Three Heads Are Better Than One: Mastering NSA's Ghidra Reverse Engineering Tool

Alexei Bulazel @0xAlexei
Jeremy Blackthorne @0xJeremy

github.com/0xAlexei/INFILTRATE2019
Disclaimer

This material is based on the publicly released Ghidra, there is no classified information in this presentation.
Alexei Bulazel  @0xAlexei

- Senior Security Researcher at River Loop Security

- Research presentations and publications:
  - Presentations at REcon (MTL & BRX), SummerCon, DEFCON, Black Hat, etc.
  - Academic publications at USENIX WOOT and ROOTS
  - Cyber policy in Lawfare, etc.

- Collaborated with Jeremy on research at RPI, MIT Lincoln Laboratory, and Boston Cybernetics Institute

- Proud RPISEC alumnus
Jeremy Blackthorne  @0xJeremy

- Instructor at the Boston Cybernetics Institute
- PhD candidate at RPI focused on environmental keying
- Former researcher at MIT Lincoln Laboratory
- United States Marine Corps 2002 - 2006
- RPISEC alumnus
Outline

1. Intro
2. Interactive Exercises
   a. Manual Static Analysis
   b. Scripting Ghidra
3. P-Code & SLEIGH
4. Discussion
5. Conclusion
Participating

1. Install OpenJDK 11, add its `bin` directory to your `PATH`
   - `jdk.java.net/11`

2. Download Ghidra
   - `ghidra-sre.org`
   - `github.com/NationalSecurityAgency/ghidra/releases`

3. Download our demo scripts and binaries
   - `github.com/0xAlexei/INFILTRATE2019`
Ghidra

- Java-based interactive reverse engineering tool developed by US National Security Agency - similar in functionality to IDA Pro, Binary Ninja, etc...
  - Static analysis only currently, debugger support promised to be coming soon
  - Runs on Mac, Linux, and Windows
- All credit for creating Ghidra goes to the developers at NSA
- Released open source at RSA in March 2019
  - 1.2M+ lines of code
- NSA has not discussed the history of the tool, but comments in source files go as far back as February 1999

$ grep -r "1999" * --include *.java
src/Generic-src/ghidra/util/exception/NotYetImplementedException.java: * @version 1999/02/05
src/SoftwareModeling-src/ghidra/program/model/address/AddressOutOfBoundsException.java: * @version 1999-03-31
src/SoftwareModeling-src/ghidra/program/model/scalar/ScalarOverflowException.java: * @version 1999-03-31
src/SoftwareModeling-src/ghidra/program/model/scalar/ScalarFormatException.java: * @version 1999/02/04
Outline

1. Intro
2. Interactive Exercises
   a. Manual Static Analysis
   b. Scripting Ghidra
3. P-Code & SLEIGH
4. Discussion
5. Conclusion
Default UI - CodeBrowser
Default UI - Program Trees
Default UI - Symbol Tree
Default UI - Data Type Manager
Default UI - Listing (Disassembly)
Default UI - Decompiler
Disassembly with P-Code
Decompilation Across Architectures - SPARC
Strings

```
.text("\%s: Error: Couldn't open \%s\n", _AppName);

.text(8);
/* WARNING: Subroutine does not return */

.initialize_bomb();

.text("Welcome to my fiendish little bomb. You
.text("which to blow yourself up. Have a nice day!

pcVar1 = _read_line();
.phase_1(pcVar1);

.phase_defused();

.text("Phase 1 defused. How about the next one?

pcVar1 = _read_line();
```
### Type System / Structure Recovery

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Mnemonic</th>
<th>DataType</th>
<th>Name</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>char *</td>
<td>char *</td>
<td>gr_name</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>char *</td>
<td>char *</td>
<td>gr_passwd</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>__gid_t</td>
<td>__gid_t</td>
<td>gr_gid</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>char **</td>
<td>char *</td>
<td>gr_mem</td>
<td></td>
</tr>
</tbody>
</table>

**Search:**

**Name:** group

**Description:**

**Category:** busybox/grp.h

**Size:** 16

**Alignment:** 4
Customize Appearance

Edit > Tool Options → Filter: Fonts
The Undo Button!
Version Tracker

- Feature for porting RE symbols, annotations, etc. between incrementally updated versions of the same program.

- In our experience, not well suited for quick 1-day discovery in patch analysis.
  - Use Diaphora or BinDiff for this purpose.
## Version Tracker - Selecting Correlation Algorithms

