

Reverse Engineering and Code Emulation with Ghidra

Karl 'Rosie' Sickendick - @kc0bfv

October 2019

Outline

Introduction to Ghidra

PCode Emulation with Python

Ghidra Scripts via Python

Section 0

Who Am I?



Who Am I?

- ▶ Karl 'Rosie' Sickendick - @kc0bfv
- ▶ Active duty Major in the Air Force
- ▶ Stationed at Idaho National Labs as part of an Air Force fellowship program
- ▶ BS in EE, MS in CS
- ▶ Experience with: cyber capability and tactics development
- ▶ Very happy to be here!



Section 1

Introduction to Ghidra

Ghidra Intro

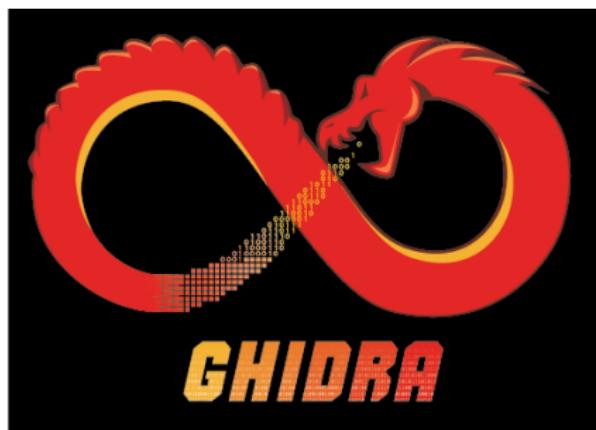
- ▶ Comprehensive software reverse engineering tool
 - ▶ NSA-produced, open source
 - ▶ Your tax dollars at work!
- ▶ Features a decompiler that supports many architectures
 - ▶ Sleigh: understands the architecture, bytes -> PCode
 - ▶ PCode: architecture-agnostic intermediate language
 - ▶ Decompiler: does data-flow analysis on the PCode
- ▶ "Easily" extensible to new architectures
 - ▶ Build a Sleigh module, map bytes -> instructions -> PCode
 - ▶ Get the decompiler for free



Ghidra Intro

► Setup

- ▶ Install OpenJDK 11
- ▶ Download zip: <https://ghidra-sre.org/>
- ▶ Unzip somewhere you can write
- ▶ Make sure Java is in your PATH...
- ▶ 'Dual monitors strongly suggested'...
- ▶ GHIDRA_INSTALL/ghidra_version/ghidraRun



Ghidra Organization

- ▶ Local
 - ▶ Non-Shared Projects
 - ▶ Folders
 - ▶ Executables
 - ▶ Libraries
 - ▶ History
 - ▶ Version Control
 - ▶ Data Types/Structures/Classes
 - ▶ Server
 - ▶ Shared Projects
 - ▶ ... Same as Non-Shared ...
 - ▶ Check-out/Check-in
 - ▶ Change Merging
 - ▶ File Systems
 - ▶ File Systems
 - ▶ ...
 - ▶ Import Files

Importing Files & Supported Types

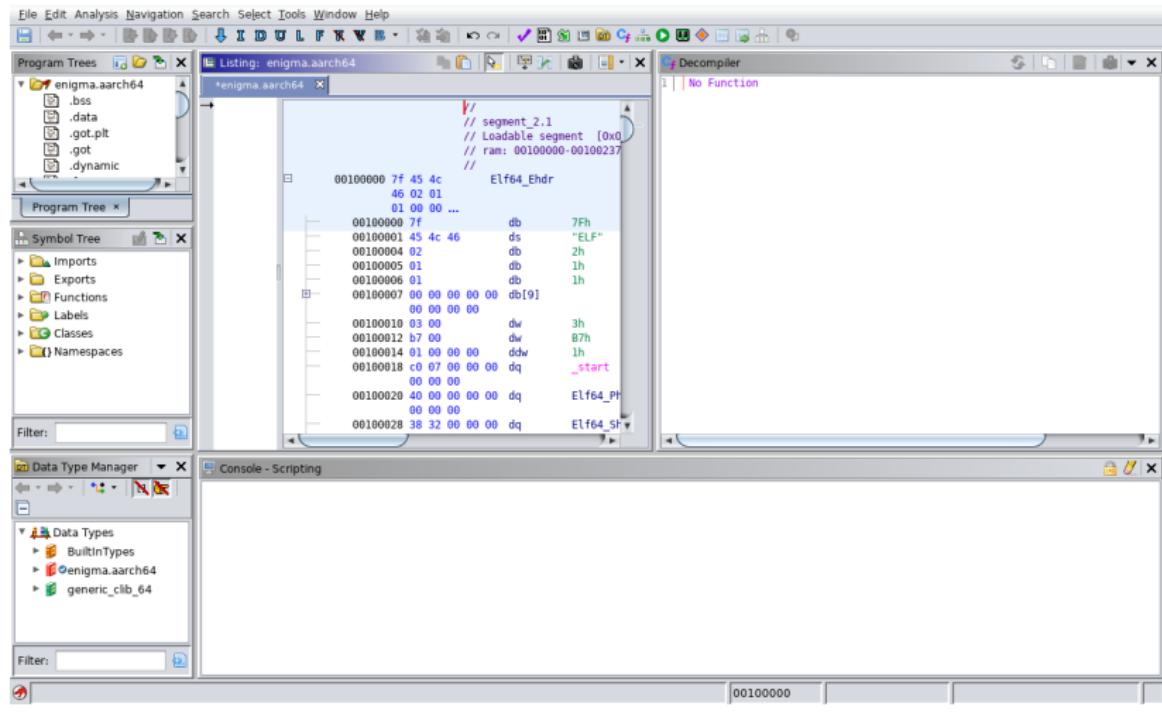
- ▶ 6502 (NES? Atari?)
- ▶ Motorola: 6805, 68000
- ▶ Intel: 8048, 8051, 8085, 80251
- ▶ ARM & AARCH64 {big,little} endian
- ▶ AVR{8,32}
- ▶ JVM
- ▶ MIPS {32,64} bit {big,little} endian
- ▶ PIC-{12,16,17,18,24}
- ▶ PowerPC {32,64} bit {big,little} endian
- ▶ Sparc
- ▶ TI_MSP430
- ▶ x86 {16,32,64} bit
- ▶ Z{80,180}

Importing Files & Supported Types

The screenshot shows a software application window with the following components:

- Menu Bar:** File, Edit, Project, Tools, Help.
- Tool Chest:** A panel on the left containing icons for a green dragon, a blue hand, and a circular arrow.
- Active Project:** TemporaryProject
- File Browser:** A tree view showing a folder named "TemporaryProject" containing a file named "test.x86".
- Filter:** A search bar at the bottom of the file browser panel.
- View Options:** Tree View and Table View buttons.
- Running Tools:** A panel at the bottom labeled "Workspace" with a dropdown arrow.
- Status Bar:** A message indicating a file system operation: "File system file:///home/finity/VM_CDs/TENS-1.7.5_public.iso?MD5=712f8dd9bdf65b48fdeac6f3a67f899c|iso9660:// was c...".
- Bottom Navigation:** A row of small navigation icons.

