-vb/
- CSubclass by Paul Canton [Paul_Caton@hotmail.com]
- Interface by David Zimmer - site http://sandsprite.com
-

IPyIDA - by: Marc-Etienne M.Léveillé

Plugin description

> IPyIDA is a python-only solution to add an IPython console to IDA Pro. Use Shift-. to open a window with an embedded Qt console. You can then benefit from IPython's autocompletion, online help, monospaced font input field, graphs, and so on. You can

readme for IPyIDA

IPyIDA is a python-only solution to add an IPython console to IDA Pro. Use `<Shift-.>` to open a window with an embedded *Qt console* . You can then benefit from IPython's autocompletion, online help, monospaced font input field, graphs, and so on.

You can also connect to the kernel outside of IDA using `ipython console --existing` .

Install

IPyIDA has been tested with IDA 6.6 and up on Windows, OS X and Linux, up to 7.6.

Fast and easy install

A script is provided to install IPyIDA and its dependencies automagically from the IDA console. Simply copy the following line to the IDA console.

Python 2

```
import urllib2; exec urllib2.urlopen('https://github.com/eset/ipyida/raw/stable/install_fro
```

Python 3

```
import urllib.request; exec(urllib.request.urlopen('https://github.com/eset/ipyida/raw/stabl
```

On macOS, `Python3.framework` does not provide a trusted CA file. You can use the system-wide file `/etc/ssl/cert.pem` .

```
import urllib.request; exec(urllib.request.urlopen('https://github.com/eset/ipyida/raw/stabl
```

The script will do the following:

1. Install `pip` if it's not already installed
2. Install the `ipyida` package from PyPi
3. Copy `ipyida_plugin_stub.py` to the user's `plugins` directory

4. Load the IPyIDA plugin

You can inspect the install_from_ida.py script if you wish to see exactly what it does.

Warning	Don't panic. It's normal to see Windows' command prompt window open during the installation on Windows.

Upgrading Rerun the install script to update to the latest version and restart IDA.

Install it the IDA way

If you'd rather manage the dependencies and updates yourself, you can copy `ipyida_plugin_stub.py` and the `ipyida` directory inside IDA's `plugins` directory.

This method requires that you manage dependencies on your own. IPyIDA requires the `ipykernel` and `qtconsole` package, and the `qasync` package if using `ipykernel` version 5 or newer.

Customizing the IPython console

By default, the console does not have any globals available. If you want to have module imported before the console is opened, IPyIDA will read the `ipyidarc.py` file from the IDA user directory (`idaapi.get_user_idadir()`). Anything you import in this file will be available in the IPython console.

This is similar to the `idapythonrc.py` file.

Dark mode

With a dark theme in IDA Pro, it's more convenient to also have a dark theme in the IPython console. To activate the dark theme built into `qtconsole` , add the following in your `ipyidarc.py` :

```
import qtconsole.styles
import ipyida.ida_qtconsole
ipyida.ida_qtconsole.set_widget_options(dict(
    style_sheet = qtconsole.styles.default_dark_style_sheet,
    syntax_style = qtconsole.styles.default_dark_syntax_style
))
```

Caveats

Notebook not working

IPython notebook cannot attach to an existing kernel like `ipython console` and `ipython qtconsole` do. There's some more background info here: ipython/ipython#4066 .

There are workarounds to this be I didn't include one yet. I will consider adding it if it's not too hackish. Pull requests are welcome.

External console not responding if IDA's window is in the background for too long

This problem is specific to OS X's App Nap feature in OS X 10.9 and up. When running Disabling App Nap for IDA Pro seems to fix the problem, but is a a hack for now and will use more battery on a laptop, we should find a better solution.

Here is how to disable App Nap for IDA Pro:

```
defaults write com.hexrays.idaq   NSAppSleepDisabled -bool YES
defaults write com.hexrays.idaq64 NSAppSleepDisabled -bool YES
```

Similar work

@james91b also successfully integrated a IPython kernel in IDA. Being a non-Windows IDA user, I needed something cross-platform and took a Python-only approach to implement it. Although the some of the implementation idea comes from that project, IPyIDA was written from scratch. Hat tip to @james91b and all IDA_IPython contributors. IDA_IPython is available on Github at https://github.com/james91b/ida_ipython .