### Select Correlation Algorithm(s)

<table>
<thead>
<tr>
<th>S...</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>Exact Data Match</td>
<td>Compares data by iterating over all defined data meeting the minimum size requirement in the source program.</td>
</tr>
<tr>
<td>✓</td>
<td>Exact Function Bytes Match</td>
<td>Compares code by hashing bytes, looking for identical functions. It reports back any that have ONLY ONE separate instance.</td>
</tr>
<tr>
<td>✓</td>
<td>Exact Function Instructions Match</td>
<td>Compares code by hashing instructions, looking for identical functions. It reports back any that have ONLY ONE separate instance.</td>
</tr>
<tr>
<td>✓</td>
<td>Exact Function Mnemonics Match</td>
<td>Compares code by hashing instructions mnemonics, looking for identical functions. It reports back any that have ONLY ONE separate instance.</td>
</tr>
<tr>
<td></td>
<td>Exact Symbol Name Match</td>
<td>Compares symbols by iterating over all defined function and data symbols meeting the minimum size requirement in the source program.</td>
</tr>
<tr>
<td></td>
<td>Data Reference Match</td>
<td>Matches functions by the accepted data matches they have in common.</td>
</tr>
<tr>
<td></td>
<td>Combined Function and Data Reference Match</td>
<td>Matches functions based on the accepted data and function matches they have in common.</td>
</tr>
<tr>
<td></td>
<td>Function Reference Match</td>
<td>Matches functions by the accepted function matches they have in common.</td>
</tr>
<tr>
<td></td>
<td>Duplicate Data Match</td>
<td>Compares data by iterating over all defined data meeting the minimum size requirement in the source program.</td>
</tr>
<tr>
<td></td>
<td>Duplicate Function Instructions Match</td>
<td>Compares code by hashing instructions (masking off operands), looking for identical functions. It reports back any that have ONLY ONE separate instance.</td>
</tr>
<tr>
<td></td>
<td>Duplicate Exact Symbol Name Match</td>
<td>Compares symbols by iterating over all defined function and data symbols meeting the minimum size requirement in the source program.</td>
</tr>
<tr>
<td></td>
<td>Similar Symbol Name Match</td>
<td>Compares symbols by iterating over all defined function and data symbols meeting the minimum size requirement in the source program.</td>
</tr>
<tr>
<td></td>
<td>Similar Data Match</td>
<td>Compares data by iterating over all defined data meeting the minimum size requirement in the source program.</td>
</tr>
</tbody>
</table>
## Version Tracker - Function Name Ported

### Version Tracking Matches

<table>
<thead>
<tr>
<th>Tag</th>
<th>Session</th>
<th>ID</th>
<th>Type</th>
<th>Score</th>
<th>Confidence</th>
<th>Votes</th>
<th># Co.</th>
<th>Mul.</th>
<th>Source Name</th>
<th>Source Label</th>
<th>Source Add.</th>
<th>Source Dest.</th>
<th>Source Dest.</th>
<th>Dest Label</th>
<th>Dest Address</th>
<th>Source L</th>
<th>Dest L</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Function</td>
<td>0.000</td>
<td>0.000</td>
<td>1</td>
<td>0</td>
<td>Global</td>
<td>deployGadget</td>
<td>004118F0</td>
<td>Global</td>
<td>deployGadget</td>
<td>004118c1</td>
<td>250</td>
<td>261</td>
<td>Implied Match</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Version Tracking Markup Items

<table>
<thead>
<tr>
<th>Status</th>
<th>Source Address</th>
<th>Dest Address</th>
<th>Markup Type</th>
<th>Source Value</th>
<th>Current Dest Value</th>
<th>Original Dest Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>004118F0</td>
<td>004118c0</td>
<td>Function Name</td>
<td>deployGadget</td>
<td>deployGadget</td>
<td>FUN_004118c0</td>
</tr>
</tbody>
</table>

### Source: `deployGadget() in /WallaceSrc.exe`

```plaintext
FUNCTION
DECLARE cdecl __cdecl deployGadget(void)
  STACK-0x8d:4 local_8
  STACK-0x8b:4 local_10
  STACK-0x24:4 local_24
  STACK-0x89:4 local_18
  STACK-0x88:4 local_10
  STACK-0x19:4 local_17
  STACK-0x18:4 local_18
  STACK-0x23:4 local_17
  STACK-0x1c:4 local_18
  STACK-0x1b:4 local_17
  STACK-0x1a:4 local_17
  STACK-0x19:4 local_18
  STACK-0x18:4
  STACK-0x17:4
  STACK-0x16:4
  STACK-0x15:4
  STACK-0x14:4
  STACK-0x13:4
  STACK-0x12:4
  STACK-0x11:4
  STACK-0x10:4
  STACK-0x0f:4
  STACK-0x0e:4
  STACK-0x0d:4
  STACK-0x0c:4
  STACK-0x0b:4
  STACK-0x0a:4
  STACK-0x09:4
  STACK-0x08:4
  STACK-0x07:4
  STACK-0x06:4
  STACK-0x05:4
  STACK-0x04:4
  STACK-0x03:4
  STACK-0x02:4
  STACK-0x01:4
  STACK-0x00:4

deployGadget XREF[1]: deployGadget:0041117
```

### Destination: `FUN_004118c0 in /WallaceVersion2.exe`

```plaintext
FUNCTION
DECLARE cdecl __stdcall deployGadget(void)
  STACK-0x8d:4 local_8
  STACK-0x8b:4 local_10
  STACK-0x24:4 local_24
  STACK-0x89:4 local_18
  STACK-0x88:4 local_10
  STACK-0x19:4 local_17
  STACK-0x18:4 local_18
  STACK-0x23:4 local_17
  STACK-0x1c:4 local_18
  STACK-0x1b:4 local_17
  STACK-0x1a:4 local_17
  STACK-0x19:4 local_18
  STACK-0x18:4
  STACK-0x17:4
  STACK-0x16:4
  STACK-0x15:4
  STACK-0x14:4
  STACK-0x13:4
  STACK-0x12:4
  STACK-0x11:4
  STACK-0x10:4
  STACK-0x0f:4
  STACK-0x0e:4
  STACK-0x0d:4
  STACK-0x0c:4
  STACK-0x0b:4
  STACK-0x0a:4
  STACK-0x09:4
  STACK-0x08:4
  STACK-0x07:4
  STACK-0x06:4
  STACK-0x05:4
  STACK-0x04:4
  STACK-0x03:4
  STACK-0x02:4
  STACK-0x01:4
  STACK-0x00:4

deployGadget XREF[1]: deployGadget:0041117
```
1:1 comparison of program memory ranges, only helpful if you had two annotated Ghidra databases for the same binary, or a version that had been statically patched.
Program Differences

