

Introduction to Software Reverse Engineering with Ghidra Session 4: Ghidra Features

Hackaday U
Matthew Alt



#Outline: Advanced Features

- Ghidra Features
 - Loading external libraries
 - Patch analysis / diffing
 - Checksum Generation
 - P-Code Review
- Ghidra Extensions
 - Setting up a development environment
 - Python Scripting
- Course Wrap-up



#Course Administration

- This is our final session for the course!
 - Feedback and thoughts on course are welcome!
 - Fill out the google form to provide more feedback
 - (released later this week!)
- Office hours will be Thursday at 6:00 ET
- Questions can also be submitted through:
 - Hackaday.io chat room
 - Hackaday messaging



#Session Goals

- Review more Ghidra features
 - Patch analysis
 - Memory Manager
 - Extension via scripting
 - P-Code and analysis
- Ghidra Development and Scripting
 - Setting up an Eclipse based development environment
 - Scripting examples
- Course Wrap-up and conclusion
 - Review materials we've covered
 - Take final questions

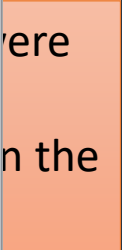


#External Libraries

- Ghidra can be used to analyze external libraries utilized by your target binary
 - Shared objects, DLLs, etc
- Ghidra can import and analyze these external libraries
 - This can be done at load time or after initial analysis
- Loading external libraries can assist with auto analysis
 - Type recognition and function prototype generation



ere
n the



#Ghidra Tip: Patch Diffing

- Ghidra allows for version tracking of all imported binaries
 - This makes collaboration when reversing simpler
- When using the version tracker, files are checked in and out of the project
 - Similar to a git workflow
- This version tracking feature can also be used to track patches and changes to binaries



#Patch Diffing: Example

Version Tracking: VT_patch_patch2

Version Tracking Matches - [Session: VT_patch_patch2] - 352 matches

Tag	Session ID	Status	Type	Score	Confidence (%)	Votes	# Conflicts	Multiple S...	Source Namespace	Source Label	Source Address	Multiple De...	Dest Namespace	Dest Label	Dest Address	Source Length	Dest Length	Algorithm
23	Function	1.000	1.000	0	2	2	Global	_init	001004e8	2	Global	_init	001004e8	23	23	Exact Function Mnemonics ...		
26	Function	1.000	1.000	0	2	2	Global	_init	001004e8	2	Global	_init	001004e8	23	23	Exact Symbol Name Match		
22	Function	1.000	1.000	0	2	2	Global	_init	001004e8	2	Global	_init	001004e8	23	23	Exact Function Bytes Match		
21	Function	1.000	0.891	0	2	2	Global	_init	001004e8	2	Global	_init	001004e8	23	23	Similar Symbol Name Match		
21	Data	1.000	1.125	0	0	0	Global	_IO_stdin_u...	00100710	Global	_IO_stdin_used	00100720	4	4	Similar Symbol Name Match			
26	Data	1.000	1.000	0	0	0	Global	_IO_stdin_u...	00100710	Global	_IO_stdin_used	00100720	4	4	Exact Symbol Name Match			
26	Function	1.000	1.000	0	0	0	<EXTERNAL>	_ITM_dereg...	External[?]	<EXTERNAL>	_ITM_deregisterTMCloneT...	External[?]	0	0	Exact Symbol Name Match			
26	Function	1.000	1.000	0	0	0	Global	_ITM_dereg...	00302000	Global	_ITM_deregisterTMCloneT...	00302000	1	1	Exact Symbol Name Match			
26	Function	1.000	1.000	0	0	0	<EXTERNAL>	_ITM_regist...	External[?]	<EXTERNAL>	_ITM_registerTMCloneTable	External[?]	0	0	Exact Symbol Name Match			
26	Function	1.000	1.000	0	0	0	Global	_ITM_regist...	00302020	Global	_ITM_registerTMCloneTable	00302020	1	1	Exact Symbol Name Match			
20	Function	1.000	1.000	0	0	0	Global	_start	00100530	Global	_start	00100530	43	43	Exact Function Instructions ...			
23	Function	1.000	1.000	0	0	0	Global	_start	00100530	Global	_start	00100530	43	43	Exact Function Mnemonics ...			
26	Function	1.000	1.000	0	0	0	Global	_start	00100530	Global	_start	00100530	43	43	Exact Symbol Name Match			
21	Function	1.000	0.648	0	0	0	Global	_start	00100530	Global	_start	00100530	43	43	Similar Symbol Name Match			
26	Function	1.000	1.000	0	0	0	Global	add	0010063a	Global	add	0010063a	20	25	Exact Symbol Name Match			
25	Data	1.000	0.398	0	2	2	Global	cie_001007...	00100768	Global	cie_001007a8	00100768	4	4	Duplicate Exact Symbol Na...			
25	Data	1.000	0.398	0	2	2	Global	cie_001007...	00100768	Global	cie_00100778	00100778	4	4	Duplicate Exact Symbol Na...			
25	Data	1.000	0.398	0	2	2	Global	cie_001007...	00100798	Global	cie_001007a8	00100798	4	4	Duplicate Exact Symbol Na...			
25	Data	1.000	0.398	0	2	2	Global	cie_001007...	00100798	Global	cie_00100778	00100798	4	4	Duplicate Exact Symbol Na...			
21	Data	1.000	1.000	0	0	0	Global	completed 7	00301010	4	Global	completed 7607	00301010	1	1	Similar Symbol Name Match		

Filter: Score Filter: 0.000 to 1.000 Confidence Filter: -9.999 to 9.999 Length Filter: 0

Version Tracking Markup Items - [Session: VT_patch_patch2] - 2 markup items

Status	Source Address	Dest Address	Markup Type	Source Value	Current Dest Value	Original Dest Value
✓✓	0010063a	0010063a	Function Signature	undefined add()	undefined add()	undefined add()
✓✓	0010063a	0010063a	Function Name	add	add	add

Filter:

Decompile View Listing View

Source: add() in /patch

```
undefined4 Stack[-0x10]:4 local_10
XREF[2]: 00100644 (R), 00100641 (W), 00100647 (R)
XREF[3]: Entry Point(*), main:00100681(c), 00100744

0010063a 55 PUSH RBP
0010063b 48 89 e5 MOV RBP,RBP
0010063e 89 7d fc MOV dword ptr [RBP + local_c],EDI
00100641 89 75 f8 MOV dword ptr [RBP + local_10],ESI
00100644 8b 55 fc MOV EDX,dword ptr [RBP + local_c]
00100647 8b 45 f8 MOV EAX,dword ptr [RBP + local_10]
0010064a 01 d0 ADD EAX,EDX
0010064c 5d POP RBP
0010064d c3 RET
```