IDAPatternSearch - by: David Lazar

Plugin description

> The IDA Pattern Search plugin adds a capability of finding functions according to bit-patterns into the well-known IDA Pro disassembler based on Ghidra's function patterns format. Using this plugin, it is possible to define new patterns according to

readme for IDAPatternSearch

IDA Pattern Search

by Argus Cyber Security Ltd.

The *IDA Pattern Search* plugin adds a capability of finding functions according to bit-patterns into the well-known IDA Pro disassembler based on Ghidra's function patterns format. Using this plugin, it is possible to define new patterns according to the appropriate CPU architecture and analyze the target binary to find and define new functions in it.

For more detailed information, including Ghidra's format for bit-patterns and how to generate new patterns, check out our blog post about this plugin.

Usage

1. Place all repo files under the IDA plugins folder (i.e. <IDA installation dir>\plugins).
2. Start IDA and load the desired program with the appropriate CPU architecture.
3. From the menu, choose: Edit → Plugins → IDA Pattern Search.
4. In case you want the plugin to search for function prologues in all possible undefined code, choose Yes in the displayed message box. However, if you want the plugin to search in specific address ranges or segments, choose No and specify in the next textbox the desired address ranges or segments (format is explained below).

Note that the plugin will identify the CPU architecture and find functions according to the CPU architecture matching patterns residing in the *function_patterns* folder.

Currently, the supported architectures are ARM/THUMB, AARCH64, PPC, v850. More can be added easily and how to do it is explained below.

It should be noted that the current version uses only post-patterns, as we find those more effective than pre-patterns. While the functionality to use pre-patterns exists in our code, it is currently disabled.

How to define the target addresses that the plugin will work on?

Target addresses can be specified in two non-exclusive ways using a python dictionary:

1. Address range(s) - either a tuple or a list of tuples specifying a start address and an exclusive end address, passed in the address_ranges argument. Passing an empty list would result in including NO addresses.

   For example:

   ```
   {"address_ranges":[(0, 0x1000), (0xFFFF0000, 0xFFFFFFFF)]}
   ```

```
{"address_ranges":(0, 0xFFFFFFFF)}
```

2. Segment name(s) - either a segment name or a list of segment names, passed in the segment's argument. Passing an empty list would include all arguments in the IDB.

   For example:

   ```
   {"segments":[".text", ".bss"]}
   {"segments":".text"}
   {"segments":[]}
   ```

- You can also include both, for example:
  `{"address_ranges":(0x0, 0xFFFFFFFF),"segments":[".text"]}`

How to add new CPU architecture?

Simply add the patterns file in the function_patterns directory. This pattern file can be simply taken from Ghidra or created from scratch.

Then, add the matching parameters to the _ *SEARCH_PARAMETERS* dictionary defined in the code.

Finally, add to the function *explore_using_patterns* a code that handles the added CPU architecture and calls *parse_and_search* function with the newly added _ *SEARCH_PARAMETERS* dictionary entry as function arguments.

IDA2Obj - by: Mickey Jin

Plugin description

> IDA2Obj is a tool to implement SBI (Static Binary Instrumentation).

readme for IDA2Obj

IDA2Obj is a tool to implement SBI (Static Binary Instrumentation) .

The working flow is simple:

- Dump object files (COFF) directly from one executable binary .
- Link the object files into a new binary, almost the same as the old one.
- During the dumping process, you can insert any data/code at any location .
  - SBI is just one of the using scenarios, especially useful for black-box fuzzing.