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Machine Code</th>
<th>Assembly</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00411870</td>
<td>55</td>
<td>PUSH EBP</td>
<td>EBP,ESP</td>
<td></td>
</tr>
<tr>
<td>00411871</td>
<td>8b, ec</td>
<td>MOV ESP, -0x1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00411875</td>
<td>48, 2e, 4b, 41, 00</td>
<td>PUSH LAB_00414b2e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>004118fa</td>
<td>64, 01, 00, 00, 00</td>
<td>MOV EAX, [ESI+0x0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00411900</td>
<td>50</td>
<td>PUSH EAX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00411901</td>
<td>81, ec, f4</td>
<td>SUB ESP, 0xf4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00411907</td>
<td>53</td>
<td>PUSH EBX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00411908</td>
<td>56</td>
<td>PUSH ESI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00411909</td>
<td>57</td>
<td>PUSH EDI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0041190a</td>
<td>8d, bd, 00</td>
<td>LEA EDI=gadgetLocal, [0x0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Function Call Graph
Decompiler Slicing

- Decompiler calculates data flow during auto-analysis.
- Users can right-click on variables to view def-use chain and forward / backward slices.
- Menu bar “Select” options allow users to trace flows to / from given points.
Decompiler Slicing

Middle Click

Highlight Forward Inst Slice
Decompiler Slicing
Outline

1. Intro
2. Interactive Exercises
   a. Manual Static Analysis
   b. Scripting Ghidra
3. P-Code & SLEIGH
4. Discussion
5. Conclusion
Scripting With Ghidra

- Available in Java (natively) and Python (via Jython)
- Can be run with interactive GUI or in headless mode
- Ghidra comes with 230+ scripts pre-installed
  - Educational examples
  - Code patching
  - Import / export
  - Analysis enhancements
    - Windows, Mac, Linux, VXWorks
    - PE, ELF, Mach-O, COFF
    - x86, MIPS, ARM/THUMB, 8051, etc...
Ghidra APIs

FlatProgramAPI
- Simple “flattened” API for Ghidra scripting
- Programmatic access to common tasks
  - query / modify / iterate / create / delete - functions / data / instructions / comments
- Mostly doesn’t require the use of Java objects
- Stable

Ghidra Program API
- More complex rich API for deeper scripting
- Object-oriented (Program, Memory, Function, Instruction, etc...)
- Utility functions help with common scripting tasks
- UI scripting / interactivity
- Prone to change between versions
API Highlights

Rich Scripting Interface
- Programmatic access to binary file formats
- P-code interaction
- Decompiler API
- C header parsing
- Interface for graphing (implementation not included)
- Cyclomatic complexity

Common Utilities Included
- UI windows
- Assembly
- Data serialization
- String manipulation
- Hashing
- Search / byte matching
- XML utilities
Eclipse Integration

- Ghidra has built-in Eclipse integration, via its “GhidraDev” Eclipse plugin

- **NOTE:** For these exercises, we’ll be using Ghidra’s built-in basic editor - don’t waste time trying to get Eclipse set up during this workshop
Scripting Demos

1. Hello World
   a. Java
   b. Python
2. Crypto Constant Search
3. Cyclomatic Complexity
4. Xor with Python
Importing Demo Scripts

Click the “Display Script Manager” button to open the Script Manager Window.
**Importing Demo Scripts**

Click the "Display Script Manager" button to open the Script Manager Window.

Click the "Script Directories" button on the Script to open the "Display Script Manager" button to open the Script Manager Window.
Importing Demo Scripts

Click the “Display Script Manager” button to open the Script Manager Window

Click the “Script Directories” button on the Script Click the “Display Script Manager” button to open the Script Manager Window to choose the script directory

Click the green plus to open the file chooser, choose the script directory
Running Script Demos

Find the “INFLTRATE” folder in the script manager.
Running Script Demos

Find the “INFLTRATE” folder in the script manager

Choose a script and click “Run Script” to run
Running Script Demos

Find the “INfiltrate” folder in the script manager.

Choose a script and click “Run Script” to run.

Make sure the “Console” window is open if you want to see output.
DEMO: Hello World

- A simple script to print “Hello World” and then iterate over all functions in the program, printing out their names

- HelloWorld.java
- HelloWorld.py

```java
import ghidra.app.script.GhidraScript;

public class HelloWorld extends GhidraScript {
    public void run() throws Exception {
        println("Hello world");
        Function currentFunc = getFirstFunction();
        while (currentFunc != null){
            println(currentFunc.getName());
            currentFunc = getFunctionAfter(currentFunc);
        }
    }
}
```
DEMO: Crypto Search

- Find MD5 constants present in a binary, report offset and function name
- Take binary endianness into account automatically without user specification
- `CryptoConstantsSearch.java`
DEMO: Crypto Search

CryptoConstantsSearch.java> Running...
MD5 Constant 0x67452301 found at 0x100001060 (n/a)
MD5 Constant 0xefcda89 found at 0x100001064 (n/a)
MD5 Constant 0x98badcfe found at 0x100001068 (n/a)
MD5 Constant 0x10325476 found at 0x10000106c (n/a)
CryptoConstantsSearch.java> Finished!
DEMO: Calculating Cyclomatic Complexity