Destination: add() in /patch2

```
0010063a 55 PUSH RBP
0010063b 48 89 e5 MOV RBP,RBP
0010063e 89 7d fc MOV dword ptr [RBP + local_c],EDI
00100641 89 75 f8 MOV dword ptr [RBP + local_10],ESI
00100644 8b 55 fc MOV EDX,dword ptr [RBP + local_c]
00100647 8b 45 f8 MOV EAX,dword ptr [RBP + local_10]
0010064a 01 c2 ADD EDX,EAX
0010064c 8b 45 f8 MOV EAX,dword ptr [RBP + local_10]
0010064f 01 d0 ADD EAX,EDX
00100651 5d POP RBP
00100652 c3 RET
```

Version Tracking Markup... x Version Tracking Impli... x

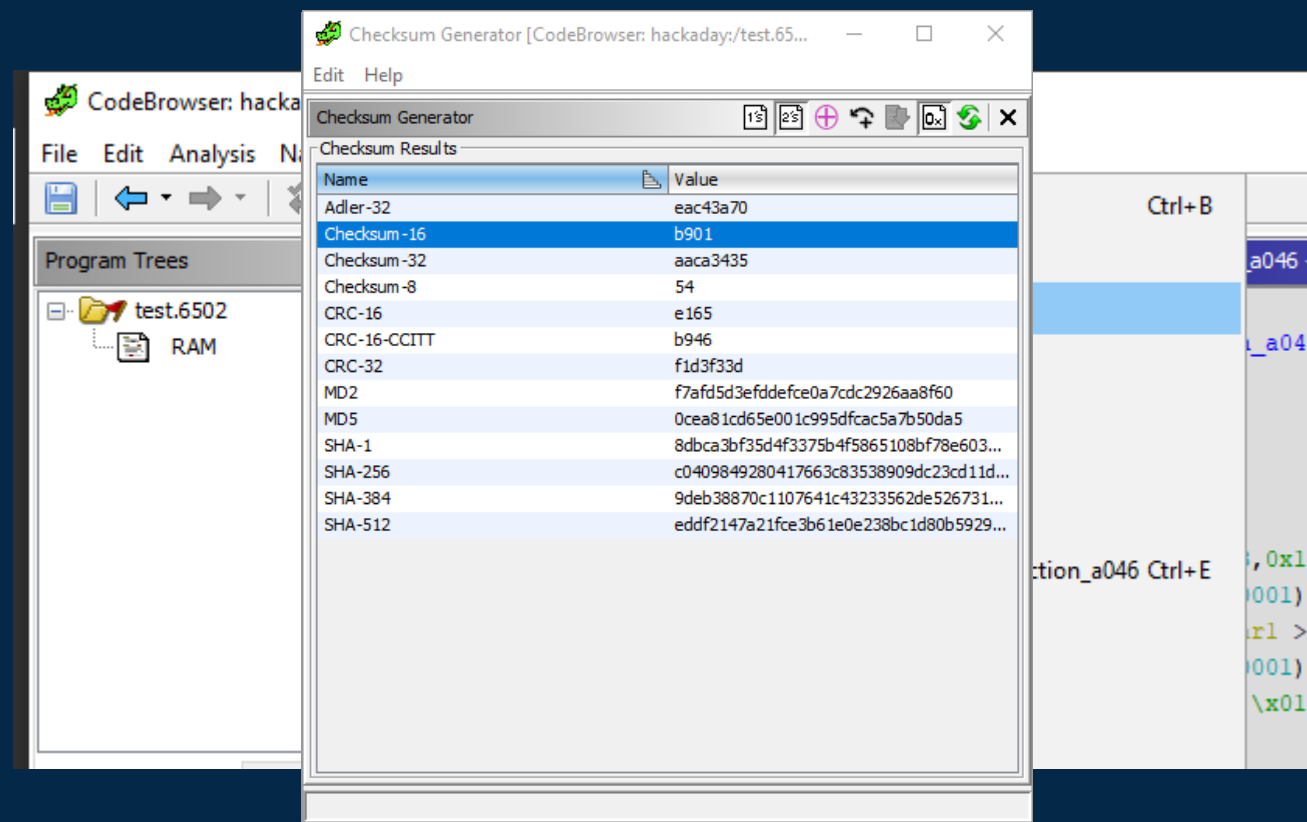


#Ghidra Tip: Checksum Tool

- Ghidra contains a built in checksum generator
- Multiple types of checksums can be generated
 - CRC-16/32
 - MD2/5
 - SHA1/256/384
- Checksums can have various operations applied to them
 - 1's/2's complement
 - Recompute with carry / xor