How to use

0. Prepare the enviroment:
   - Set `AUTOIMPORT_COMPAT_IDA695 = YES` in the `idapython.cfg` to support the API with old IDA 6.x style.
   - Install dependency: `pip install cough`
1. Create a folder as the workspace.
2. Copy the target binary which you want to fuzz into the workspace.
3. Load the binary into IDA Pro, choose Load resources and manually load to load all the segments from the binary.
4. Wait for the auto-analysis done.
5. Dump object files by running the script `MagicIDA/main.py`.
   - The output object files will be inside `${workspace}/${module}/objs/afl`.
   - If you create an empty file named `TRACE_MODE` inside the workspace, then the output object files will be inside `${workspace}/${module}/objs/trace`.
   - By the way, it will also generate 3 files inside `${workspace}/${module}`:
     - exports_afl.def (used for linking)
     - exports_trace.def (used for linking)
     - hint.txt (used for patching)
6. Generate lib files by running the script `utils/LibImports.py`.
   - The output lib files will be inside `${workspace}/${module}/libs`, used for linking later.
7. Open a terminal and change the directory to the workspace.
8. Link all the object files and lib files by using `utils/link.bat`.
   - e.g. `utils/link.bat GdiPlus dll afl /RELEASE`
   - It will generate the new binary with the pdb file inside `${workspace}/${module}`.
9. Patch the new built binary by using `utils/PatchPEHeader.py`.
   - e.g. `utils/PatchPEHeader.py GdiPlus/GdiPlus.afl.dll`
   - For the first time, you may need to run `utils/register_msdia_run_as_administrator.ba` as administrator.

10. Run & Fuzz.

More details

HITB Slides : https://github.com/jhftss/jhftss.github.io/blob/main/res/slides/HITB2021SIN%20-%20IDA2Obj%20-%20Mickey%20Jin.pdf

Demo : https://drive.google.com/file/d/1N3DXJCts5jG0Y5B92CrJOTIHedWyEQKr/view?usp=sharing

---

FunctionInliner - by: Tomer Harpaz

Plugin description

> FunctionInliner is an IDA plugin that can be used to ease the reversing of binaries that have been space-optimized with function outlining (e.g. clang -moutline).

readme for FunctionInliner

FunctionInliner

FunctionInliner is an IDA plugin that can be used to ease the reversing of binaries that have been space-optimized with function outlining (e.g. `clang --moutline`).

Our plugin works by creating a clone of the outlined function per each of its xref callers, and linking it to the caller directly by replacing the BL to with a regular branch to the clone and the RET of the clone with a branch back to the caller (thus adding the clone as a function chunk of its caller).

In case the an outlined function has been succesfully inlined into all of its callers, it'll be renamed to have the `inlined_` prefix, to make it easy to identify in functions/xrefs listing.

The plugin supports both manually choosing functions to inline from their context menu, and heuristically identifying all outlined functions and inlining them.

Why?

Code with outlined functions is a pain to reverse because the outlined functions usually use registers and memory that is local to their *caller* (i.e. they don't conform to the ABI). Therefore you don't have the entire context when reversing

the caller and have to jump back and forth into those outlined parts to follow what's going on.

Moreover, reversing code with outlined functions using Hex Rays simply doesn't work since Hex Rays assumes that functions conform to the ABI in order to do its magic. Moreover, if you'll try to jump into the outlined function in Hex Rays you'll often see them as empty because of that.

Example

As an example we used gzip 1.3.5 which is a single source file that was easy to work with, and we looked at the beginning of a single function from it (`bi_windup`):

On the left, you see the function compiled with `-O3` and in the middle you see it compiled with `-O3 -moutline` . Calls to outlined functions were highlighted (obviously, these wouldn't have stood out from other calls in case symbols have been stripped).

We've also marked some screwups in Hex Rays' decompilation that were caused by these outlined functions not conforming to the ABI.

On the right, you see the same function after our whole-IDB analysis has been applied. You can see that most outlined functions have been automatically inlined, and all decompilation screwups have been resolved.

To be exact, in this file our whole-IDB analysis found and automayically inlined 130 out of 165 outlined functions, with no false positives. The rest of the outlined functions can be easily inlined from their context menu in case they're manually identified later.

Specifically in this example, you can also see that `OUTLINED_FUNCTION_13` (which was not automatically inlined) is a simple wrapper to `write` which specifies `nbytes = 0x4000` . In this case we could never determine whether this was an original wrapper function or an outlined function that we should inline.

