This work is licensed under a Creative Commons license

Attribution-NonCommercial-NoDerivatives 4.0 International
(CC BY-NC-ND 4.0)

You are free to:

Share copy and redistribute the material in any medium or format.

Under the following terms:

Attribution You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial You may not use the material for commercial purposes.

NoDerivatives If you remix, transform, or build upon the material, you may not distribute the modified material.

Advanced Ghidra

Scripting with Python

Giovanni Lagorio

giovanni.lagorio@unige.it
https://csec.it/people/giovanni_lagorio
Twitter & GitHub: zxgio

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy

www.zenhack.it

Outline

Scripting

- Java via Eclipse
- Jython (AKA Python 2) via Eclipse+PyDev
- Python 3
  - Ghidra Bridge
    `https://github.com/justfoxing/ghidra_bridge`
  - Ghidrathon
    `https://github.com/mandiant/Ghidrathon`

Hello world

- Script Manager
- `Examples/HelloWorldScript.java`
- `Run/Rerun`
- `Edit`
- `In-Tool/Key`

Metadata:

`@category` Category path; levels are separated by "`.`"

`@keybinding` `case sensitive; e.g., L, or ctrl alt shift F12 or ctrl shift COMMA`

`@menupath` `Level dot-separated, see next slide`

`@toolbar` `icon for the toolbar, searches in script directories and then Ghidra installation directory`

Hello world from Python

- New script → Python

```
#Prints hello in the Console
#@author me
#@category Examples.Python
#@menupath Tools.Hi from Py

from __future__ import print_function
import sys

print("Hello from Python", sys.version)
```

APIs

- Ghidra Program API
  - OO
  - huge, can change from version to version
- Script API
  - flat
  - includes most common features
  - stable

`docs/api/ghidra/program/flatapi/FlatProgramAPI.html`

(you need to unzip `docs/GhidraAPI_javadoc.zip`)

Global state

Useful functions

- `currentProgram()` the current active open program
  - type: `ProgramDB`, see `docs/api/ghidra/program/database/ProgramDB.html`
  - many `get...` methods; e.g. `getExecutablePath()`
- `currentLocation()` the program location of the cursor
  - a `ProgramLocation`, the `currentLocation().getAddress()` is...
- `currentAddress()` the current address of the location of the cursor
  - this is an `Address`, in an address space
  - usually you just need the offset: `getOffset()`
  - `toAddr(`*`offset`*`)` returns an `Address`; e.g., `goTo(toAddr(0x1234))`
  - given an address, you can "move" with `next()`/`previous()`/`add(`*`n`*`)`/`subtract(`*`n`*`)`
- `currentSelection()` the current selection or None
  - a `ProgramSelection`, which allows us to iterate over `AddressRanges`, `getMinAddress()`, `getMaxAddress()`, ...

Input/Output

- `print` and `popup`
- `ask...`
  - `askString(title, msg)`
  - `askYesNo(title, msg)` — returns a boolean, `True` for yes
  - `askAddress(title, msg)`
  - `askInt(title, msg)`
  - `askFile(title, approve_btn_text)`
  - `askDirectory(title, approve_btn_text)`

  they raise a `CancelledException` when the user cancels

Monitor

Long operations should check `monitor().isCancelled()`

Memory

Reading

- `get{Byte,Int,Long}(addr)`
  - return *signed* values
  - to convert to unsigned, you can use: `... & 0xff`, `... & (2**32-1)` and `... & (2**64-1)`
- `getBytes(addr,len)`
  - returns a `jep.PyJArray`; you can use `bytes(b & 0xff for b in getBytes(...))` to get Python `bytes`

Writing

- `set{Byte,Int,Long}(addr, value)`
- `setBytes(addr, bytes)`

to change/edit existing data, you need to `clearListing` first

Demo/exercises

→ `bulz_6a5210`

1. Complete `read_c_string`, in `bulz_1.py`, to read a C-string returned as Python `bytes`/`bytearray`, from a starting address `addr`
   - hint: create a `bytearray()`, and then `append` bytes to it
   - `read_c_string(toAddr(0x10002124))` should return a path
   - remember to check `monitor().isCancelled()` in your loop
     - return `None` if true
2. Let's start reversing from `D`; can you spot a string decryption routine?
   - Reverse engineer the decryption routine; how does it work?
   - Complete `bulz_2.py` by implementing the decryption routine; then, check the result of the script

Code units (=Data and Instructions)

- Data
  - `getFirstData()`, ...
- Instructions
  - `getFirstInstruction()`, `getInstructionAt(`*addr*`)`, `getInstructionAfter(`*inst*`)`, ...
  - members: `getNumOperands()`, `getOpObjects(`*idx*`)`, `getPrevious()`/`getNext()`, ...