#Ghidra Tip: Checksum Tool

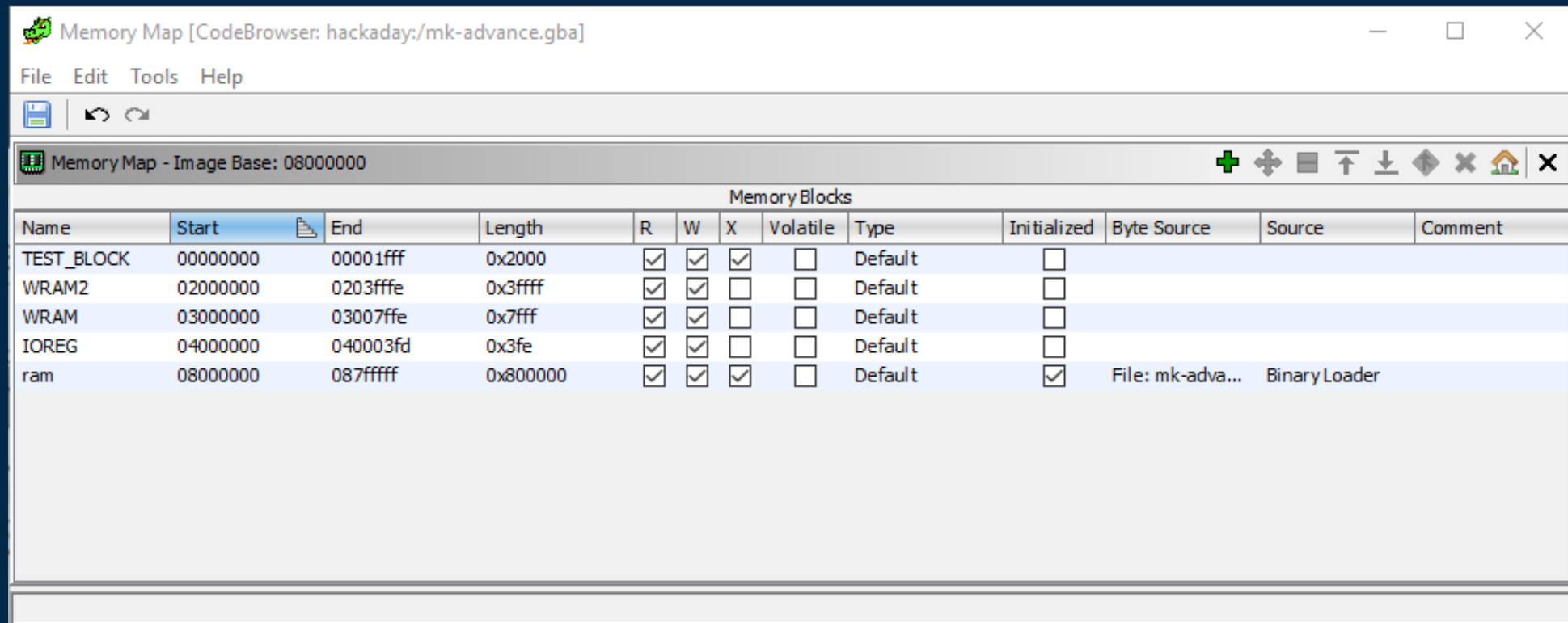


#Ghidra Tip: Memory Manager

- Memory regions can be added manually via Ghidra's Memory Map tool
 - This is often used when looking at firmware images
 - Programs can be rebased from this tool as well
- Each memory region has the following attributes in the manager
 - Start address
 - Size/End address
 - Read / Write / Execute
 - Overlay



#Ghidra Tip: Memory Manager



#Ghidra Tip: Memory Manager

- Binary files can be imported into memory regions
 - This is useful for RAM dumps if you can acquire them!
 - File -> Add to Program
- Some memory regions can be set up as overlay blocks
 - Useful for older consoles with regions that can be bank switched
- The memory manager functionality can also be instrumented via Ghidra's API
 - This is how loaders automatically generate appropriate regions!



#Ghidra: Internals

- One of Ghidra's most powerful features is having a decompiler for each processor that it supports
- To implement a processor module in Ghidra, one must first write a processor module in SLEIGH
- This processor module will handle both:
 - Decoding of bytes to assembly code (disassembly)
 - Generation of Pcode
 - Generation of decompilation based on Pcode



#SLEIGH: Overview

- SLEIGH specifies the translation from assembly code to P-Code
- SLEIGH is the language that is used to create processor modules for Ghidra
- SLEIGH modules define the processor's instructions, registers and features
 - Examples are found at Ghidra/Processors
- SLEIGH modules are used for both disassembly and P-Code creation
 - P-Code is what is analyzed by the decompiler



#Sleigh: 6502 Example

- Each processor definition will consist of the following files:
 - CPU.cspec
 - CPU.ifdefs
 - CPU.pspec
 - CPU.slaspec
- LDEFS: Language definitions
- PSPEC: Processor Specification
- CSPEC: Compiler Specification
- SLASPEC: SLEIGH Specification



#Sleigh: 6502 Example – LDEFS

```
<language processor="6502"  
  endian="little"  
  size="16"  
  variant="default"  
  version="1.0"  
  slafile="6502.sla"  
  processorspec="6502.pspec"  
  manualindexfile="../manuals/6502.idx"  
  id="6502:LE:16:default">  
  <description>6502 Microcontroller Family</description>  
  <compiler name="default" spec="6502.cspect" id="default"/>  
  <external_name tool="IDA-PRO" name="m6502"/>  
  <external_name tool="IDA-PRO" name="m65c02"/>  
</language>
```

Language definitions include endianness, size – as well as definitions for the rest of the specification files for this particular processor.



#Sleigh: 6502 Example – PSPEC

```
<default_symbols>
  <symbol name="PORTA" address="0"/>
  <symbol name="PORTB" address="1"/>
  <symbol name="PORTC" address="2"/>
  <symbol name="PORTD" address="3"/>
  <symbol name="DDRA" address="4"/>
  <symbol name="DDRB" address="5"/>
  <symbol name="DDRC" address="6"/>
  <symbol name="DDRD" address="7"/>
  <symbol name="SPCR" address="A"/>
  <symbol name="SPSR" address="B"/>
  <symbol name="SPDR" address="C"/>
  <symbol name="BAUD" address="D"/>
</default_symbols>
<default_memory_blocks>
  <memory_block name="LOW_RAM" start_address="0x0000" length="0x0100" initialized="false"/>
  <memory_block name="STACK" start_address="0x0100" length="0x0100" initialized="false"/>
</default_memory_blocks>
<symbol name="PC" address="PC" type="code_ptr"/>
</default_symbols>
```

Processor Specification is used to define processor specific information such as:

- Program Counter
- Reset Vectors
- Interrupt Handlers
- Memory Mapped IO



#Sleigh: 6502 Example – CSPEC

```
<compiler_spec>
  <global>
    <range space="RAM"/>
  </global>
  <stackpointer register="SP" space="RAM" growth="negative"/>
  <returnaddress>
    <varnode space="stack" offset="1" size="2"/>
  </returnaddress>
  <default_proto>
    <prototype name="__stdcall" extrapop="2" stackshift="2" strategy="register">
      <input>
        <pentry minsize="1" maxsize="1">
          <register name="A"/>
        </pentry>
        <pentry minsize="1" maxsize="1">
          <register name="X"/>
        </pentry>
        <pentry minsize="1" maxsize="1">
          <register name="Y"/>
        </pentry>
      </input>
      <output>
        <pentry minsize="1" maxsize="1">
          <register name="A"/>
        </pentry>
      </output>
      <unaffected>
        <register name="SP"/>
      </unaffected>
    </prototype>
  </default_proto>
</compiler_spec>
```

The Compiler Specification defines things such as:

- Sizes for various data types
- Alignment rules / directives
- Stack growth / behavior
- Function prototypes
- Calling conventions



#Sleigh: 6502 Example – SLASPEC

```
# Immediate
define_endian=little;
OP1: "#imm8" is bbb=2; imm8 { tmp:1 = imm8; export * 1 imm8; }
define_alignment=1;
# Zero Page
```

```
OP1: imm8 is bbb=1; imm8 { export * 1 imm8; }
define_space RAM type=ram_space size=2 default;
define_space register type=register_space size=1;
```

```
:STA OP1 is (cc-1 & aaa-4) & OP1
define_register offset=0x00 size=1 [ A X Y P ];
define_register offset=0x20 size=2 [ PC SP ];
define_register offset=0x20 size=1 [ PCL PCH S S ];
define_resultFlags(A);
define_register offset=0x30 size=1 [ N V B D I Z C ], # Status Bits
```

The SLASPEC file contains things such as:

- Alignment definitions
- Address space definitions
- Register definitions
- Instruction definitions

SLASPEC defines how instructions are decoded / disassembled:

- Defines the parameters for disassembly
- Defines how instructions are shown in the listing

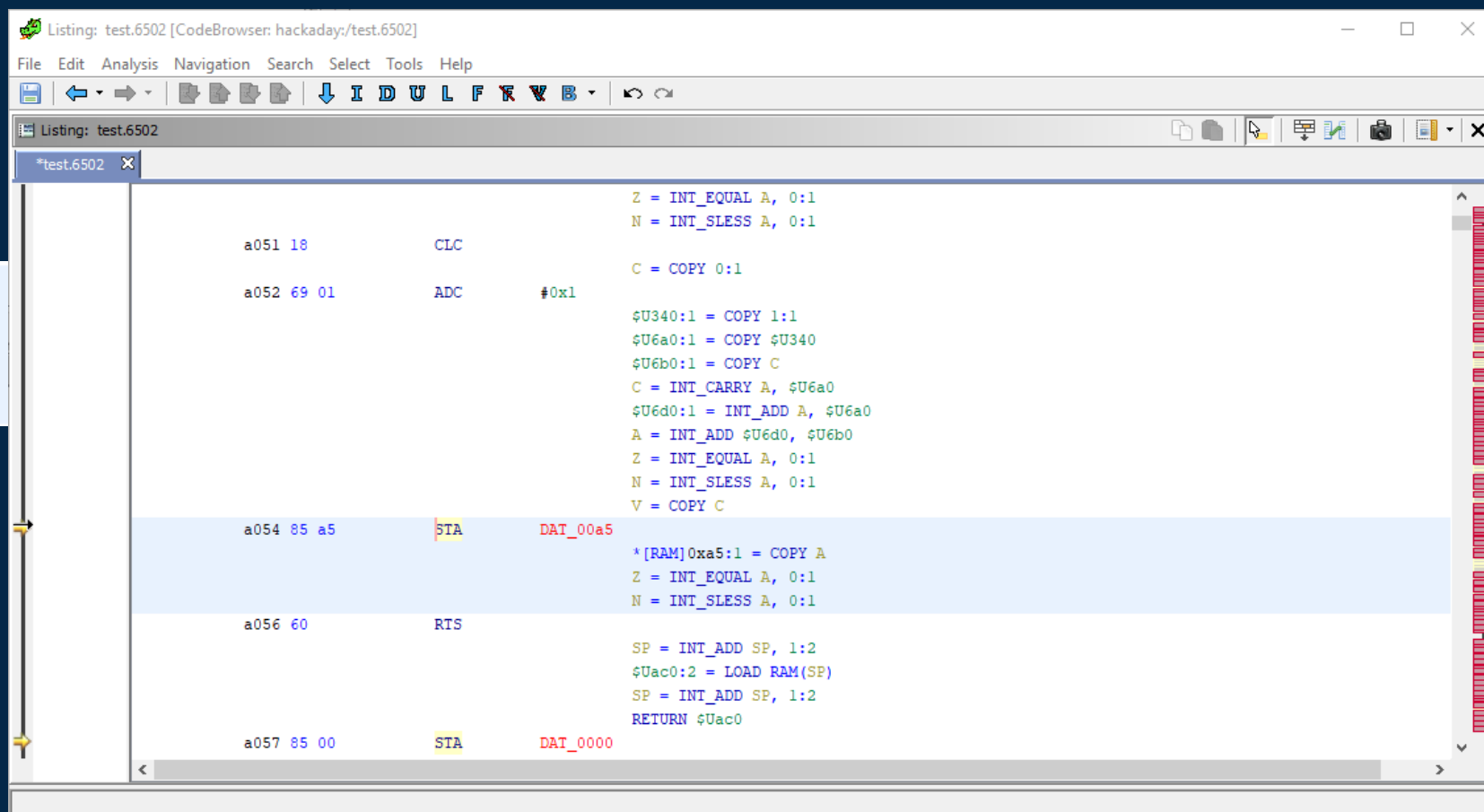


#P-Code: Overview

- Designed to be capable of modeling a general purpose processor
- P-Code is generated during the initial analysis process
- P-Code is a register transfer language
 - Used as an abstraction layer for CPU specific functions
 - Intermediate representation of assembly code
- P-Code is used to generate decompiled code



#P-Code: Example



```
Listing: test.6502 [CodeBrowser: hackaday:/test.6502]
File Edit Analysis Navigation Search Select Tools Help
Listing: test.6502
*test.6502 x
a051 18      CLC
a052 69 01    ADC      #0x1
a054 85 a5    STA      DAT_00a5
a056 60      RTS
a057 85 00    STA      DAT_0000

Z = INT_EQUAL A, 0:1
N = INT_SLESS A, 0:1
C = COPY 0:1
$U340:1 = COPY 1:1
$U6a0:1 = COPY $U340
$U6b0:1 = COPY C
C = INT_CARRY A, $U6a0
$U6d0:1 = INT_ADD A, $U6a0
A = INT_ADD $U6d0, $U6b0
Z = INT_EQUAL A, 0:1
N = INT_SLESS A, 0:1
V = COPY C
*[RAM]0xa5:1 = COPY A
Z = INT_EQUAL A, 0:1
N = INT_SLESS A, 0:1
SP = INT_ADD SP, 1:2
$Uac0:2 = LOAD RAM(SP)
SP = INT_ADD SP, 1:2
RETURN $Uac0
```

a054 85 a5

very detailed



#P-Code: Exercise

- Load the exercises in `session-four/exercises/script-exercises/`
 - PPC
 - ARM
 - I386
 - X86_64
- Examine the P-code for each example
 - How are they different?
 - Are they similar in any way?