Starting Analysis



Starting Analysis

Analyzers

Enabled	Analyzer Name
<input type="checkbox"/>	Aggressive Instruction Finder (Prototype)
<input checked="" type="checkbox"/>	Apply Data Archives
<input checked="" type="checkbox"/>	ASCII Strings
<input checked="" type="checkbox"/>	Call Convention Identification
<input checked="" type="checkbox"/>	Call-Fixup Installer
<input type="checkbox"/>	Condense Filler Bytes (Prototype)
<input checked="" type="checkbox"/>	Create Address Tables
<input checked="" type="checkbox"/>	Data Reference
<input type="checkbox"/>	Decompiler Parameter ID
<input checked="" type="checkbox"/>	Decompiler Switch Analysis

Description

Options

Buttons

Select All Deselect All Restore Defaults

Analyze Cancel

Symbol Tree

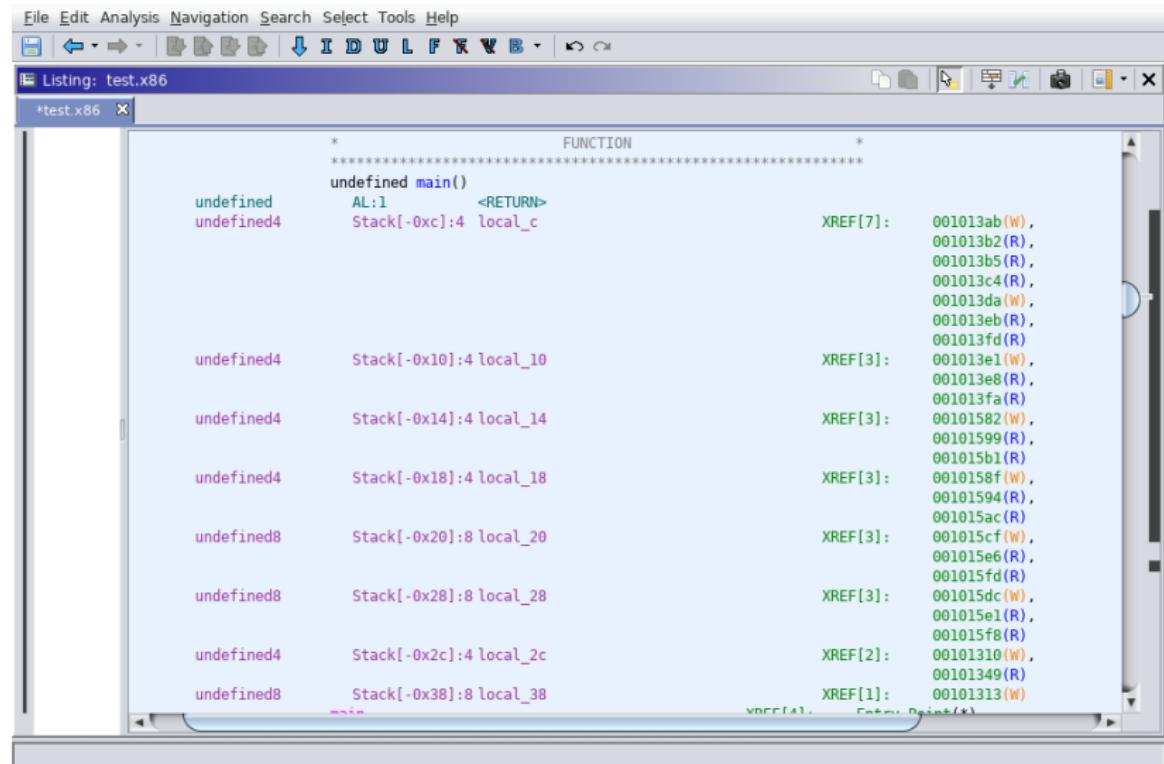
The screenshot shows the Immunity Debugger interface with several windows open:

- Program Tree (test.x86):** Shows the file structure of the test.x86 binary.
- Symbol Tree:** A tree view of symbols, including:
 - Imports
 - Exports
 - Functions
 - Labels
 - Classes
 - Namespaces
- Listing - test.x86:** Displays assembly code:

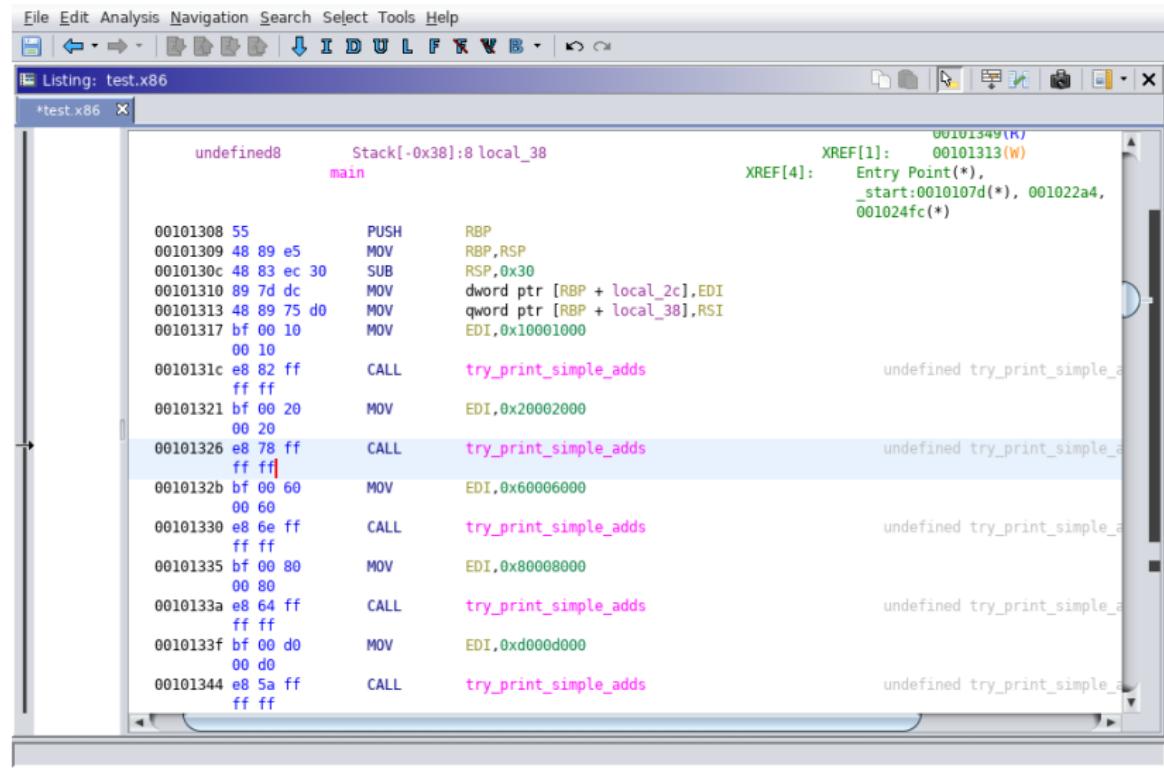
```
// segment_2.1
// Loadable segment [0x0 - 0x5b7]
// ram: 00100000-001002a7
//
assume DF = 0x0 (Default)
00100000 7f 45 4c      Elf64_Ehdr
    46 02 01
    01 00 00 ...
00100000 ff            db    7fh
00100001 45 4c 46      ds    "ELF"
00100004 02            db    2h
00100005 01            db    1h
00100006 01            db    1h
00100007 00 00 00 00 dq[9]          e_ident_mag...
00100008 00 00 00       dw    3h
00100009 00 00 00       dw    3ch
00100014 01 00 00 00 ddw   1h
00100018 00 20 00 00 dq[1]          _start
00100029 40 00 00 00 dq[1]          Elf64_Phdr_ARRAY_00100...
0010002a 68 30 00 00 dq[1]          Elf64_Shdr_ARRAY_elfs...
00100030 00 00 00 00 ddw   0h
00100034 40 00           dw    40h
00100036 38 00           dw    38h
00100038 0b 00           dw    8h
    ...
```
- Decompile main - (test.x86):** Displays C decompiled code:

```
1 undefined8 main(int param_1)
2
3
4 {
5     uint uVar1;
6     undefined8 uVar2;
7     div_t dVar3;
8     float extraout_0040_0a;
9     undefined8 extraout_0040_0a;
10
11     try_print_simple_addr(0x10001000);
12     try_print_simple_addr(0x20002000);
13     try_print_simple_addr(0x60006000);
14     try_print_simple_addr(0xd000d000);
15     try_print_simple_addr(0xb000b000);
16     uVar2 = update_many_regs((long)param_1);
17     printf("Update many regs result %bx\n",
18         uVar2 - read_rpl());
19     printf("RIP value %bx\n",uVar2);
20     uVar2 = another_sub(0x10,0x40);
21     printf("Another sub %bx\n",uVar2);
22     uVar2 = investigate_add_flags(0x7fffffff,
23         printf("Investigate add flags 0x%08x\n");
24     uVar2 = investigate_sub_flags(0,0x80000000);
25     printf("Investigate sub flags 0x%08x\n");
26     uVar1 = big_sub(1);
27     printf("Big sub 1 %in\n", (ulong)uVar1);
28     uVar1 = big_sub(0);
29     printf("Big sub 0 %in\n", (ulong)uVar1);
30     uVar1 = big_sub(0xffffffff);
31     printf("Big sub -1 %in\n", (ulong)uVar1);
```
- Console - Scripting:** An empty console window.