  `getOpObjects` returns `Address`, `Scalar`, `Register`, ...

all Code units have `toString()`, and `get...` functions

- `MinAddress`, `MaxAddress`, `Bytes`, `Length`, `MnemonicString`, `PrimarySymbol`, ...

Symbols and References

- `Symbol`, lookup:
  - `getSymbolAt(`*`addr`*`)`

  members:
  - `getName()`
  - `getAddress()`
- `Reference`, lookup:
  - `getReferencesFrom(`*`addr`*`)`
  - `getReferencesTo(`*`addr`*`)`

  members:
  - `getFromAddress()`
  - `getToAddress()`
  - `getReferenceType()`, a `RefType`
    - `isCall()`, `isRead()`, ...

Demo/exercises

→ `bulz_6a5210` (again)

Let's find all calls to `0x10001210`; complete `bulz_3.py` so that it:

1. gets all references to such an address, by implementing `get_references_to_decrypt_function`
   - use function `getReferencesTo`, then
   - filter out any reference that is not a call, if any (see `getReferenceType` and `isCall`)
2. prints, in hex, the address where each call takes place

Functions

`FunctionDB` lookup:

- `getFirstFunction()`, ...(as before)
- `getGlobalFunctions(`*`name`*`)`
- `currentProgram().`
  `getFunctionManager().`
  `getFunctionAt(`*`addr`*`)`

members:

- `getEntryPoint()`

Label, Bookmarks and Comments

- `createLabel(address, name, makePrimary)`
  - `makePrimary` is a boolean; the *primary* label is the one used to represent the address everywhere it is displayed
- `createBookmark(address, category, note)`
  - `category` can be `None`
- `setPreComment(addr, comment)`
- `setEOLComment(addr, comment)`
- ...

Demo/exercises

→ `bulz_6a5210` (again)

Complete `bulz_4.py`, a script to comment all calls to the decryption routine. The script can use:

- your `get_references_to_decrypt_function` to get the call sites
- `extract_constant_args_of_call` to extract the parameters
  - this is non-trivial, we provide it for you; in order to work, this function needs information generated by the *Decompiler Parameter ID* analysis
- your `decrypt_str` to get the decoded bytes
  - you can convert them to a "real" string by using `.decode('latin1')`
- `setEOLComment`/`setPreComment` to set a comment at each call site

The Headless Analyzer

A command-line version of Ghidra that allows users to:

- Create and populate projects
- Perform analysis on imported or existing binaries
- Run non-GUI scripts in a project

Useful to perform repetitive tasks on a project

See `support/analyzeHeadlessREADME.html` for all details

Demo/exercises

→ `rbreaker_24596b`

1. What is the function called most?
2. What does it do?
   - complete `rbreaker-cleanup.py` to get a cleaner version, by NOPping the "garbage"
   - this seems *extremely* similar to something we already reverse engineered; can you spot the similarity?
   - spoiler (ROT13)
     - hfr ohym_2 gb qrpbqr gur pnyy ng nqqerff 0k401020
     - nqncg ohym_4 gb guvf arj fnzcyr (eha vg ba gur bevtvany fnzcyr, abg gur bar pyrnarq hc ol bhe fpevcg)
   - such a function is used to dynamically load/find new code; can we rename the global variables used to store the function pointers? Complete rbreaker-labeler.py

Demo/exercises

→ `mumble_a64e7e`

- `entry` contains something fishy
- let's focus on `FUN_08048106`
  1. what is it? Does Ghidra detect the arguments correctly?
  2. clean-up the code, then re-implement it in function `mumble_decrypt` of `mumble_1.py`
  3. print the result of `mumble_decrypt(0x080482f9, 0xd)`
  4. are there any other interesting strings?