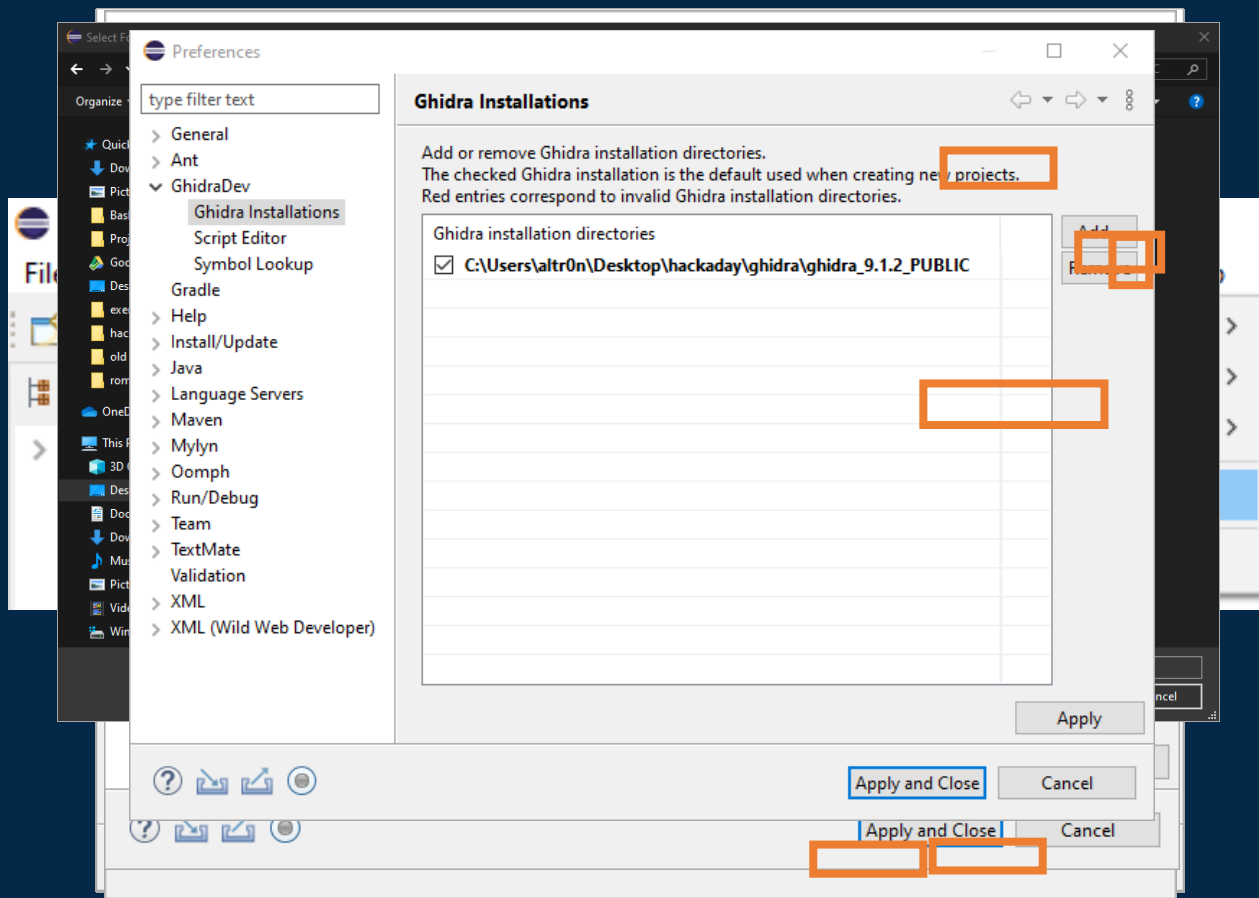


#Ghidra Extension: Eclipse

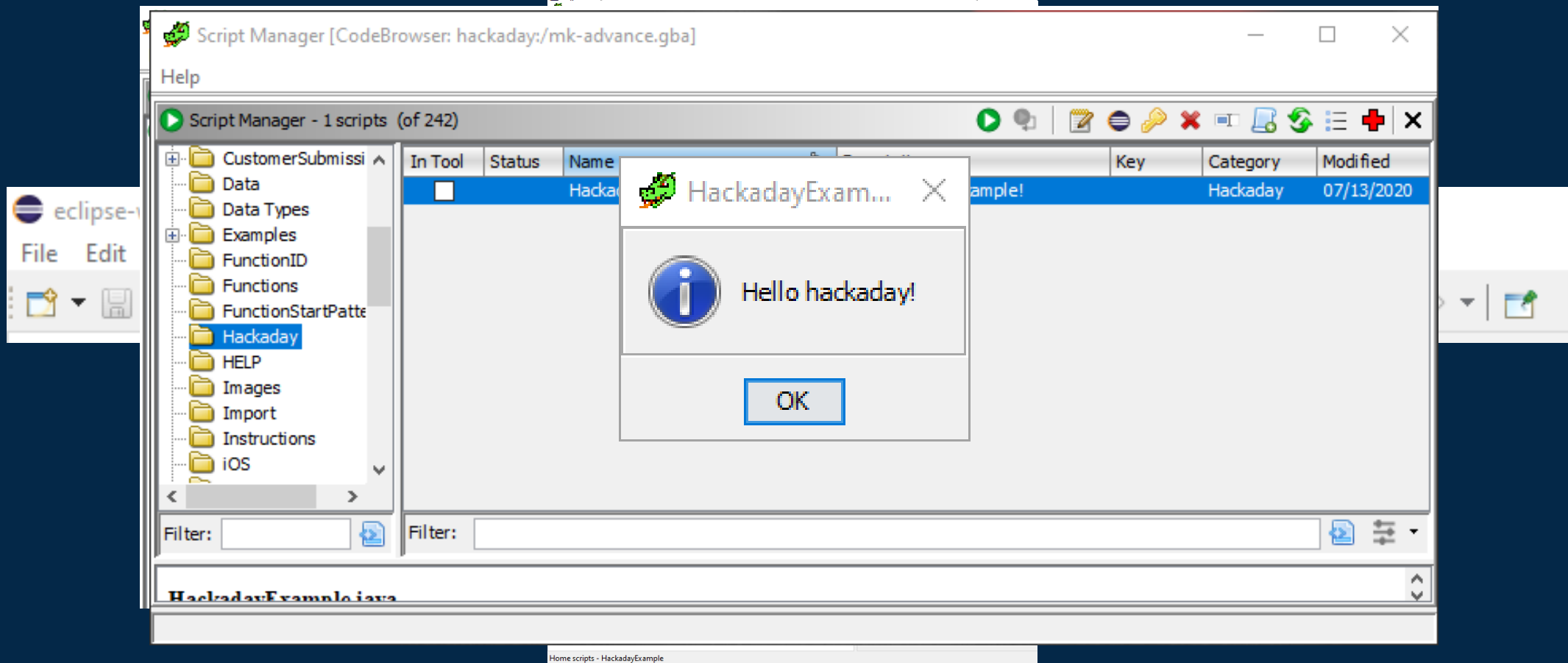
- Ghidra can be extended via plugins
 - Loaders
 - Scripts
 - Modules
- These extensions can be developed in Eclipse using the GhidraDev plugin included with Ghidra
- Ghidra provides both a Java and Python API