Installation

1. Install the dependencies listed in `requirements.txt` where IDA can import them. For example using `/path/to/python3/used/by/ida -m pip install -r requirements.txt` .
2. Install keypatch .
3. Clone this repository and symlink `~/.idapro/plugins/functioninliner.py` to `functioninliner.py` in the cloned repo.

Usage TL;DR

From the menu select `Edit -> Plugins -> FunctionInliner -> Patch constant register-based calls to regular calls` and then `Edit -> Plugin -> FunctionInliner -> Inline all outlined functions` in order to try and do everything we can to make the IDB more readable.

Per-function usage

Note: all of the context menus described below work both in IDA views and in Psuedocode views.

Inlining outlined functions

Right-click on a `BL` to an outlined function, or on the beginning of an outlined function and choose `Inline function` (or use the keyboard shortcut `Meta-P`, i.e. `Cmd/WinKey-P`).

Note that the cloning logic does not support functions which consist of multiple function chunks. For such cases, you should dechunk the function manually, or have it done automatically by running our whole-IDB processing.

Undoing inlining of outlined functions

Right-click on a `B` to the cloned code that was originally outlined, on the begining of the cloned code, or on the beginning of the original outlined function and choose `Undo function inlining`

Whole-IDB usage

The plugin also supports working on the entire IDB and inlining *every* function that is identified as an outlined function. See the `Principals of operation` section for the heuristics used to identify these.

Inlining all outlined functions

From the menu select `Edit -> Plugins -> FunctionInliner -> Inline all outlined functions` in order to scan all of the functions in the binary and inline those who are identified as outlined.

Note that we first do some preprocessing on the entire IDB in order to fix various situations that may have occured from IDA auto-analyzing the IDB without taking outlined functions into consideration.

Patching constant register-based calls

In some cases the compiler and linker generate register-based calls for constant addresses (and not regular calls). IDA obviously doesn't generate call xrefs in these cases (but data xrefs) and so our inlining logic cannot patch these calls.

From the menu select `Edit -> Plugins -> FunctionInliner -> Patch constant register-based calls to regular calls` in order to scan all of the IDB for these patterns and patch them to regular calls.

Since this behaviour is actively patching the IDB we kept it as a separate (optional) action, and do not do this as part of the `Inline all outlined functions` preprocessing logic.

Principals of operation

What preprocessing is done prior to inlining all outlined functions

Our preprocessing is comprised of a number of steps:

1. Exploration steps are repeated until there's nothing new to be done:
    i. We create functions at xref targets that IDA didn't make a function out of.
    ii. We identify NORET functions that IDA didn't identify as such.
2. Preprocessing steps are done afterward the exploration:
    i. We dechunks all of the functions in the binary (split each chunk into a separate function). This helps us identify later on which chunks were outlined and which are "real" functions. Plus, our cloning logic doesn't support chunked functions.
    ii. We split functions that are placed right before another function they tail-call into, and were treated by IDA as one whole function.
    iii. We split adjacent functions that were treated by IDA as one whole function.

How cloning is done

For each xref to the outlined function, we create a new segment named `inlined_0x{func_ea:x}_for_0x{src_ea:x}` and clone the function there.

When cloning, we in fact have to translate some of the opcodes on the way -- if an opcode has relative data or code xrefs we need to fix them to work from the new location. We also may have to fix relative xrefs inside the cloned code because our translation may move stuff around in the clone as well.

We then replace the original `BL` to the outlined function with a `B` to the cloned code, and replace the `RET` in the end of the outlined function with a `B` back to the caller.

There are of course edge cases when the outlined function tail-calls some other function, or when the outlined function is tail-called by its caller, which should be handled.

We also take care to find a spot for the cloned code segment which will be close enough to the caller and to outgoing xrefs from the clone in order to use regular branches back and forth.

How outlined functions are identified

Currently we use a few heuristics to identify outlined functions.

There may be false-negatives (i.e. we may miss some outlined functions) but we expect their count to be pretty low and they can always be inlined manually when encountered.