Outline

Emulators

Emulators allow us to “statically execute” code

- without actually running it
- fully controlled/customizable execution

Ghidra Emulator

- Ghidra has always exposed its own emulator via the class `ghidra.app.emulator`
- A third-party, open-source GUI was developed: GhidraEmu `https://github.com/Nalen98/GhidraEmu`
- Since version `10.3`, official GUI; see, for instance, `https://medium.com/@cy1337/first-look-ghidras-10-3-emulator-7f74dd55e12d`

Unicorn

Unicorn is a lightweight CPU emulator framework

- Multi-architectures: ARM, ARM64 (ARMv8), m68k, MIPS, PowerPC, RISC-V, S390x (SystemZ), SPARC, TriCore & x86 (include x86_64)
- Clean/simple/lightweight/intuitive architecture-neutral API
- Implemented in pure C language, with bindings for Pharo, Crystal, Clojure, Visual Basic, Perl, Rust, Haskell, Ruby, Python, Java, Go, D, Lua, JavaScript, .NET, Delphi/Pascal & MSVC available
- Native support for Windows & *nix
- High performance by using Just-In-Time compiler technique
- Support fine-grained instrumentation at various levels
- Thread-safe by design
- Distributed under free software license GPLv2

`https://www.unicorn-engine.org/`

Get started

1. import `Uc` and constants; e.g.

```
from unicorn import Uc, UC_ARCH_X86, UC_MODE_64, ...
from unicorn.x86_const import UC_X86_REG_RSP, ...
```

2. create an emulator instance; e.g.

```
emu = Uc(UC_ARCH_X86, UC_MODE_64)
```

3. map/populate *pages* (i.e., for x86 the mapping length must be a multiple of 4k) and set registers; e.g.

```
emu.mem_map(CODE_START, 4096, UC_PROT_READ | UC_PROT_EXEC)
emu.mem_write(CODE_START, code) # type(code)==bytes

emu.mem_map(STACK_ADDR, STACK_SIZE, UC_PROT_READ |
                                        UC_PROT_WRITE)
emu.reg_write(UC_X86_REG_RSP, STACK_ADDR + STACK_SIZE)
```

Optionally, set hooks

4. you can set various hooks; e.g. `UC_CODE`:

```
def hook_code(emu, address, size, user_data):
    if address in [...]: # skip instructions
        emu.reg_write(UC_X86_REG_RIP, address + size)
    elif address == 0x1234: # jump to 0x5678
        emu.reg_write(UC_X86_REG_RIP, 0x5678)

emu.hook_add(UC_HOOK_CODE, hook_code [, user_data])
```

You can hook basic-blocks, special instructions, memory accesses, . . . see: https://github.com/unicorn-engine/unicorn/blob/master/bindings/python/sample_x86.py

Run and profit!

5. run and read from the final state; e.g.

```
emu.emu_start(0x1234, 0x5678) # run code in the range
                              # [0x1234, 0x5678); i.e.,
                              # does not try to execute
                              # instruction @ 0x5678

emu.mem_read(0xabcd, 1234) # reads 1234 bytes
                           # at address 0xabcd

rax = emu.reg_read(UC_X86_REG_RAX)
```

To trace the execution you can use a hook:

```
def hook_code(emu, address, size, user_data):
        print(f'Instruction at 0x{address:x}, size = {size}')
```

Related projects

Dumpulator a Python library for emulating code in minidump files
`https://github.com/mrexodia/dumpulator`

- On Linux, see `unicorn-emulate` from GEF-extra

Speakeasy a portable, modular, binary emulator designed to emulate Windows kernel and user mode malware
`https://github.com/mandiant/speakeasy`

Demo/exercises

→ `mumble_a64e7e` (again, this time using Unicorn)

1. create an emulator object for x86 in 32-bit mode
   - you can find the starting code in `mumble_2.py`
2. map all program memory (see the *Memory Map* view) as `UC_PROT_ALL`
3. create a stack and set up register `ESP`
4. emulate from `0x804807e` to `0x80480a1`, skipping instruction at `0x804808d`
5. print memory content at:
   - `0x080482f9`, length `0xd`
   - `0x08048307`, length `0x1706`

Further exercises

→ `caddy_e5c03b`

This sample uses *stack-strings*. Write a Python script to emulate the selected instructions and then extract all strings from stack memory

- when `currentSelection()` is not `None`, you can use a `getMinAddress()` and `getMaxAddress()`
- the last selected address can be obtained by using `getInstructionContaining(`*`addr`*`)`
- ASCII/Unicode strings can be found by re-using `https://gist.github.com/jedimasterbot/39ef35bc4324e4b4338a210298526cd0`

A solution/generalization of this idea is:
`https://github.com/zxgio/ghidra_stack_strings`