- Leverage Ghidra’s API for calculating cyclomatic complexity* to easily analyze a whole program
- Pop a GUI window if running interactively, else print to the terminal in headless mode
- `ComputeCyclomaticComplexityForAllFunctions.java`

* Cyclomatic complexity is a measure of the number of unique paths which may be taken through a given function. It can be helpful in finding complex functions likely to have vulnerabilities, e.g., complex parsing routines or state machines.
DEMO: Calculating Cyclomatic Complexity
DEMO: Python Scripting

- Ghidra can run Python in a Jython environment, the Ghidra Java API is exposed to Python
- This script takes a user address selection and XORs the bytes in place with 0x41
- XorMemoryScript.py
### DEMO: Python Scripting

<table>
<thead>
<tr>
<th>Addresses</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>100001ce8 00</td>
<td>?? 00h</td>
</tr>
<tr>
<td>100001ce9 00</td>
<td>?? 00h</td>
</tr>
<tr>
<td>100001cea 24</td>
<td>?? 24h $</td>
</tr>
<tr>
<td>100001ceb 2f</td>
<td>?? 2Fh /</td>
</tr>
<tr>
<td>100001cec 22</td>
<td>?? 22h &quot;</td>
</tr>
<tr>
<td>100001ced 33</td>
<td>?? 33h 3</td>
</tr>
<tr>
<td>100001cee 38</td>
<td>?? 38h 8</td>
</tr>
<tr>
<td>100001cef 31</td>
<td>?? 31h 1</td>
</tr>
<tr>
<td>100001cfe 35</td>
<td>?? 35h 5</td>
</tr>
<tr>
<td>100001cf0 24</td>
<td>?? 24h $</td>
</tr>
<tr>
<td>100001cf1 25</td>
<td>?? 25h %</td>
</tr>
<tr>
<td>100001cf2 61</td>
<td>?? 61h a</td>
</tr>
<tr>
<td>100001cf3 2c</td>
<td>?? 2Ch ,</td>
</tr>
<tr>
<td>100001cf4 24</td>
<td>?? 24h $</td>
</tr>
<tr>
<td>100001cf5 32</td>
<td>?? 32h 2</td>
</tr>
<tr>
<td>100001cf6 32</td>
<td>?? 32h 2</td>
</tr>
<tr>
<td>100001cf7 20</td>
<td>?? 20h</td>
</tr>
<tr>
<td>100001cf8 26</td>
<td>?? 26h &amp;</td>
</tr>
<tr>
<td>100001cf9 24</td>
<td>?? 24h $</td>
</tr>
<tr>
<td>100001cfa 00</td>
<td>?? 00h</td>
</tr>
<tr>
<td>100001cfc 00</td>
<td>?? 00h</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addresses</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>100001ce8 00</td>
<td>?? 00h</td>
</tr>
<tr>
<td>100001ce9 00</td>
<td>?? 00h</td>
</tr>
<tr>
<td>100001cea 65</td>
<td>?? 65h e</td>
</tr>
<tr>
<td>100001ceb 6e</td>
<td>?? 66h n</td>
</tr>
<tr>
<td>100001cec 63</td>
<td>?? 63h c</td>
</tr>
<tr>
<td>100001ced 72</td>
<td>?? 72h r</td>
</tr>
<tr>
<td>100001cee 79</td>
<td>?? 79h y</td>
</tr>
<tr>
<td>100001cef 70</td>
<td>?? 70h p</td>
</tr>
<tr>
<td>100001cfe 74</td>
<td>?? 74h t</td>
</tr>
<tr>
<td>100001cf0 65</td>
<td>?? 65h e</td>
</tr>
<tr>
<td>100001cf1 64</td>
<td>?? 64h d</td>
</tr>
<tr>
<td>100001cf2 20</td>
<td>?? 20h</td>
</tr>
<tr>
<td>100001cf3 6d</td>
<td>?? 6Dh m</td>
</tr>
<tr>
<td>100001cf4 65</td>
<td>?? 65h e</td>
</tr>
<tr>
<td>100001cf5 73</td>
<td>?? 73h s</td>
</tr>
<tr>
<td>100001cf6 73</td>
<td>?? 73h s</td>
</tr>
<tr>
<td>100001cf7 61</td>
<td>?? 61h a</td>
</tr>
<tr>
<td>100001cf8 67</td>
<td>?? 67h g</td>
</tr>
<tr>
<td>100001cfa 65</td>
<td>?? 65h e</td>
</tr>
<tr>
<td>100001cfa 00</td>
<td>?? 00h</td>
</tr>
<tr>
<td>100001cfc 00</td>
<td>?? 00h</td>
</tr>
</tbody>
</table>
Outline

1. Intro
2. Interactive Exercises
   a. Manual Static Analysis
   b. Scripting Ghidra
3. P-Code & SLEIGH
4. Discussion
5. Conclusion
P-Code

- Ghidra’s intermediate language
  - Dates back to at least 2005 according to documentation
- Code for different processors can be lifted into p-code, data-flow analysis and decompilation can then run over the p-code
- Pseudo-assembly, represents lifted instructions as small atomic operations without side-effects
- Built-in floating point support

```plaintext
MOVSD       RAX, EDX
            (register, 0x0, 8) = INT_SEXT (register, 0x10, 4)
LEA          RAX, [RAX + RAX*0x4]
            (unique, 0x660, 8) = INT_MULTIPLY (register, 0x0, 8), (const, 0x4, 8)
            (unique, 0x680, 8) = INT_ADD (register, 0x0, 8), (unique, 0x660, 8)
            (register, 0x0, 8) = COPY (unique, 0x680, 8)
```
P-Code Design

- The language is machine independent.
- The language is designed to model general purpose processors.
- Instructions operate on user defined registers and address spaces.
- All data is manipulated explicitly. Instructions have no indirect effects.
- Individual p-code operations mirror typical processor tasks and concepts.