Also, in case there will be any false-positives (i.e. we'll identify some real functions as outlined and inline them into their callers) the effect shouldn't be that bad for RE and can also be undone manually.

The heuristics we use are the following:

1. ~~We expect outlined functions to have more than one caller (otherwise it wouldn't have been useful to outline them)~~ for some reason this doesn't hold in real cases, so we've dropped this heuristic.
2. We expect all outlined functions not to have a prologue (not really sure about that, but it makes sense). This is more of an optimization for us, in order not to statically analyze *every* function in the IDB.
3. We expect some outlined functions not to conform to the ABI and to make use of non-argument registers that were not initialized internally.
4. We expect some outlined functions not to conform to the ABI and to leave side-effects on non-result registers (that are not propagated to any result register or stored in memory).
5. The last two heuristics also hold for condition flags and not registers (i.e. if the function is using uninitializing/setting unused condition flags).

Future improvements

There are some cases of outlined functions that we currently don't auto detect with our heuristics:

1. Some outlined functions leave side-effects on higher result registers but do not set all of the lower ones, so it's obvious that they're not just returning a structure by value.
2. Consider removing the heuristic about outlined functions not having prologues, since we've seen cases of outlined prologues.

There are some cases which our cloning logic doesn't support:

1. Some uses of conditional opcodes (e.g. when the call to the outlined function is a conditional tail-call).

Some other stuff:

1. When running our heuristics, we currently stop analyzing a function if we encounter a BL or a tail-call in it, because that may lead to another outlined function. The proper handling should probably be to analyze and inline all functions in order of a topological sort based on call targets (i.e.

first analyze and inline functions which don't call anything, then those who call already analyzed functions, and so on).
2. When running our heuristics, we usually stop analyzing a function if we encounter more than one basic block. We could theoretically continue analyzing recursively in each of the branches.

Limitations

The plugin currently works only on ARM64 binaries that conform to the ABI .

Fixing corrupted state

If for any reason (e.g. IDA crashed in the middle of function inlining) the IDB has gotten into corrupted state with regards to FunctionInliner, where for example:

1. You have deleted but not disabled addresses that were once inlined clones.
2. You have inlined clones with unpatched source calls.
3. You have patched source calls with missing clones.
4. FunctionInliner thinks it has already inlined (or undid inlining of) something which it hasn't.

You should run the `fix_state.py` script in the context of the corrupted IDB.

In our testing, this happened once after months of heavy usage, and we suspect another conflicting plugin to cause this, so we didn't bother integrating the fixing logic into the plugin.

Meta

Authored by Tomer Harpaz of Cellebrite Security Research Labs. Developed and tested for IDA 7.6 on macOS with Python 3.7.9.

D-810 - by: Boris Batteux

Plugin description

> D-810 is a plugin which aims at removing several obfuscation layer (including MBA, opaque predicate and control flow flattening) during decompilation. It relies on the Hex-Rays microcode API to perform optimization during the decompilation. The plugi

readme for D-810

Introduction

What is D-810

D-810 is an IDA Pro plugin which can be used to deobfuscate code at decompilation time by modifying IDA Pro microcode. It was designed with the following goals in mind:

- It should have as least as possible impact on our standard reverse engineering workflow
  - Fully integrated to IDA Pro
- It should be easily extensible and configurable
  - Fast creation of new deobfuscation rules
  - Configurable so that we don't have to modify the source code to use rules for a specific project
- Performance impact should be reasonable
  - Our goal is to be transparent for the reverse engineer
  - But we don't care if the decompilation of a function takes 1 more second if the resulting code is much more simplier.

Installation

Only IDA v7.5 or later is supported with Python 3.7 and higher (since we need the microcode Python API)

Copy this repository in .idapro/plugins

We recommend to install Z3 to be able to use several features of D-810:

```
pip3 install z3-solver
```

Using D-810

- Load the plugin by using the Ctrl-Shift-D shortcut, you should see this configuration GUI
- Choose or create your project configuration
  - If you are not sure what to do here, leave default_instruction_only.json.
- Click on the Start button to enable deobfuscation
- Decompile an obfuscated function, the code should be simplified (hopefully)
- When you want to disable deobfuscation, just click on the Stop button.