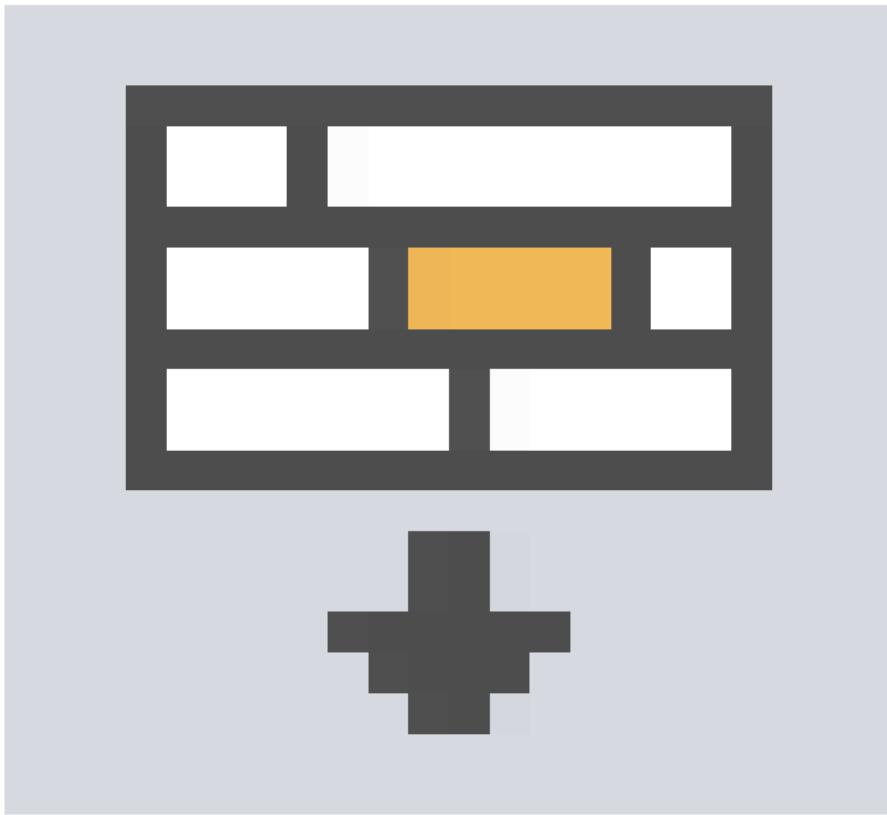
Program Listing



Program Listing



Showing PCode



Showing PCode

File Edit Analysis Navigation Search Select Tools Help

Listing: test.x86

*test.x86

Address Break Plate Function Variable Instruction/Data Open Data Array

Register Transition		Pre-Comment		Label XRef Header XRef	
+	Address	Bytes	... Mnemonic	Operands	EOL Comment
					PCode
					Post-Comment
			Space		
			main	XREF[4]:	Entry Point(*), _start:0010107d(*), 001022 001024fc(*)
00101308	55	PUSH	RBP	\$U2510:8 = COPY RBP RSP = INT_SUB RSP, 8:8 STORE ram(RSP), \$U2510	
00101309	48 89 e5	MOV	RBP,RSP	RBP = COPY RSP	
0010130c	48 83 ec 30	SUB	RSP,0x30	CF = INT_LESS RSP, 48:8 OF = INT_SBORROW RSP, 48:8 RSP = INT_SUB RSP, 48:8 SF = INT_SLESS RSP, 0:8 ZF = INT_EQUAL RSP, 0:8	
00101310	89 7d dc	MOV	dword ptr [RBP + local_2c],EDI	\$U620:8 = INT_ADD RBP, -36:8 \$U1fd0:4 = COPY EDI STORE ram(\$U620), \$U1fd0	
00101312	48 89 75 10	MUL			

Navigation icons: back, forward, search, etc.

PCode Explanation

PUSH

RBP

\$U2510:8 = COPY RBP
RSP = INT_SUB RSP, 8:8
STORE ram(RSP), \$U2510

PCode Explanation

File Edit Analysis Navigation Search Select Tools Help

I D U L F K V B

Listing: test.aarch64.out

*test.aarch64.out X

Address Break Plate Function Variable Instruction/Data Open Data Array

Register Transition			
		Pre-Comment	XRef Header
		Label	
Address	Bytes	... Mnemonic	Operands
PCode			
Post-Comment			
Space			
main		XREF[5]: Entry Point(* _start:00100E 00101190(*),	
001009ac	fd 7b bb a9	stp	x29,x30,[sp, #local_50]! sp = INT_ADD sp, 0xfffffffffffffb0:8 STORE ram(sp), x29 \$U71e0:8 = INT_ADD sp, 8:8 STORE ram(\$U71e0), x30
001009b0	fd 03 00 91	mov	x29,sp x29 = COPY sp
001009b4	e8 27 01 6d	stp	d8,d9,[sp, #local_40] \$Uf90:8 = INT_ADD sp, 16:8 STORE ram(\$Uf90), d8 \$U18d60:8 = INT_ADD \$Uf90, 8:8 STORE ram(\$U18d60), d9
001009b8	e0 2f 00 b9	str	w0,[sp, #local_24] \$Uc70:8 = INT_ADD sp, 44:8 STORE ram(\$Uc70), w0
001009bc	e1 13 00 f9	str	x1,[sp, #local_30] \$Uc90:8 = INT_ADD sp, 32:8 STORE ram(\$Uc90), x1

Decompilation

The screenshot shows a debugger interface with two main panes. The left pane displays assembly code from a file named `test.x86.stripped.out`. The right pane shows the corresponding decompiled pseudocode.

Assembly Listing:

```
*test.x86 stripped.out
0000000000401030 31f 7e 1b    JLE    LAB_0010133c
0000000000401031 321 48 8b 45 c0  MOV    RAX,qword ptr [RBP + local_48]
0000000000401032 325 48 8b 00 00  MOV    RAX,qword ptr [RAX]
0000000000401033 328 48 89 c6  MOV    RSI,RAX
0000000000401034 32b 48 8d 3d f1 0c 00 00 LEA    RDI,[s_Prog_name:_$00102023]
0000000000401035 332 b8 00 00 00 00  MOV    EAX,0x0
0000000000401036 337 e8 04 fd ff ff CALL   printf
0000000000401037 33c 48 8b 45 c0  MOV    RAX,qword ptr [RBP + local_48]
```

Decompiled Pseudocode:

```
1 undefined8 FUN_00101308(int param_1,long *param_2,long *param_
2
3 {
4     long *plVar1;
5     uint uVar2;
6     undefined8 uVar3;
7     div_t dVar4;
8     float extraout_XMM0_Da;
9     undefined8 extraout_XMM0_Qa;
10    long *local_10;
11
12    local_10 = param_2;
13    if (0 < param_1) {
14        printf("Prog name: %s\n",*param_2);
15    }
16    while (plVar1 - param_3, *local_10 != 0) {
17        printf("Arg: %s\n",*local_10);
18        local_10 = local_10 + 1;
19    }
20    while (local_10 - plVar1, *local_10 != 0) {
21        printf("Env: %s\n",*local_10);
22        plVar1 = local_10 + 1;
23    }
24
25    FUN_001012a3(0x10001000);
26    FUN_001012a3(0x20002000);
27    FUN_001012a3(0x60006000);
28    FUN_001012a3(0x80008000);
29    FUN_001012a3(0xd000d000);
30    uVar3 = FUN_00101233((long)param_1);
31    printf("Update many regs result 0x%lx\n",uVar3);
32    uVar3 = FUN_00101216();
```

Improving Decompilation - Function Signatures

```
int main (int argc, char ** argv, char ** envp)
```

Function Name:

Calling Convention:

Function Attributes:

Varargs In Line
 No Return Use Custom Storage

Function Variables

Index	Datatype	Name	Storage
1	int	<RETURN>	EAX:4
2	int	argc	EDI:4
3	char **	argv	RSI:8
4	char **	envp	RDX:8

Call Fix...
-NONE- ▾

OK **Cancel**

Improving Decompilation - Structures

The screenshot shows the Immunity Debugger interface with several windows open:

- Program Tree (test.x86):** Shows the file structure with sections like .bss, .data, .got.plt, .got, .dynamic, .fini_array, and .init_array.
- Listing (test.x86):** Displays the assembly listing for the test.x86 file. The assembly code includes comments indicating segments and memory locations. It shows the ELF header and various section headers (e.g., .text, .data, .bss) with their respective addresses and contents.
- Symbol Tree:** Shows the symbol tree for the program, including Imports, Exports, Functions, Labels, Classes, and Namespaces.
- Data Type Manager:** Shows the data type manager for the test.x86 file, listing built-in types and generic structures.
- Console - Scripting:** A command-line interface for running scripts or commands.
- decompile main - (test.x86):** A window showing the decompiled C-like code for the main function. The code includes variable declarations (uVar1, uVar2, dVar3), function calls (try_print_simple_adds), and printf statements. It also contains comments related to memory analysis and flag investigation.

```
1 undefined8 main(int param_1)
2
3
4 {
5     uint uVar1;
6     undefined8 uVar2;
7     div_t dVar3;
8     float extraout_3940_0x0;
9     undefined8 extraout_3940_0x0;
10
11    try_print_simple_adds(0x10001000);
12    try_print_simple_adds(0x20002000);
13    try_print_simple_adds(0x60006000);
14    try_print_simple_adds(0xd000d000);
15    try_print_simple_adds(0xb000b000);
16    uVar2 = update_many_regs((long)param_1);
17    printf("Update many regs result %ox\n", uVar2);
18    uVar2 = read_rpt();
19    printf("RPT value %ox\n", uVar2);
20    uVar2 = another_sub(0x10, 0x40);
21    printf("Another sub %ox\n", uVar2);
22    uVar2 = investigate_add_flags(0xffffffff);
23    printf("Investigate add flags 0x%08x\n");
24    uVar2 = investigate_sub_flags(0, 0x80000000);
25    printf("Investigate sub Flags 0x%08x\n");
26    uVar1 = big_sub(1);
27    printf("Big sub 1 %ox\n", (ulong)uVar1);
28    uVar1 = big_sub(0);
29    printf("Big sub 0 %ox\n", (ulong)uVar1);
30    uVar1 = big_sub(0xfffffff);
31    printf("Big sub -1 %ox\n", (ulong)uVar1);
```

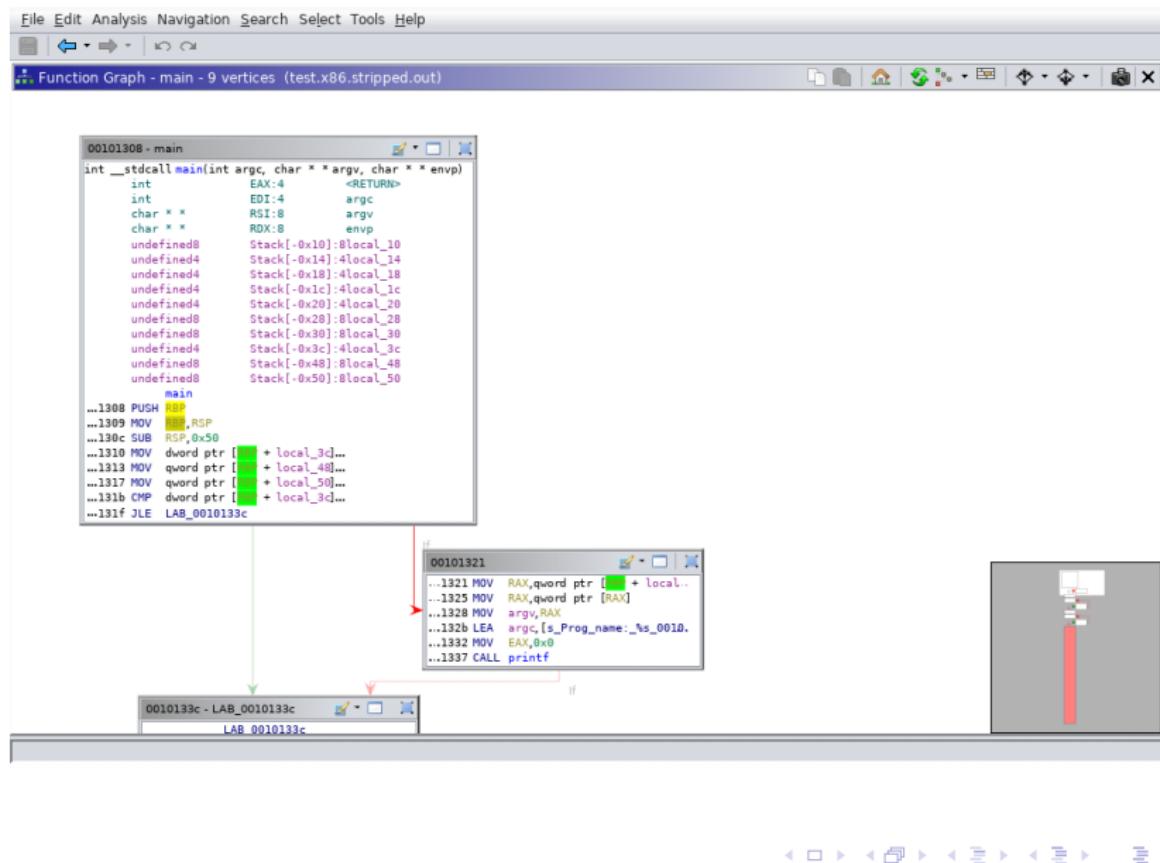
Improving Decompilation - Structures

The screenshot shows the CnC decompiler interface with the title "CnC Decompile: FUN_00101318 - (test.x86.stripped.out W...)" at the top. The main window displays the following C-like pseudocode:

```
1 ulong FUN_00101318(__time_t param_1, long param_2)
2 {
3     timespec local_38;
4     timespec local_28;
5     uint local_10;
6     int local_c;
7
8     local_38.tv_sec = 0;
9     local_38.tv_nsec = 0;
10    local_c = -1;
11    local_10 = 0;
12    local_28.tv_sec = param_1;
13    local_28.tv_nsec = param_2;
14    while (local_c != 0) {
15        local_10 = local_10 + 1;
16        local_c = nanosleep(&local_28, &local_38);
17        local_28.tv_sec = local_38.tv_sec;
18        local_28.tv_nsec = local_38.tv_nsec;
19    }
20    return (ulong)local_10;
21 }
22 }
```

The code uses color-coded syntax highlighting: blue for keywords like `ulong`, `while`, and `return`, green for numbers like `0` and `1`, yellow for variable names like `local_38` and `param_1`, and purple for function names like `nanosleep`. The interface includes standard window controls and a toolbar with icons for file operations.

Function Graphs



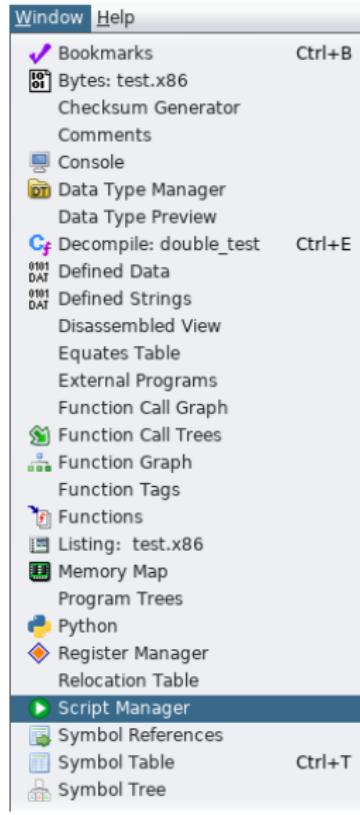
Other Features

- ▶ Undo!
- ▶ Middle clicking an entity to highlight all instances
- ▶ Binary view
- ▶ Highlight interdependence as forward and backward slices
- ▶ Function Call Trees
- ▶ C++ classes
- ▶ Linking to other binaries
- ▶ Server mode
- ▶ Scripts

UNDO!