#Ghidra Extension: Eclipse



#Ghidra Script Project Creation



#Ghidra Scripts: Python

- Ghidra scripts can be written in Python as well as Java
 - Requires the pydev plugin for eclipse integration
- Ghidra also features a python shell
 - Window -> Python
- The Python interpreter can utilize the FlatProgramAPI
 - This is what we will focus on!



#Ghidra Scripts: Globals

- Ghidra creates multiple global variables for use while scripting
 - currentAddress
 - currentHighlight
 - currentProgram
 - currentSelection
 - currentLocation
- All of these can be used to determine information relevant to your current cursor location!



#Ghidra Scripts: Addresses

- Ghidra utilizes the Address datatype heavily within it's API
 - `currentAddress` returns an Address object
- Address objects can NOT be treated like integers
 - Contain special function to add and subtract
 - `.add()` `.subtract()`, etc
- Address objects can be created using the AddressFactory
 - `currentProgram.getAddressFactory.getAddress("0x10000")`



#Scripting: Data

- Data can be generated automatically via scripting
 - createByte(currentAddress)
 - createChar(currentAddress)
 - createDWord(currentAddress)
 - createAsciiString(currentAddress)
- These functions all take a Ghidra address object as an argument
 - Eq: createDWord(currentAddress)

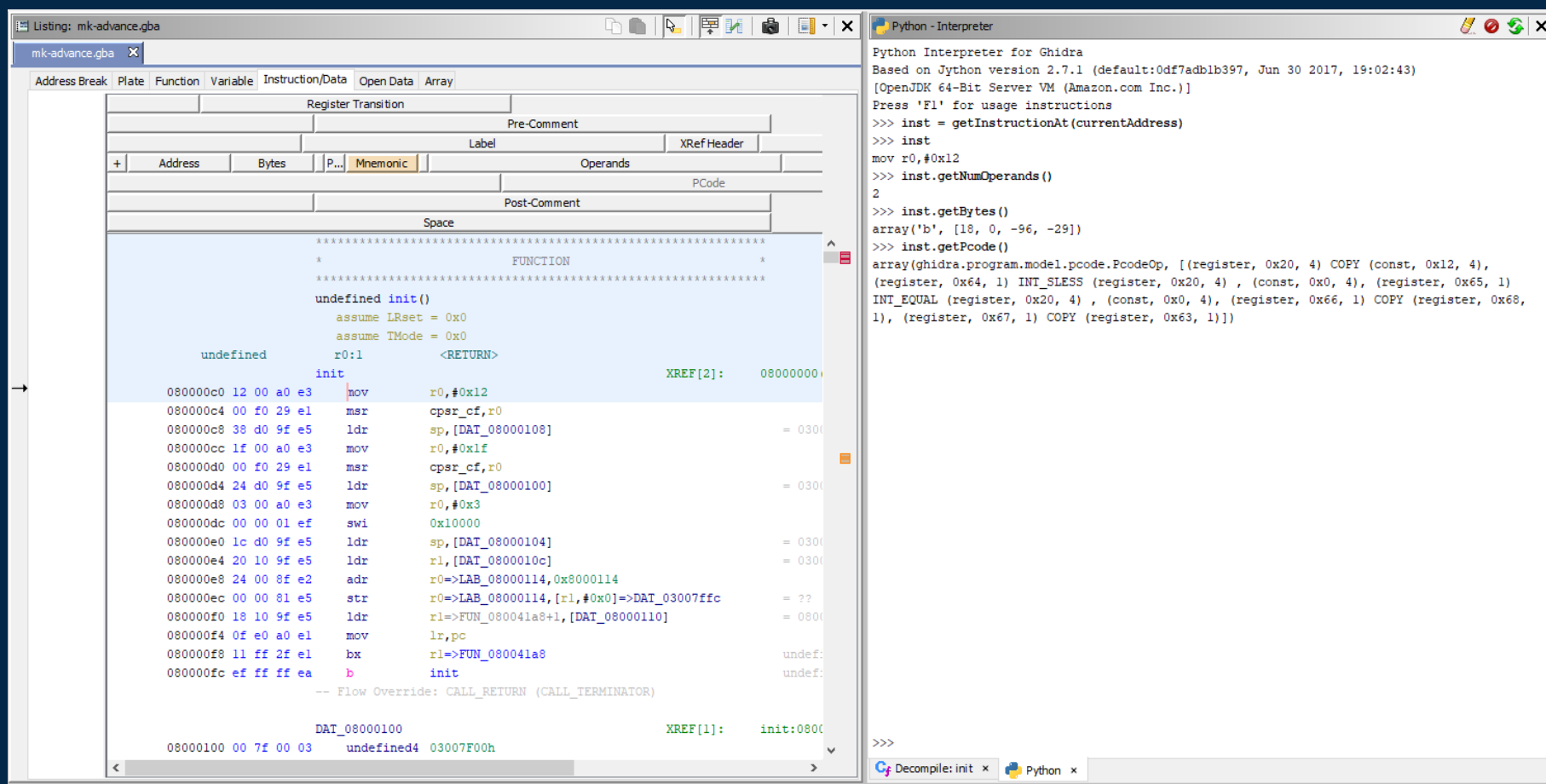


#Scripting: Instructions

- Ghidra scripts can parse and operate at the instruction level
- `getInstructionAt(address)`
 - Used to get an Instruction object representing the instruction at that address
- From an instruction object multiple instruction components can be derived
 - Number of operands
 - P-code representation
 - Operand objects (scalars,etc)



#Scripting: Instructions



The screenshot displays the Ghidra IDE interface with the 'mk-advance.gba' file open. The main window shows assembly instructions for a function named 'init'. The instructions are listed with their addresses, hex values, mnemonics, and operands. The Python Interpreter window on the right shows the decompiled code for the 'init' function, which is a Python script using the Ghidra API to interact with the instruction set.