Quoted from docs/languages/html/sleigh.html

Processor to p-code modeling:
- RAM → *address space*
- Register → *varnode*
- Instruction → *operation*
<table>
<thead>
<tr>
<th>Category</th>
<th>P-Code Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Moving</td>
<td>COPY, LOAD, STORE</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>INT_ADD, INT_SUB, INT_CARRY, INT_SCARRY, INT_SBORROW, INT_2COMP, INT_MULT, INT_DIV, INT_SDIV, INT_REM, INT_SREM</td>
</tr>
<tr>
<td>Logical</td>
<td>INT_NEGATE, INT_XOR, INT_AND, INT_OR, INT_LEFT, INT_RIGHT, INT_SRIGHT</td>
</tr>
<tr>
<td>Int Comparison</td>
<td>INT_EQUAL, INT_NOTEQUAL, INT_SLESS, INT_SLESSEQUAL, INT_LESS, INT_LESSEQUAL</td>
</tr>
<tr>
<td>Boolean</td>
<td>BOOL_NEGATE, BOOL_XOR, BOOL_AND, BOOL_OR</td>
</tr>
<tr>
<td>Floating Point</td>
<td>FLOAT_ADD, FLOAT_SUB, FLOAT_MULT, FLOAT_DIV, FLOAT_NEG, FLOAT_ABS, FLOAT_SQRT, FLOAT_NAN</td>
</tr>
<tr>
<td>FP Compare</td>
<td>FLOAT_EQUAL, FLOAT_NOTEQUAL, FLOAT_LESS, FLOAT_LESSEQUAL</td>
</tr>
<tr>
<td>FP Conversion</td>
<td>INT2FLOAT, FLOAT2FLOAT, TRUNC, CEIL, FLOOR, ROUND</td>
</tr>
<tr>
<td>Branching</td>
<td>BRANCH, CBRANCH, BRANCHIND, CALL, CALLIND, RETURN</td>
</tr>
<tr>
<td>Extension / Truncation</td>
<td>INT_ZEXT, INT_SEXT, PIECE, SUBPIECE</td>
</tr>
</tbody>
</table>
DEMO: Source-Sink Analysis

- Use Ghidra p-code and the decompiler’s analysis to identify the sources for values passed to function calls of interest (`malloc`), particularly function calls accepting user input.

- *Solving* for the actual arguments requires a *solver*, this is a much simpler analysis that can empower a human analyst to hone in on interesting calls.

- Start at the varnode for each argument to `malloc`, then trace back to the p-code operation that it’s derived from:
  - From there, recursively trace back the p-code operation(s) defining the varnode(s) that define that the inputs to those operations.

- At function call sites, trace in, and find how the returned values are derived.

- When a parameter is used, trace back to call sites which set the parameter.
SLEIGH

- Ghidra’s language for describing instruction sets to facilitate RE
- **Disassembly**: translate bit-encoded machine instructions into human-readable assembly language statements
- **Semantics**: translate machine instructions into p-code instructions (one-to-many) for decompilation, analysis, and emulation
- Based off of SLED (Specification Language for Encoding and Decoding), a 1997 academic IL
SLEIGH Example - x86 JMP rel8

Raw bytes: 0xEB 0x03

x86 instruction: JMP $+5
SLEIGH Example - x86  JMP rel8

Raw bytes: 0xEB 0x03

x86 instruction:  JMP $+5
SLEIGH Example - x86  JMP rel8

Raw bytes: 0xEB 0x03

x86 instruction:  JMP $+5

SLEIGH:
:JMP rel8 is vexMode=0 & byte=0xeb; rel8 {
  goto rel8;
}

00401f16  eb 03  JMP  LAB_00401f1b
  BRANCH *[ram]0x401f1b:8
SLEIGH Example - x86 \texttt{JMP rel8}

Raw bytes: \texttt{0xEB 0x03}

x86 instruction: \texttt{JMP \$+5}

\begin{verbatim}
rel8: reloc is simm8 [ reloc=inst_next+simm8; ] {
    export *[ram]:$(SIZE) reloc;
}
\end{verbatim}

SLEIGH:

\begin{verbatim}
:JMP rel8 is vexMode=0 & byte=0xeb; rel8 {
    goto rel8;
}
\end{verbatim}

\begin{verbatim}
00401f16 eb 03 \texttt{JMP LAB_00401f1b}
\end{verbatim}

\begin{verbatim}
BRANCH *[ram]0x401f1b:8
\end{verbatim}
SLEIGH Example - x86  JMP  rel8

Raw bytes: 0xEB  0x03
x86 instruction:  JMP  $+5

SLEIGH:
:JMP  rel8  is  vexMode=0  &  byte=0xeb;  rel8  {
    goto  rel8;
}

00401f16  eb 03  JMP  LAB_00401f1b  BRANCH  *[ram]0x401f1b:8
SLEIGH Example - x86  JMP rel8

Raw bytes: 0xEB 0x03

x86 instruction:  JMP $+5

SLEIGH:

:JMP rel8 is vexMode=0 & byte=0xeb; rel8 {
  goto rel8;
}

00401f16 eb 03  JMP      LAB_00401f1b

BRANCH *[ram]0x401f1b:8
SLEIGH Example - x86 XOR AL, imm8

Raw bytes: 0x34 0x57

x86 instruction: XOR AL, 0x57
SLEIGH Example - x86  XOR AL, imm8

Raw bytes: 0x34 0x57
x86 instruction: XOR AL, 0x57

34 57 XOR AL, 0x57

CF = COPY 0:1
OF = COPY 0:1
AL = INT_XOR AL, 0x57:1
SF = INT_SLESS AL, 0:1
ZF = INT_EQUAL AL, 0:1
SLEIGH Example - x86  XOR AL, imm8

Raw bytes: 0x34  0x57

x86 instruction:  XOR AL, 0x57

SLEIGH:
:XOR AL,imm8 is vexMode=0 & byte=0x34; AL & imm8 {
    logicalflags();
    AL = AL ^ imm8;
    resultflags( AL );
}

34 57  XOR  AL,0x57

CF = COPY 0:1
OF = COPY 0:1
AL = INT_XOR AL, 0x57:1
SF = INT_SLESS AL, 0:1
ZF = INT_EQUAL AL, 0:1
SLEIGH Example - x86 XOR AL, imm8

Raw bytes: 0x34 0x57
x86 instruction: XOR AL, 0x57

SLEIGH:
:XOR AL,imm8 is vexMode=0 & byte=0x34; AL & imm8 {
    logicalflags();
    AL = AL ^ imm8;
    resultflags( AL );
}

macro logicalflags() {
    CF = 0;
    OF = 0;
}

macro resultflags(result) {
    SF = result s < 0;
    ZF = result == 0;
    # PF, AF not implemented
}
SLEIGH Example - x86 XOR AL, imm8

Raw bytes: 0x34 0x57

x86 instruction: XOR AL, 0x57

SLEIGH:
:XOR AL,imm8 is vexMode=0 & byte=0x34; AL & imm8 {
    logicalflags();
    AL = AL ^ imm8;
    resultflags( AL );
}

macro logicalflags() {
    CF = 0;
    OF = 0;
}

macro resultflags(result) {
    SF = result s < 0;
    ZF = result == 0;
    # PF, AF not implemented
}
SLEIGH Example - x86  XOR AL, imm8

Raw bytes: 0x34 0x57

x86 instruction:  XOR AL, 0x57

SLEIGH:
:XOR AL,imm8 is vexMode=0 & byte=0x34; AL & imm8 {
    logicalflags();
    AL = AL ^ imm8;
    resultflags( AL );
}

macro logicalflags() {
    CF = 0;
    OF = 0;
}

macro resultflags(result) {
    SF = result s < 0;
    ZF = result == 0;
    # PF, AF not implemented
}
SLEIGH Example - x86  XOR AL, imm8

Raw bytes: 0x34 0x57

x86 instruction:  XOR AL, 0x57

SLEIGH:
:XOR AL,imm8 is vexMode=0 & byte=0x34; AL & imm8 {
  logicalflags();
  AL = AL ^ imm8;
  resultflags( AL );
}

macro logicalflags() {
  CF = 0;
  OF = 0;
}

macro resultflags(result) {
  SF = result s < 0;
  ZF = result == 0;
  # PF, AF not implemented
}
SLEIGH Example - x86 XOR AL, imm8

Raw bytes: 0x34 0x57

x86 instruction: XOR AL, 0x57

SLEIGH:

:XOR AL,imm8 is vexMode=0 & byte=0x34; AL & imm8 {
    logicalflags();
    AL = AL ^ imm8;
    resultflags( AL );
}

macro logicalflags() {
    CF = 0;
    OF = 0;
}

macro resultflags(result) {
    SF = result s < 0;
    ZF = result == 0;
    # PF, AF not implemented
}
SLEIGH Example - x86  RDTSC

Raw bytes: 0x0F 0x31

x86 instruction:  RDTSC
SLEIGH Example - x86  RDTSC

Raw bytes: 0x0F 0x31

x86 instruction:  RDTSC
SLEIGH Example - x86  RDTSC

Raw bytes: 0x0F 0x31

x86 instruction: RDTSC

SLEIGH:
:RDTSC is vexMode=0 & byte=0xf; byte=0x31 {
    tmp:8 = rdtsc();
    EDX = tmp(4);
    EAX = tmp(0);
}

$U9c60:8 = CALLOTHER "rdtsc"
EDX = SUBPIECE $U9c60, 4:4
EAX = SUBPIECE $U9c60, 0:4
SLEIGH Example - x86 RDTSC

Raw bytes: 0x0F 0x31
x86 instruction: RDTSC

SLEIGH:
:RDTSC is vexMode=0 & byte=0xf; byte=0x31 {
  tmp:8 = rdtsc();
  EDX = tmp(4);
  EAX = tmp(0);
}

define pcodeop rdtsc;

$U9c60:8 = CALLOThER "rdtsc"
EDX = SUBPIECE $U9c60, 4:4
EAX = SUBPIECE $U9c60, 0:4
SLEIGH Example - x86 RDTSC

Raw bytes: 0x0F 0x31

x86 instruction: RDTSC

SLEIGH:
:RDTSC is vexMode=0 & byte=0xf; byte=0x31 {
  tmp:8 = rdtsc();
  EDX = tmp(4);
  EAX = tmp(0);
}

define pcodeop rdtsc;

```
$U9c60:8 = CALL OTHER "rdtsc"
EDX = SUBPIECE $U9c60, 4:4
EAX = SUBPIECE $U9c60, 0:4
```
SLEIGH Example - x86 RDTSC