Scripts



Scripts

Help

Script Manager - 242 scripts

In Tool	Status	Name	Description	Key	Category	Modified
		AddCommentToProgramScript.java	Adds a comment to a program. Di...		Examples	09/23/2019
		AddCommentToProgramScriptPy.py	With cursor on switch's "add pc,		Examples->...	09/23/2019
		AddReferencesInSwitchTable.java	With a user-inputted base address, ...		ARM	09/23/2019
		AddSingleReferenceInSwitchTable....			ARM	09/23/2019
		AppleSingleDoubleScript.java	Given a raw binary Apple Single/D...		Binary	09/23/2019
		ArmThumbFunctionTableScript.java	Makes functions out of a run of sel...		ARM	09/23/2019
		AsciiToBinaryScript.java	Converts an ascii hex file into binar...		Conversion	09/23/2019
		AskScript.java	An example of asking for user input...		Examples	09/23/2019
		AskScriptPy.py	An example of asking for user input...		Examples->...	09/23/2019
		AssembleBlockScript.java	Assemble hard-coded block of inst...		Assembly	09/23/2019
		AssembleCheckDevScript.java	Test assembly of the instruction u...	Ctrl-H	Assembly	09/23/2019
		AssembleScript.java	Assemble a single instruction, over...		Assembly	09/23/2019
		AssemblyThrasherDevScript.java	Thoroughly test the assembler by ...		Assembly	09/23/2019
		AutoRenameLabelsScript.java	Renames default labels in a selecte...		Symbol	09/23/2019
		AutoRenameSimpleLabels.java	A ghidra script that renames simpl...		Symbol	09/23/2019
		AutoVersionTrackingScript.java	An example of how to create Versi...		Examples->...	09/23/2019
		BadInstructionCleanup.java	This script cleans up the disassem...		iOS	09/23/2019
		BatchRename.java	Recursively finds a folder that mat...		Project	09/23/2019
		BatchSegregate64bit.java	Separates co-mingled n-bit and 64...		Project	09/23/2019
		BinaryToAsciiScript.java	Converts a binary file into an ascii ...		Conversion	09/23/2019
		BTreeAnnotationsScript.java	Annotates an HFS+ attributes b-Tr...		iOS	09/23/2019
		BuildFuncDB.java			CodeAnalysis	09/23/2019
		BuildGhidrajarScript.java	An example of building a single mi...		Examples	09/23/2019
		CallAnotherScript.java	Example of a script calling another...		Examples->...	09/23/2019
		CallAnotherScriptForAllPrograms.java	Shows how to run a script on all of...		Examples	09/23/2019
		CallAnotherScriptForAllProgramsPy...	Shows how to run a script on all of...		Examples->...	09/23/2019

Scripts

Help

Script Manager - 5 scripts (of 242)

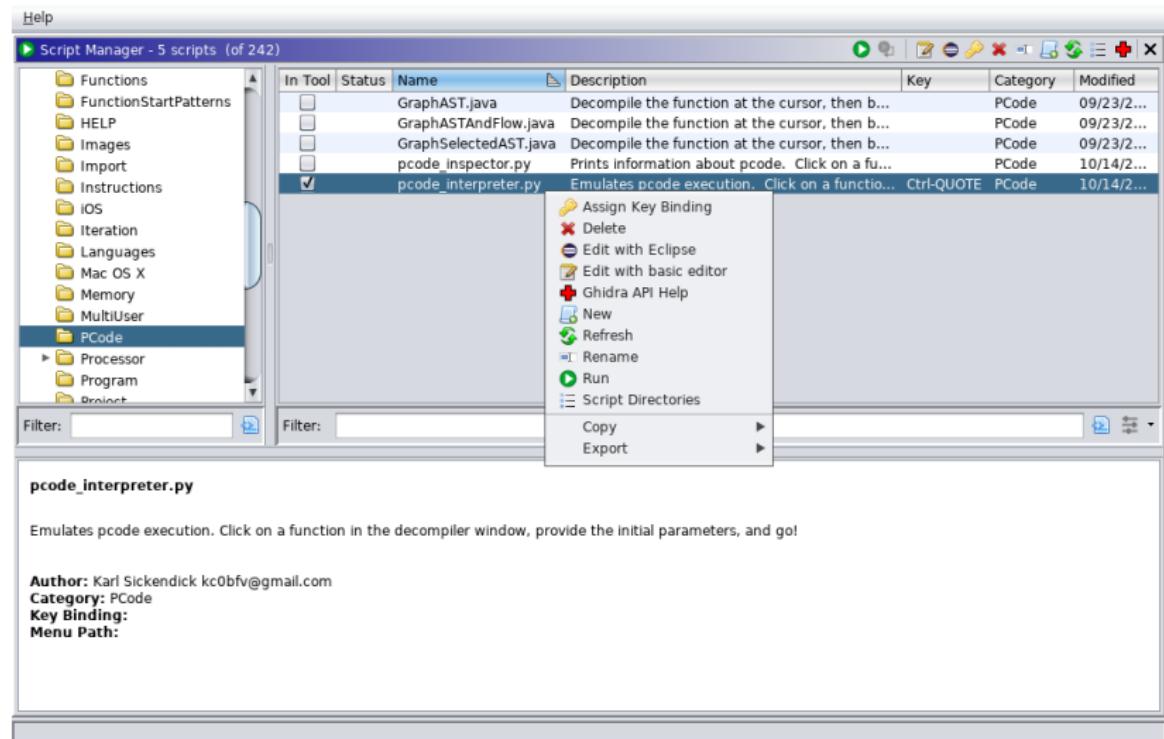
In Tool	Status	Name	Description	Key	Category	Modified
<input type="checkbox"/>		GraphAST.java	Decompile the function at the cursor, then b...		PCode	09/23/2...
<input type="checkbox"/>		GraphASTAndFlow.java	Decompile the function at the cursor, then b...		PCode	09/23/2...
<input type="checkbox"/>		GraphSelectedAST.java	Decompile the function at the cursor, then b...		PCode	09/23/2...
<input type="checkbox"/>		pcode_inspector.py	Prints information about pcode. Click on a fu...		PCode	10/14/2...
<input checked="" type="checkbox"/>		pcode_interpreter.py	Emulates pcode execution. Click on a functio...	Ctrl-QUOTE	PCode	10/14/2...

Filter: Filter:

pcode_interpreter.py

Emulates pcode execution. Click on a function in the decompiler window, provide the initial parameters, and go!

Author: Karl Sickendick kc0bfv@gmail.com
Category: PCode
Key Binding:
Menu Path:



The screenshot shows the Ghidra Script Manager window. On the left is a tree view of script categories. The 'PCode' category is selected, and its sub-items 'Processor', 'Program', and 'Project' are visible. The main pane displays a table of scripts with columns for In Tool, Status, Name, Description, Key, Category, and Modified. The 'pcode_interpreter.py' script is selected, indicated by a checkmark in the 'In Tool' column. A context menu is open over this script, listing options like Assign Key Binding, Delete, Edit with Eclipse, Edit with basic editor, Ghidra API Help, New, Refresh, Rename, Run, and Script Directories. Below the table, there are fields for Filter and another Filter, and buttons for Copy and Export. At the bottom, there's descriptive text about the pcode_interpreter.py script and its key binding information.

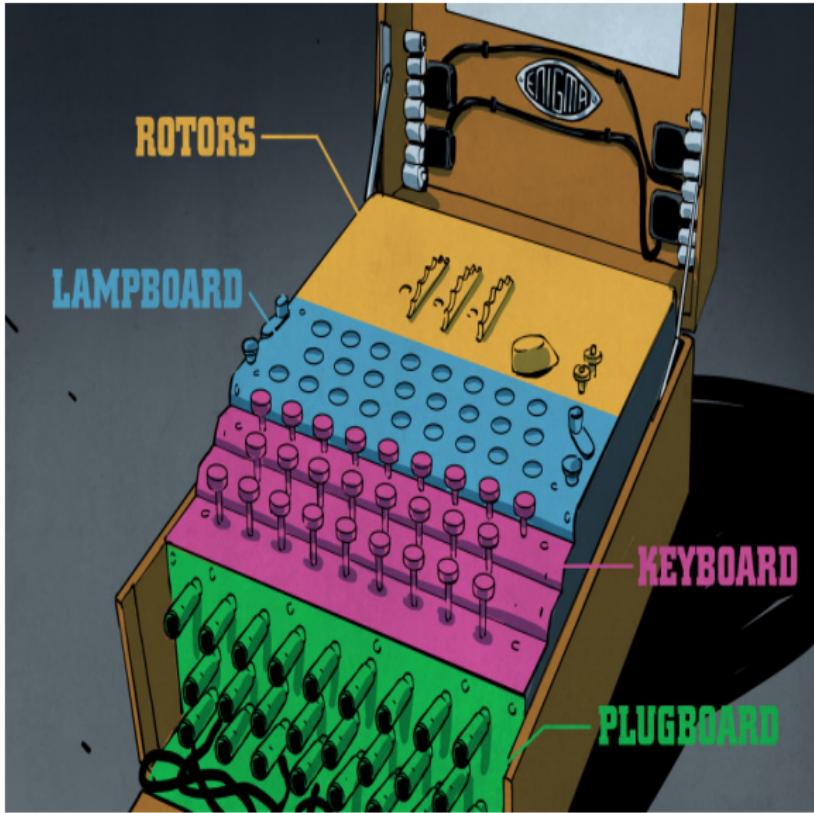
Section 2

PCode Emulation with Python

PCode Emulation with Python - Use Cases

- ▶ General Benefits
 - ▶ Middle ground between static & dynamic analysis
 - ▶ Simplifies analysis requiring hardware breakpoints
 - ▶ Customize environment for dynamic-ish analysis
- ▶ Obvious Use Cases
 - ▶ Examining malicious code behavior
 - ▶ Reversing crypto algorithms
 - ▶ Reversing code from an unusual architecture