Assembly Instructions:

Address	Bytes	Mnemonic	Operands	PCode
080000c0	12 00 a0 e3	mov	r0, #0x12	
080000c4	00 f0 29 e1	msr	cpsr_cf, r0	
080000c8	38 d0 9f e5	ldr	sp, [DAT_08000108]	= 0300
080000cc	1f 00 a0 e3	mov	r0, #0x1f	
080000d0	00 f0 29 e1	msr	cpsr_cf, r0	
080000d4	24 d0 9f e5	ldr	sp, [DAT_08000100]	= 0300
080000d8	03 00 a0 e3	mov	r0, #0x3	
080000dc	00 00 01 ef	swi	0x10000	
080000e0	1c d0 9f e5	ldr	sp, [DAT_08000104]	= 0300
080000e4	20 10 9f e5	ldr	r1, [DAT_0800010c]	= 0300
080000e8	24 00 8f e2	adr	r0=>LAB_08000114, 0x8000114	
080000ec	00 00 81 e5	str	r0=>LAB_08000114, [r1, #0x0]=>DAT_03007ffc	= ??
080000f0	18 10 9f e5	ldr	r1=>FUN_080041a8+1, [DAT_08000110]	= 0800
080000f4	0f e0 a0 e1	mov	lr, pc	
080000f8	11 ff 2f e1	bx	r1=>FUN_080041a8	undef:
080000fc	ef ff ff ea	b	init	undef:

Python Interpreter Code:

```
Python Interpreter for Ghidra
Based on Jython version 2.7.1 (default:0df7adb1b397, Jun 30 2017, 19:02:43)
[OpenJDK 64-Bit Server VM (Amazon.com Inc.)]
Press 'F1' for usage instructions
>>> inst = getInstructionAt(currentAddress)
>>> inst
mov r0, #0x12
>>> inst.getNumOperands()
2
>>> inst.getBytes()
array('b', [18, 0, -96, -29])
>>> inst.getPcode()
array(ghidra.program.model.pcode.PcodeOp, [(register, 0x20, 4) COPY (const, 0x12, 4),
(register, 0x64, 1) INT_SLESS (register, 0x20, 4), (const, 0x0, 4), (register, 0x65, 1)
INT_EQUAL (register, 0x20, 4), (const, 0x0, 4), (register, 0x66, 1) COPY (register, 0x68,
1), (register, 0x67, 1) COPY (register, 0x63, 1)])
>>>
```



#Scripting: Functions

- Ghidra's FlatProgramAPI can be used to create and modify functions
 - Set signature, parameters, etc
- Functions can be accessed by providing an address:
 - `Func = getFunctionAt(address)`
- Functions can also be created at a specified address:
 - `createFunction(address)`



#Scripting: Functions

The screenshot displays the Ghidra IDE interface with a decompiled function named `init` and a Python interpreter window.

Ghidra Window (mk-advance.gba):

- Register Transition:** Shows the state of registers before and after the function.
- Pre-Comment:** Empty.
- Label:** `init`
- XRef Header:** Shows cross-references.
- Address Break:** A table with columns: Address, Bytes, P..., Mnemonic, Operands, PCode.
- Post-Comment:** Empty.
- Space:** A table with columns: Address, Bytes, P..., Mnemonic, Operands, PCode.

Function Analysis:

```
FUNCTION
*****
undefined init()
    assume LRset = 0x0
    assume TMode = 0x0
    undefined r0:l <RETURN>
    init
    080000c0 12 00 a0 e3 mov r0,#0x12
    080000c4 00 f0 29 e1 msr cpar_cf,r0
    080000c8 38 d0 9f e5 ldr sp,[DAT_08000108] = 0300
    080000cc 1f 00 a0 e3 mov r0,#0x1f
    080000d0 00 f0 29 e1 msr cpar_cf,r0
    080000d4 24 d0 9f e5 ldr sp,[DAT_08000100] = 0300
    080000d8 03 00 a0 e3 mov r0,#0x3
    080000dc 00 00 01 ef swi 0x10000
    080000e0 1c d0 9f e5 ldr sp,[DAT_08000104] = 0300
    080000e4 20 10 9f e5 ldr r1,[DAT_0800010c] = 0300
    080000e8 24 00 8f e2 adr r0=>LAB_08000114,0x8000114
    080000ec 00 00 81 e5 str r0=>LAB_08000114,[r1,#0x0]=>DAT_03007ffc = ??
    080000f0 18 10 9f e5 ldr r1=>FUN_080041a8+1,[DAT_08000110] = 0800
    080000f4 0f e0 a0 e1 mov lr,pc
    080000f8 11 ff 2f e1 bx r1=>FUN_080041a8 undef:
    080000fc ef ff ff ea b init undef:
    -- Flow Override: CALL_RETURN (CALL_TERMINATOR)

DAT_08000100 XREF[1]: init:0800
08000100 00 7f 00 03 undefined4 03007F00h
```

Python Interpreter:

```
Python Interpreter for Ghidra
Based on Jython version 2.7.1 (default:0df7adblb397, Jun 30 2017, 19:02:43)
[OpenJDK 64-Bit Server VM (Amazon.com Inc.)]
Press 'F1' for usage instructions
>>> func = getFunctionAt(currentAddress)
>>> func.getEntryPoint()
080000c0
>>> func.getName()
u'init'
>>> func.getReturnType()
undefined
```