Warnings

This plugin is still in early stage of development, so issues ~~may~~ will happen.

- Modifying incorrectly IDA microcode may lead IDA to crash. We try to detect that as much as possible to avoid crash, but since it may still happen save you IDA database often
- We only tested this plugin on Linux, but it should work on Windows too.

Documentation

Work in progress

Currently, you can read our blog post to get some information.

Licenses

Authors

See AUTHORS for the list of contributors to the project.

Acknowledgement

Rolf Rolles for the huge work he has done with his HexRaysDeob plugin and all the information about Hex-Rays microcode internals described in his blog post . We are still using some part of his plugin in D-810.

Dennis Elser for the genmc plugin plugin which was very helpful for debugging D-810 errors.

CTO - by: Hiroshi Suzuki

Plugin description

CTO (Call Tree Overviewer) is an IDA Pro plugin for visualizing function call tree. It can also summarize function information such as internal function calls, API calls, static linked library function calls, unresolved function calls, string referen

readme for CTO

CTO (Call Tree Overviewer)

CTO (Call Tree Overviewer) is an IDA plugin for creating a simple and efficiant function call tree graph. It can also summarize function information such as internal function calls, API calls, static linked library function calls, unresolved indirect function calls, string references, structure member accesses, specific comments.

CTO has another helper plugin named "CTO Function Lister", although it can work as a standalone tool. You can think this is an enhanced version of functions window. It lists functions with summarized important information, which is the same as the CTO's one. You can use a regex filter to find nodes with a specific pattern as well.

An introduction video is here.
https://youtu.be/zVCpb82UfFs

You can also check the presentation at VB2021 localhost.
https://vblocalhost.com/conference/presentations/cto-call-tree-overviewer-yet-another-function-call-tree-viewer/

Submitted paper
https://vblocalhost.com/uploads/VB2021-Suzuki.pdf

Presentation slides
https://vblocalhost.com/uploads/2021/09/VB2021-14.pdf

Requirements

- IDA Pro 7.4 or later (I tested on 7.5 SP3 and 7.6 SP1)
- Python 3.x (I tested on Python 3.8 and 3.9)

You will need at least IDA Pro 7.4 or later because of the APIs that I use. And use Python 3.x. It should work on Python 2.7 but I did not test enough and I do not support it because it has already obsoleted and deprecated.

Optional 3rd Party Software

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

- findcrypt.py
  https://github.com/you0708/ida/tree/master/idapython_tools/findcrypt
- findguid.py
  https://github.com/you0708/ida/tree/master/idapython_tools/findguid
- IDA_Signsrch
  https://sourceforge.net/projects/idasignsrch/
- SusanRTTI
  https://github.com/nccgroup/SusanRTTI

How to Install

See " INSTALL " file.

How to Use

To start CTO, press Alt+Shift+C.

Double-click "..." symbol if you want to expand the path. If you want to create a graph based on a different target function, jump to the target function, click the CTO window, and press "F" key. See the help by pressing "H" key on the CTO window.

To start CTO Function Lister, press Alt+Shift+F. See the help by pressing "H" key on the CTO Function Lister window as well.

Note

CTO is still under development and it is unstable yet. I might change the data structure drastically. CTO accesses sensitive internal data structure of IDA such as low level APIs and PyQt5. And it might cause a crash of IDA. Do not use this in important situations. I don't take responsibility for any damage or any loss caused by the use of this.

I'm not a programmer. I'm a malware analyst. Please do not expect product-level code.