Raw bytes: 0x0F 0x31

x86 instruction: RDTSC

SLEIGH:
:RDTSC is vexMode=0 & byte=0xf; byte=0x31 {
    tmp:8 = rdtsc();
    EDX = tmp(4);
    EAX = tmp(0);
}
define pcodeop rdtsc;

$U9c60:8 = CALLOTHER "rdtsc"
EDX = SUBPIECE $U9c60, 4:4
EAX = SUBPIECE $U9c60, 0:4
SLEIGH Example - x86  RDTSC

Raw bytes: 0x0F 0x31

x86 instruction:  RDTSC

SLEIGH:
:RDTSC is vexMode=0 & byte=0xf; byte=0x31 {
  tmp:8 = rdtsc();
  EDX = tmp(4);
  EAX = tmp(0);
}

define pcodeop rdtsc;

$U9c60:8 = CALLOTHE "rdtsc"
EDX = SUBPIECE $U9c60, 4:4
EAX = SUBPIECE $U9c60, 0:4
Outline

1. Intro
2. Interactive Exercises
   a. Manual Static Analysis
   b. Scripting Ghidra
3. P-Code & SLEIGH
4. Discussion
5. Conclusion
Other Major Features

- Multi-user collaboration
- Version tracking
- Extensibility for new binary loaders and architectures
- Headless mode
- GhidraDev Eclipse plugin
- Debugger promised at RSA
- Undocumented p-code emulator
IDA vs Binary Ninja vs Ghidra

**IDA**
- Maturity
- Windows support
- Decompiler
- Existing corpus of powerful plugins
- Debugger
- Support for paid customers
- Well tested
- Industry standard

**Binary Ninja**
- Innovation and modern design
- Program analysis features (SSA)
- Multi-level IL
- Rich API
- Embeddable
- Python-native scripting
- Clean modern UI
- Community

**Ghidra**
- Maturity
- Embedded support
- Decompiler
- Massive API
- Documentation
- Breath of features
- Collaboration
- Version tracking
- Price and open source extensibility
Decompiler - IDA Hex-Rays vs Ghidra

**IDA Hex-Rays**
- Optional add-on for IDA for IDA
- Microcode-based
- Supports limited architectures
- Better built-in support for Windows
- Variables, data, and functions can be xrefed from decompiler
- Variables can be mapped
- Variable representation can be changed in the decompiler (decimal, hex, char immediate, etc)
- Click to highlight

**Ghidra Decompiler**
- Deeply integrated with Ghidra
- P-code based
- Supports all architectures
- No way to xref from decompiler
- Produces fewer `goto` statements and seemingly more idiomatic C
- Built in program analysis features, e.g., slicing and data flow
- Variables cannot be mapped
- Variable representation cannot be changed in the decompiler
- *Middle click* to highlight
ILs - Binary Ninja vs Ghidra

Binary Ninja
- Multi-level: LLIL, MLIL, forthcoming HLIL
- Machine consumable and human readable
- SSA form
- Designed in light of years of program analysis research
- Feels nicer to work with
- Deferred flag calculations

Ghidra
- Single level p-code, but can be enhanced by decompiler analysis
- Designed for machine consumption first, not human readability
- Uses SSA during decompilation, but raw p-code is not SSA
- Design origins based off of program analysis research from 20+ years ago
ILs - Binary Ninja vs Ghidra

phase_4:
```c
rsp = rsp - 0x18
rcx = rsp + 0xc {var_c}
rdx = rsp + 8 {var_10}
esi = 0x4025cf {"%d %d"}
eax = 0
call(__isoc99_sscanf)
if (eax != 2) then 7 @ 0x401035 else 9 @ 0x401033
```

LLIL

phase_4:
```c
int32_t* rcx = &var_c
int32_t* rdx = &var_10
rax = __isoc99_sscanf(arg1, 0x4025cf, rdx, rcx) {"%d %d"}
if (rax.eax != 2) then 4 @ 0x401035 else 6 @ 0x401033
```

MLIL
We Like Ghidra For...

- Scripting reverse engineering
- Firmware / embedded systems analysis
- Analysis of software that Hex-Rays can’t decompile
- Collaborative long-term professional RE
- Professional reversing at a computer workstation with multiple monitors, full keyboard with function keys, mouse with middle click and scroll wheel, etc...
Scripting - Java vs Python

- Java will catch errors at compile time, Ghidra’s API is highly object-oriented and benefits from this

- Complex Python scripts feel like binding together Java API calls with Python control flow and syntax

- **Recommended workflow:** prototype and experiment with APIs / objects in the Python interpreter, write final code in Java
For Reverse Engineers, By Reverse Engineers

- Built for multi-monitor use
- "Moving ants" highlight on control flow graphs
- Configurable "tool" views
- Hotkeys mappable to actions and scripts
- Right click > "extract and import"
- Processor manual integration
- Undo button
- Import directly from zip file
- Snapshot views
- Configurable listings
- Version tracker

- Project-based multi-binary RE
- F1 to open help on whatever the mouse is pointing at
- File System browser
- Highly configurable assembly code listing
- Data flow analysis built into UI
- Embedded image detection
- Search for matching instructions
- Unique windows
  - Checksum Generator
  - Disassembled View
  - Data Type Preview
  - Function Tags
  - Symbol tree
Contributing to Ghidra