An Enigma



PCode Emulation - Initial RAM Environment Setup

- ▶ Output @ 0x0: "\x00\x00\x00\x00\x00\x00\x00\x00"
- ▶ Key / initial rotor states @ 0x8: "AAA"
- ▶ Rotors selected @ 0x10: "123"
- ▶ Swaps @ 0x18 : 0x00
- ▶ Input text @ 0x20: "AAAAAAA\x00"

```
00000000: 0000 0000 0000 0000 4141 4100 0000 0000  
00000010: 3132 3300 0000 0000 0000 0000 0000 0000  
00000020: 4141 4141 4141 4100 0000 0000 0000 0000
```

```
state.ram.store(0x08, 8, 0x0000000000414141)  
state.ram.store(0x10, 8, 0x0000000000333231)  
state.ram.store(0x18, 8, 0x0000000000000000)  
state.ram.store(0x20, 8, 0x0041414141414141)
```

PCode Emulation - Parameter 0

The screenshot shows the Immunity Debugger interface with several windows open:

- Program Trees**: Shows the file structure of `enigma.x64`, including sections like `.bss`, `.data`, `.go`, and `.dy`.
- Listing: enigma.x64**: Displays assembly code for the `run_alg` function.
- Decompile: run_alg - (enigma.x64)**: Shows the C decompiled code for the `run_alg` function.
- Parameter Entry**: A modal dialog box asking for an integer value for parameter 0, with the input field containing "0".
- Console - Scripting**: Displays the command `pcode_interpreter.py> Running...` and features a cartoon dragon icon.
- Data Type Manager**: Shows data types including `BuiltinTypes`, `enigma.x64`, and `generic_clib_64`.

At the bottom, the assembly view shows the instruction `001011bc run_alg PUSH RBP`. The status bar also displays the same assembly address and instruction.

PCode Emulation - Parameter 1

The screenshot shows the Immunity Debugger interface during PCode emulation. The main window displays assembly code for the `run_alg` function:

```
1 int run_alg(char *out_text,char *key,char *rotors,char *swaps,char *text)
2 {
3     char *pcVar1;
4     char cVar2;
5     int iVar3;
6     size_t sVar4;
7     undefined8 local_48;
8     undefined8 local_40;
9     undefined8 local_38;
10    undefined8 local_30;
11    undefined8 local_28;
12 }
```

A parameter entry dialog is open, prompting for an integer value for parameter 1, with the key type set to `char *`. The value `8` is entered.

The bottom status bar shows the assembly address `001011bc`, the current instruction `run_alg`, and the register state `PUSH RBP`.

PCode Emulation - Parameter 2

The screenshot shows the Immunity Debugger interface during a PCode emulation session. The main window displays assembly code for the `run_alg` function from the `enigma.x64` program. A parameter entry dialog is open, prompting for an integer value for parameter 2, which is defined as `rotors type char`. The user has entered the value `16`. In the bottom right corner, a modal window titled `pcode_interpreter.py` is visible, featuring a cartoon dragon logo and the message `Running...`. The bottom status bar shows the assembly address `001011bc`, the current instruction `run_alg`, and the stack frame `PUSH RBP`.

File Edit Analysis Navigation Search Select Tools Window Help

Program Trees Listing: enigma.x64

Parameter Entry

Specify integer value for parameter 2 rotors type char * 16

OK Cancel

Decompile: run_alg - (enigma.x64)

```
1
2 int run_alg(char *out_text,char *key,char *rotors,char *swaps,char *text)
3 {
4     char *pcVar1;
5     char cVar2;
6     int iVar3;
7     size_t sVar4;
8     undefined8 local_a8;
9     undefined8 local_a0;
10    undefined8 local_98;
11    undefined8 local_90;
12 }
```

Console - Scripting

pcode_interpreter.py> Running...

Cancel

Data Type Manager

enigma.x64

generic_clib_64

001011bc run_alg PUSH RBP

PCode Emulation - Parameter 3

The screenshot shows the Immunity Debugger interface during a PCode emulation session. The main window displays assembly code for the `run_alg` function from the `enigma.x64` module. A parameter entry dialog is open, prompting for an integer value for parameter 3, with the current input being `241`. In the bottom right corner, a modal window titled `pcode_interpreter.py` is visible, featuring a cartoon dragon illustration and the text `pcode_interpreter.py> Running...`. The bottom status bar shows the assembly address `001011bc`, the current instruction `run_alg`, and the stack frame indicator `PUSH RBP`.

File Edit Analysis Navigation Search Select Tools Window Help

Program Trees Listing: enigma.x64

Parameter Entry

Specify integer value for parameter 3 swaps type char *

OK Cancel

Decompile: run_alg - (enigma.x64)

```
1
2 int run_alg(char *out_text,char *key,char *rotors,char *swaps,char *text)
3 {
4     char *pcVar1;
5     char cVar2;
6     int iVar3;
7     size_t sVar4;
8     undefined8 local_a8;
9     undefined8 local_a0;
10    undefined8 local_98;
11    undefined8 local_90;
12 }
```

Console - Scripting

pcode_interpreter.py> Running...

Cancel

Registers, CPU, Names

- enigma.x64
 - .bss
 - .data
 - .go
 - .go
 - .dy
 - .dy
- Program T

Symbol Table

- Registers, CPU, Names
 - run_alg
 - step_rotor
 - strlen
 - strncpy
 - to_upper
 - transform_rotor
- Labels
- Classes

Filter:

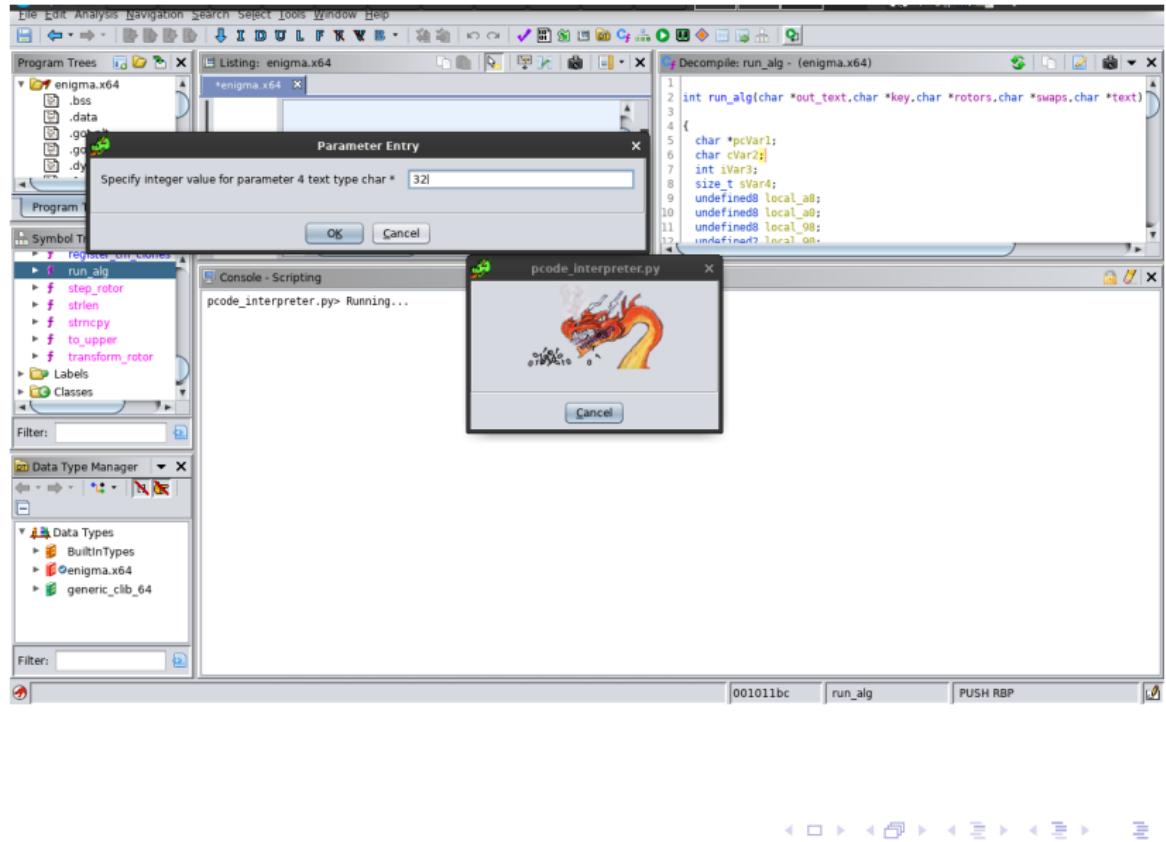
Data Type Manager

- Data Types
 - BuiltinTypes
 - enigma.x64
 - generic_clib_64

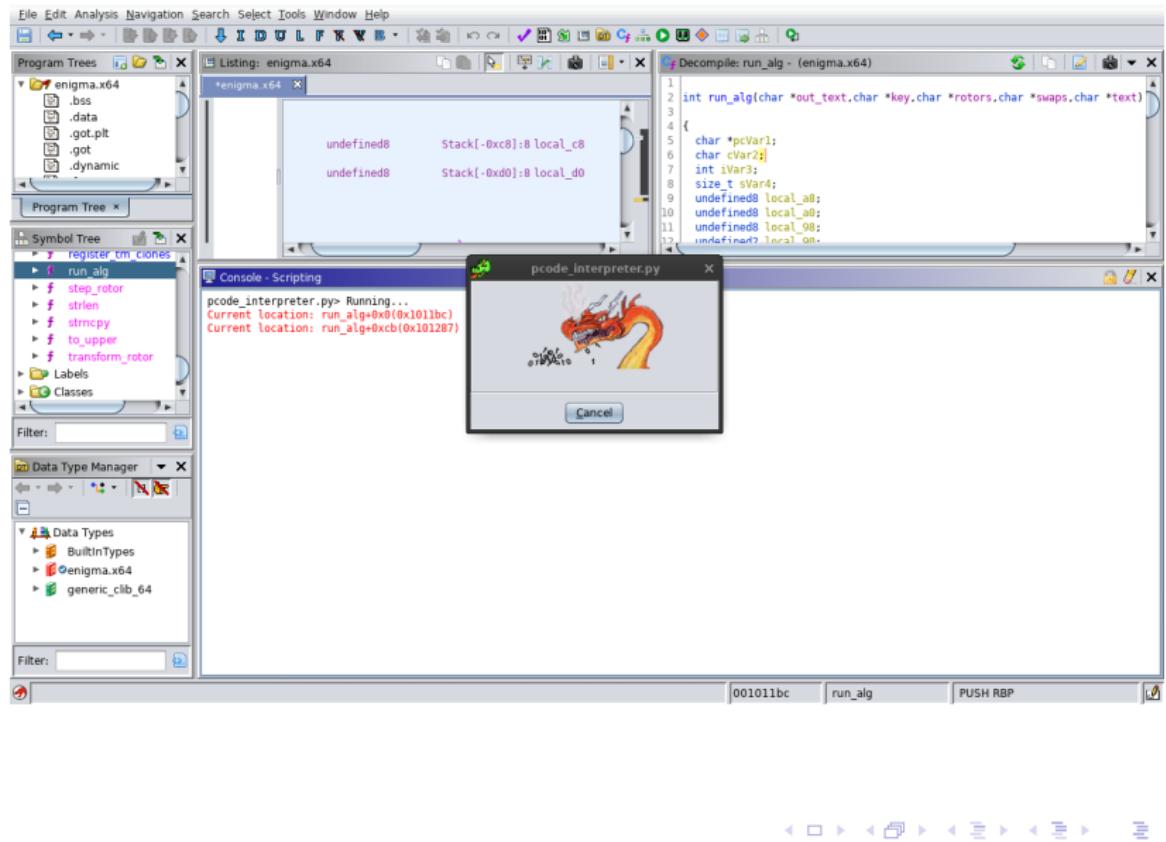
Filter:

001011bc run_alg PUSH RBP

PCode Emulation - Parameter 4



PCode Emulation - Running...



PCode Emulation - Running...

The screenshot shows the Immunity Debugger interface during the analysis of an Enigma cipher. The main window displays assembly code for the `run_alg` function, which handles the encryption process. The assembly view shows instructions like `undefined8 Stack[-0xc8]:8 local_c8` and `undefined8 Stack[-0x0]:8 local_d0`. Below the assembly is a decompiled Python script for the `pcode_interpreter.py` module, which contains logic for the cipher's internal state management.

The left sidebar includes:

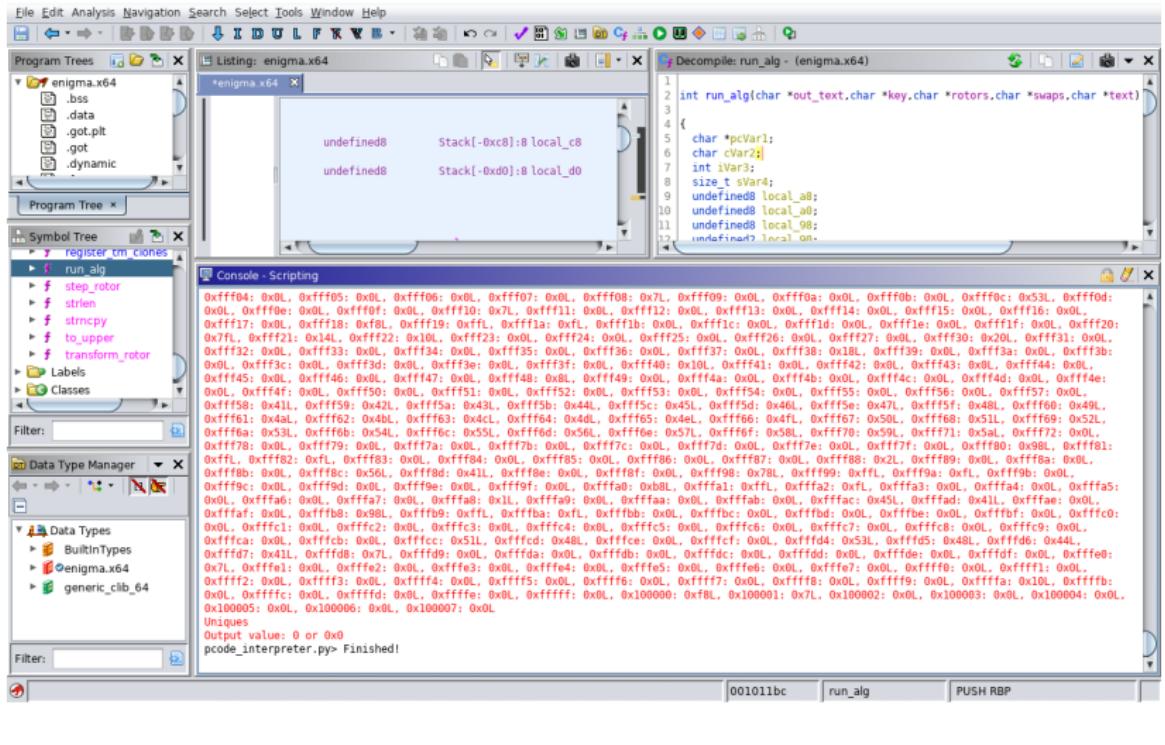
- Program Trees**: Shows the file structure of `enigma.x64`, including sections like `.bss`, `.data`, `.got.plt`, `.got`, and `.dynamic`.
- Symbol Tree**: Lists symbols such as `register_tm_clones`, `run_alg`, `step_rotor`, `strlen`, `strcpy`, `to_upper`, `transform_rotor`, and labels like `Labels` and `Classes`.
- Data Type Manager**: Manages data types for `enigma.x64`, `BuiltinTypes`, and `generic_clib_64`.