PRs are welcome. Just complaining and a bug report without enough information are NOT welcome ;-)

Known Issues

- CTO Function Lister will crash on IDA on Linux for some reasons while it works on Windows. But I can't fix it because I don't have that.

```
QSortFilterProxyModel: index from wrong model passed to mapToSource
```

- Currently, CTO focuses on Intel x64/x86 architecture. If you want to extend other architectures, please send the PR to me.
- On IDA 7.6 including SP1, you will not be able to use ESC for looking backward location history on CTO's window because of a bug of IDA. Instead, it will close the CTO window if you press it. I reported the bug and it was fixed internally but not released yet. If you want to use it, you will need a fixed ida*.exe binary. Ask hex-rays support. Please do not ask me.

CollaRE - by: Martin Petran

Plugin description

> CollaRE enables collaboration using multiple reverse engineering tools for more complex projects where one tool cant provide all required features or where each member of the team prefers a different tool to do the job.

readme for CollaRE

Introduction

CollaRE is a tool for collaborative reverse engineering that aims to allow teams that do need to use more then one tool during a project to collaborate without the need to share the files on a separate location. It also contains a very simple user management and as such can be used for a multi-project server where different teams work on different projects. The back-end of the tool is a simple `Flask app` with `nginx` in front of it running in Docker that works with files and JSON based manifests that hold the relevant data. The front-end is a PyQT based GUI tool with a simple interface that allows managing the projects and working with the binary files and their corresponding reverse engineering databases. As of now the tool supports `Binary Ninja`, `Cutter (Rizin)`, `Ghidra`, `Hopper Dissassembler`, `IDA`, `JEB` and `Android Studio (Decompiled by JADX)`. The implementation is abstracted from the inner workings of these tools as much as possible to avoid issues with any API changes and thus does not integrate directly into those tools in form of a plugin (except for data migration plugins described below). The work is based purely on managing the files produced by these tools (literally just based on the well known file extensions) and simple SVN-style `check-out` and `check-in` operations.

Installation

Grab the latest binary release from this repository or clone the repo and run `sudo python3 setup.py install` on Linux or use command line on Windows and run `python3 setup.py install`. On Linux this will install the tool to the `PATH` and you will be able to run it simply with `collare` command. On Windows this will put the file into the `C:\Users\<USERNAME>\AppData\Local\Programs\Python\<PYTHON_VERSION>\Scripts\collare.exe` (depending on how you installed Python).

For Gnome based desktop UIs you can use following desktop file (paths to files may vary based on version of CollaRE and Python):

```
[Desktop Entry]
Type=Application
Encoding=UTF-8
Name=CollaRE
Exec=/usr/local/bin/collare
Icon=/usr/local/lib/python3.8/dist-packages/collare-1.2-py3.8.egg/collare/icons/collare.png
Terminal=false
```

Supported Tools

Cutter (Rizin)

To enable support for Cutter add a file `Cutter` to your path (when you open `cmd` / `terminal` writing `Cutter` should start the application). When saving Cutter (rizin) projects you have to manually append `.rzdb`. Do not remove the extension that the file already has (`exe` or `so` for example).

Binary Ninja

To enable support for Binary Ninja add a file `binaryninja` to your path (when you open `cmd` / `terminal` writing `binaryninja` should start the application). Binary Ninja is removing file extensions by default, however the tool accounts for this so there is no need to put the original file extension back manually. Saving the projects as is in a default path is enough to be able to successfully push local `bndb` database.

Hopper Disassembler

To enable support for Hopper Disassembler add a file `Hopper` to your path (when you open `cmd` / `terminal` writing `Hopper` should start the application). Hopper is removing file extensions by default, however the tool accounts for this so there is no need to put the original file extension back manually. Saving the projects simply with `Ctrl+S` is enough to be able to successfully push local `hop` database.

JEB

To enable support for JEB add a file `jeb` to your path (when you open `cmd` / `terminal` writing `jeb` should start the application). This can be done by renaming the default runner script file for your OS to `jeb` (for Windows this would actually be `jeb.bat`).

IDA Pro

To enable support for IDA tool add a files `ida64` and `ida` to your path (when you open `cmd` / `terminal` writing `ida64` / `ida` should start the application).