- Ghidra code is available on Github
  - Apache License 2.0
- NSA has been responsive to community questions and bug reports posted on Github
  - The agency has already published two minor-version updates

Official site: ghidra-sre.org
Open source: github.com/NationalSecurityAgency/ghidra
github.com/NationalSecurityAgency/ghidra-data
Outline

1. Intro
2. Interactive Exercises
   a. Manual Static Analysis
   b. Scripting Ghidra
3. P-Code & SLEIGH
4. Discussion
5. Conclusion
Conclusion

Ghidra is a powerful binary reverse engineering tool built by the US National Security Agency

- For reverse engineers, by reverse engineers
- Interactive and headless scripting
- Built for program analysis
- We have yet to see what the community will do with Ghidra, this is just the beginning

Demo material: github.com/0xAlexei/INFILTRATE2019
IDA keybindings: github.com/JeremyBlackthorne/Ghidra-Keybindings
Official NSA sites: github.com/NationalSecurityAgency/ghidra
ghidra-sre.org
Ghidra training: ringzer0.training

Acknowledgements:
- NSA’s Ghidra team
- Rob Joyce
- Rolf Rolles
- Evan Jensen
- Sophia d’Antoine
- Dave Aitel and the INFILTRATE team

@0xAlexei / @0xJeremy
Architecture Support

- Supports a variety of common desktop, embedded, and VM architectures
- Can handle unique compiler idioms and CPU modes
- Users can extend Ghidra with their own custom processor modules
- Ghidra can decompile anything it can disassemble
Documentation

- Help > Contents
- **F1** or **Help** key while pointing at any field or menu option
- **docs** directory
  - JavaDoc
  - Several classes
  - P-code and SLEIGH
- Doxygen in source files
Unlike Java, which must be compiled in order to run, Python can be run inside Ghidra in an interactive REPL shell.

The shell can be helpful for exploring unfamiliar objects - Ghidra has great Python object `__str__` implementations:

```python
>>> memcpy = getFunctionAt(currentAddress)
>>> refs = getReferencesTo(memcpy.getEntryPoint())
>>> refs
array(ghidra.program.model.symbol.Reference, [From: 004006ac To: 00400510 Type: UNCONDITIONAL_CALL Op: 0 DEFAULT, From: 004006d5 To: 00400510 Type: UNCONDITIONAL_CALL Op: 0 DEFAULT])
```
Script vs. Plugin vs. Extension

- **Scripts:** do a single thing with defined start and end point

- **Plugins:** base components for everything you interact with in Ghidra, such as UI panes

- **Extensions:** sets of plugins for extended functionality, e.g., custom binary format loaders, interfaces to external tools, libraries for use by other scripts
  - **Example:** Rolf Rolles’ GhidraPAL
    [github.com/RolfRolles/GhidraPAL/releases](https://github.com/RolfRolles/GhidraPAL/releases)

Tools

- Ghidra tools are assemblies of plugins
- Ghidra comes with two tools: CodeBrowser and VersionTracker
- Tools can be configured and customized to suit unique RE needs - though currently there doesn’t seem to be much point
IDA Interoperability

Ghidra comes with importers and exporters to enable Ghidra / IDA interoperability
Decompiler Windows Structs - Hex-Rays vs Ghidra
P-Code Decompilation / Analysis

- “Raw p-code” = direct translation of one CPU instructions to p-code ops
- During decompilation, p-code is analyzed, and may be modified
  - Insertion of MULTIEQUAL instructions (SSA phi-nodes)
  - Association of parameters with CALL ops and return values with RETURN ops
  - Construction of abstract syntax tree
  - etc... - see linked documents
- The Decompiler is a C++ binary that runs on the host system

- When writing scripts interacting with p-code expect to experiment, read source code, and glean usage from example included scripts

docs/languages/html/additionalpcode.html
Ghidra/Features/Decompiler/src/decompile/cpp/docmain.hh
Links
Recommended Readings

Elias Bachaalany’s quick overview: 0xeb.net/2019/03/ghidra-a-quick-overview

Danny Quist on getting started with Ghidra: 
github.com/dannyquist/re/blob/master/ghidra/ghidra-getting-started.md

Rolf Rolles’ GhidraPAL program analysis library: 
github.com/RolfRolles/GhidraPAL

Travis Goodspeed on reversing MD380 firmware with Ghidra: 
github.com/travisgoodspeed/md380tools/wiki/GHIDRA
Additional Recommended Readings

Links to many loaders and processor modules: 
groups.google.com/forum/?utm_medium=email&utm_source=footer#!msg/sleigh/pD12wcoKUQM/rG8esJAVBAAJ

Writing a WASM Loader: habr.com/en/post/443318/

Sega Genesis loader: zznop.github.io/romhacking/2019/03/14/sega-genesis-rom-hacking-with-ghidra.html

Because Security’s first impressions, with many Ghidra video links: 
blog.because-security.com/t/ghidra-wiki/431#Firstimpress283

Elias Bachaalany’s “Daenerys” IDA Pro / Ghidra interoperability framework: 
0xeb.net/2019/03/daenerys-ida-pro-and-ghidra-interoperability-framework
Academic Work Related to SLEIGH / P-code

citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.51.360&rep=rep1&type=pdf

personales.ac.upc.edu/vmoya/docs/00825697.pdf

The New Jersey Machine-Code Toolkit (1990s)
www.cs.tufts.edu/~nr/toolkit/

See docs/languages/html/sleigh.html