The bottom status bar shows memory addresses `001011bc` and `run_alg`, along with the instruction `PUSH RBP`.

PCode Emulation - Running...

About 2.25 minutes later...

PCode Emulation - Complete!



PCode Emulation - Complete!

The screenshot shows the Immunity Debugger interface with the following panes:

- Program Trees**: Shows the file structure of `enigma.x64`, including sections like .bss, .data, .got.plt, .got, and .dynamic.
- Listing: enigma.x64**: Displays assembly code with two stack frames:
 - undefined8 Stack[-0xc8]:8 local_c8
 - undefined8 Stack[-0x0]:8 local_d0
- Decompile: run_alg - (enigma.x64)**: Shows the decompiled C-like pseudocode for the `run_alg` function.
- Console - Scripting**: Lists the assembly locations where specific strings were found during the analysis.
- Data Type Manager**: Manages data types, currently showing entries for `enigma.x64` and `generic_clib_64`.
- Registers**: Shows the state of CPU registers at the final state.
- Stack**: Shows the current state of the stack.
- Memory Dump**: Provides a hex dump of memory starting at address `001011bc`.

PCode Emulation - Checking Work

- ▶ Emulation output:
 - ▶ ‘\x46\x54\x5a\x4d\x47\x49\x53’ == ‘FTZMGIS’

PCode Emulation - Checking Work

- ▶ Emulation output:
 - ▶ '\x46\x54\x5a\x4d\x47\x49\x53' == 'FTZMGIS'
- ▶ Actual code output:
 - ▶ './enigma AAA 123 "' AAAA' == 'FTZMGIS'

PCode Emulation with Python - Shortcomings & Next Steps

- ▶ Shortcomings
 - ▶ Interface is rough
 - ▶ Problems with headless mode
 - ▶ Syscall and library handling...
- ▶ Improvements
 - ▶ Improve those shortcomings
 - ▶ More analysis methods than 'execute a function'
 - ▶ Breakpoints!
 - ▶ Simplify environment input and output

Section 3

Ghidra Scripts via Python

Ghidra Scripts

- ▶ Why use Python?
 - ▶ The alternative is Java
 - ▶ The standard library is massive
 - ▶ Tons of open source projects to import (Angr?)
 - ▶ Simplified prototyping (The console?)
- ▶ How does Ghidra employ Python?
 - ▶ Jython - Java Python
 - ▶ Python 2.7.1

Minimal Python Script Example

```
#Documentation here
#@author Author name and email
#@category PCode
#@keybinding
#@menupath
#@toolbar

from __future__ import print_function

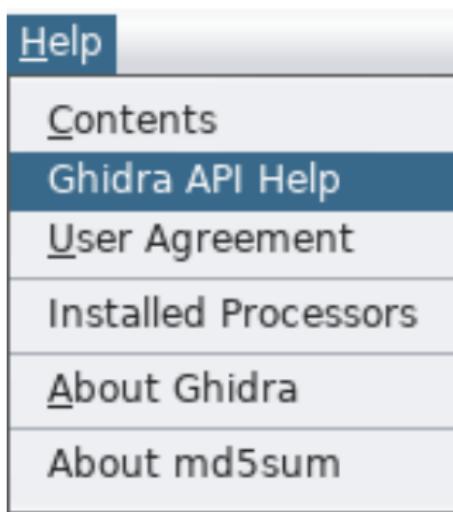
import logging

logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)

def func_here():
    logger.info("HelloWorld!")

if __name__ == "__main__":
    func_here()
```

Help Documentation



Help Documentation

Modifier and Type	Method	Description
void	<code>analyzeAll(Program program)</code>	Starts auto-analysis on the specified program and performs complete analysis of the entire program.
void	<code>analyzeChanges(Program program)</code>	Starts auto-analysis if not started and waits for pending analysis to complete.
Address	<code>askAddress(java.lang.String title, java.lang.String message)</code>	Returns an Address, using the String parameters for guidance.
byte[]	<code>askBytes(java.lang.String title, java.lang.String message)</code>	Returns a byte array, using the String parameters for guidance.
<T> T	<code>askChoice(java.lang.String title, java.lang.String message, java.util.List<T> choices, T defaultValue)</code>	Returns an object that represents one of the choices in the given list.
<T> java.util.List<T>	<code>askChoices(java.lang.String title, java.lang.String message, java.util.List<T> choices)</code>	Returns an array of Objects representing one or more choices from the given list.
<T> java.util.List<T>	<code>askChoices(java.lang.String title, java.lang.String message, java.util.List<T> choices, java.util.List<java.lang.String> choiceLabels)</code>	Returns an array of Objects representing one or more choices from the given list.
java.io.File	<code>askDirectory(java.lang.String title, java.lang.String approveButtonText)</code>	Returns a directory File object, using the String parameters for guidance.
DomainFile	<code>askDomainFile(java.lang.String title)</code>	Returns a DomainFile, using the title parameter for guidance.
double	<code>askDouble(java.lang.String title, java.lang.String message)</code>	Returns a double, using the String parameters for guidance.

Useful API

- ▶ Request input from the user
 - ▶ Normal data types
 - ▶ `askBytes(...)` -> bytes
 - ▶ `askInt(...)` -> int
 - ▶ `askString(...)` -> string
 - ▶ Ghidra-specific types
 - ▶ `askAddress(...)` -> GenericAddress
 - ▶ `askLanguage(...)` -> LanguageCompilerSpecPair
 - ▶ `askProgram(...)` -> ProgramDB
- ▶ Build Ghidra-specific types
 - ▶ `toAddr(int)` -> GenericAddress
 - ▶ `parseAddress(...)` -> GenericAddress
 - ▶ `parseLanguageCompileSpecPair(...)` -> ...
 - ▶ `parseProjectFolder(...)` -> ...

Useful API

- ▶ Introspect the open program
 - ▶ currentProgram -> ProgramDB
 - ▶ currentAddress -> GenericAddress
 - ▶ currentProgram.getLanguage() -> SleighLanguage
 - ▶ getFirstFunction() & getLastFunction() -> FunctionDB
 - ▶ getFunctionContaining(Address) -> FunctionDB
 - ▶ getDataContaining(Address) -> ...
 - ▶ getInstructionContaining(Address) -> ...
 - ▶ getPreComment(Address) -> string
 - ▶ setCurentLocation(...)

Useful API

- ▶ `currentProgram.getLanguage()`
 - ▶ `.getLanguageDescription() -> LanguageDescription`
 - ▶ `.getProgramCounter() -> Register` (includes name like 'RIP')
 - ▶ `.isBigEndian() -> bool`
- ▶ `currentProgram.getLanguage().getLanguageDescription()`
 - ▶ `.getProcessor() -> Processor` (includes name like 'x86')
 - ▶ `.getSize() -> int`

Useful API

- ▶ `currentProgram.getCompilerSpec()`
 - ▶ `.getStackPointer() -> Register`
 - ▶ `.stackGrowsNegative() -> bool`
- ▶ `currentProgram.getLanguage().getProgramCounter()`
 - ▶ `.getOffset() -> int`
 - ▶ `.getMinimumByteSize() -> int`
 - ▶ `.getBitLength() -> int`

Section 4

Resources

Ghidra Resources

- ▶ Official Ghidra Page: <https://ghidra-sre.org/>
- ▶ Great walkthrough:
<https://github.com/0xAlexei/INFILTRATE2019>
- ▶ PCode and Sleigh reference documentation
 - ▶ \${GHIDRA_INSTALL}/docs/languages/html/{pcoderef,sleigh}.html
 - ▶ Also found at -
<https://ghidra.re/courses/languages/html/>
- ▶ Ghidra Tutorials
 - ▶ \${GHIDRA_INSTALL}/GhidraClass
 - ▶ Also found at -
<https://ghidra.re/courses/GhidraClass/>

Questions?