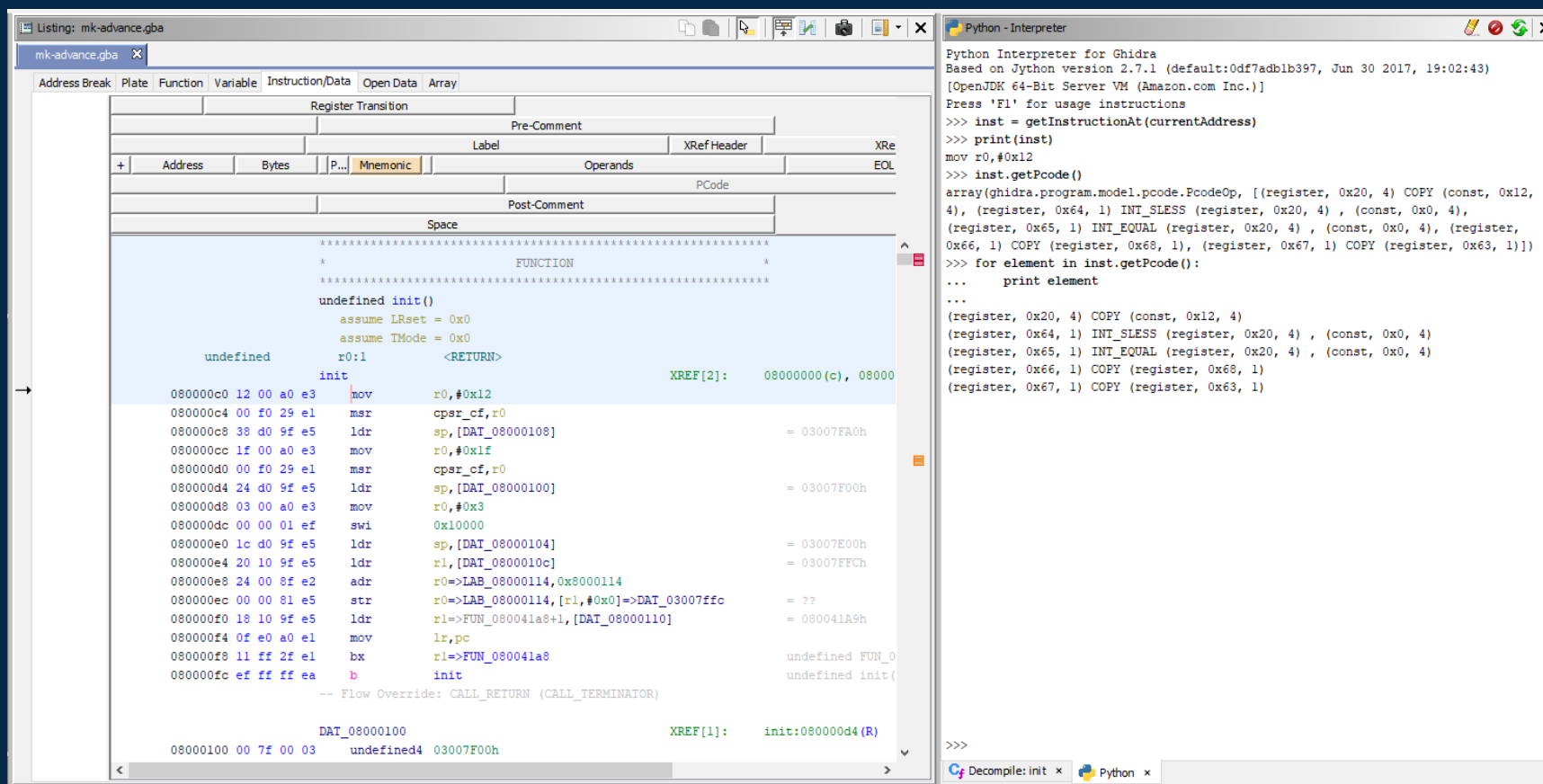


#Scripting: Pcode Extraction

- Pcode objects can be accessed from instruction objects
 - `Inst.getPcode()`
- PcodeOps describe the Pcode operations for a given instruction
- PcodeOps can be emulated using Ghidra's Pcode emulator



#Scripting: Pcode Extraction



The screenshot displays the Ghidra IDE interface with the 'mk-advance.gba' file open. The main window shows assembly code with a function 'init' starting at address 080000c0. The assembly code includes instructions like 'mov r0, #0x12', 'msr cpsr_cf, r0', 'ldr sp, [DAT_08000108]', and 'swi 0x10000'. The right-hand pane shows the 'Python - Interpreter' window, which is running a script to extract pcode from the current instruction. The script uses the 'inst.getPcode()' method and iterates over the resulting pcode elements, printing their details. The output shows a series of pcode operations, including 'COPY', 'INT_SLESS', 'INT_EQUAL', and 'COPY' instructions with their respective registers and constants.

```
Python Interpreter for Ghidra
Based on Jython version 2.7.1 (default:0df7adblb397, Jun 30 2017, 19:02:43)
[OpenJDK 64-Bit Server VM (Amazon.com Inc.)]
Press 'F1' for usage instructions
>>> inst = getInstructionAt(currentAddress)
>>> print(inst)
mov r0,#0x12
>>> inst.getPcode()
array(ghidra.program.model.pcode.PcodeOp, [(register, 0x20, 4) COPY (const, 0x12,
4), (register, 0x64, 1) INT_SLESS (register, 0x20, 4) , (const, 0x0, 4),
(register, 0x65, 1) INT_EQUAL (register, 0x20, 4) , (const, 0x0, 4), (register,
0x66, 1) COPY (register, 0x68, 1), (register, 0x67, 1) COPY (register, 0x63, 1)])
>>> for element in inst.getPcode():
...     print element
...
(register, 0x20, 4) COPY (const, 0x12, 4)
(register, 0x64, 1) INT_SLESS (register, 0x20, 4) , (const, 0x0, 4)
(register, 0x65, 1) INT_EQUAL (register, 0x20, 4) , (const, 0x0, 4)
(register, 0x66, 1) COPY (register, 0x68, 1)
(register, 0x67, 1) COPY (register, 0x63, 1)
```



#Final Exercises

- The final exercises for this course will be pushed shortly after class today
 - `Session-four/exercises/crackmes.one`
- Exercises have been pulled from `crackmes.one`
 - They are of ascending difficulty
 - Use this site for more reversing practice!
- Source for the exercises will be released after office hours this week



#Wrap Up

- This will be the final video for this series
 - Thank you for watching and participating!
- If you have questions or commentary, please let us know for the office hour!
 - You can also fill out the google form to give us more feedback
- Feel free to use the course chatroom to ask RE questions and Ghidra questions in the future



#Questions

?