Ghidra

To enable support for this tool add a file `ghidraRun` and `analyzeHeadless` (`.bat` for Windows) to your path (when you open `cmd` / `terminal` writing `ghidraRun` should start the application). Note that `analyzeHeadless` is in `support` folder in the Ghidra root directory so make sure to adjust `PATH` to accommodate both files. The process of initializing the database with Ghidra is a bit more complicated as there is no way that Ghidra will process file without creating a project . So to be able to push the Ghidra database (referred to as `ghdb`) you will be prompted to create a project manually whenever automatic processing fails (basically whenever the file you process is not ELF/PE) and then specify the path to the `gpr` file (sorry for that).

Android Studio

As APK and JAR files are often encountered during the reverse engineering efforts the CollaRE tool also supports working with these types of files. To enable support for these tools it is necessary to make sure that files `android-studio` and `jadx` are both in the path (when you open `cmd` / `terminal` writing `android-studio` / `jadx` should start the application). The JADX tool is used to perform the decompilation of the JAR/APK file and the Android Studio is used to open the resulting files. Note that use of Android Studio is optional as you can alias any other tool that handles Gradle projects under `android-studio` command (such as IntelliJ IDEA).

Usage

After deploying the server side as mentioned in its own readme file, it is necessary to distribute the used certificate file to all users of the application as well as use the default `admin` account with `admin` password to create other user accounts (don't forget to change password of `admin` user) via the

`Admin` tab. When the users are configured anyone can create their own projects and start working with the tool itself.

Creating projects

To create a project user has to first authenticate to the remote server by entering the URL, credentials and provide a certificate to validate the server identity. After that, the status will change to `Connected` and it is possible to select or delete existing projects or create a new project by simply entering the name (alphanumeric characters and `_` only) and selecting users that will be participating on the project (can be changed later in the `Admin` tab). Note that the user that is creating the project is automatically added to the user list so you do not have to select yourself.

Project Structure and File Uploads

Once you are on the `Project View` tab you can create new folders (alphanumeric characters and `_` only, sorry) and use drag and drop to upload files (or folders).

Pushing Local DB Files

Since the tool currently does not have any plugins or native hooks that would allow automatic uploads when the project is saved it is required that the local DB file push is triggered manually after creating the desired databases. This can be done by right-clicking on the uploaded binary file and choosing the tool you want to process the binary in. You can do basic analysis but it is strongly recommended to just save the file without changing anything (apart from appending `rzdb` in Cutter and completely different process with Ghidra). DO NOT CHANGE THE PATH AND FILENAME . After doing this and closing the disassembler you can just right click on the binary name and select option `Push Local DBs` . This will upload the local database and from now on when you want to work with the DB file you need to perform `Check-out` . Note that each binary can be processed in all the tools separately but only one DB file per binary and tool can exist.

Working with DB Files

When you just want to inspect the file you can right-click the desired DB file and select option `Open File` (or just double-click). If the file is checked-out to you this will open the local file and you can freely perform any changes to the DB file. When done (or when you simply want to push the changes) you can select the `Check-in` option. This will upload the changes to the server and prompt you whether you want to keep the file checked-out for further changes. If you want to discard your local changes select the `Undo Check-out` option from the context menu. This will discard your changes and allow you to continue with the file from the server. Opening a file without

doing a `Check-out` operation first will open it in a fake read-only mode (you can do changes to the DB file but those will be lost next time you check-out or open the file).

Versioning

The tool also supports versioning the DB files in a way that every `Check-in` action counts as a new version of the DB file. You will be prompted to insert a comment for the version which is used to give more context to the changes that are applied in that version. It is then possible to open or check-out the previous versions of the files and work on those.

Plugins

The plugins folder within this repository contains plugins for the supported tools which allow you to share comments and function names between the tools in case that you work on one binary with multiple tools. Follow the standard plugin installation instructions for the tool you are interested in. Each plugin offers an `Import` and an `Export` function. When you plan to share the data between the tools always make sure that you `Import` data first to avoid renaming functions that were already renamed by someone else. If the plugin comes with some catches, those are mentioned in the README file of the given plugin. Note that the plugins are intended to migrate the data to other tool rather then for a simultaneous collaboration of multiple people.
