ripr

Run slices of binary code from Python

Patrick Biernat
Shmoocon ‘17

Obligatory: >> `whoami`

Patrick Biernat

(super recent) Graduate of Rensselaer Polytechnic Institute

Frequent CTF-er with RPISEC

Gravitate towards pwning and reverse engineering

Obligatory: >> `whoami`

RPISEC

- CTF Team and Computer Security club at RPI
- Competition and Education
  - We play lots of CTFs
  - We hold multiple weekly meetings teaching people how to pwn
  - We teach semester long classes on *hard* and *interesting* security topics
    - Modern Binary Exploitation 2015, 2017 (Starts in a few days!!)
    - Malware Analysis (2016)
- Coolest thing I've been a part of

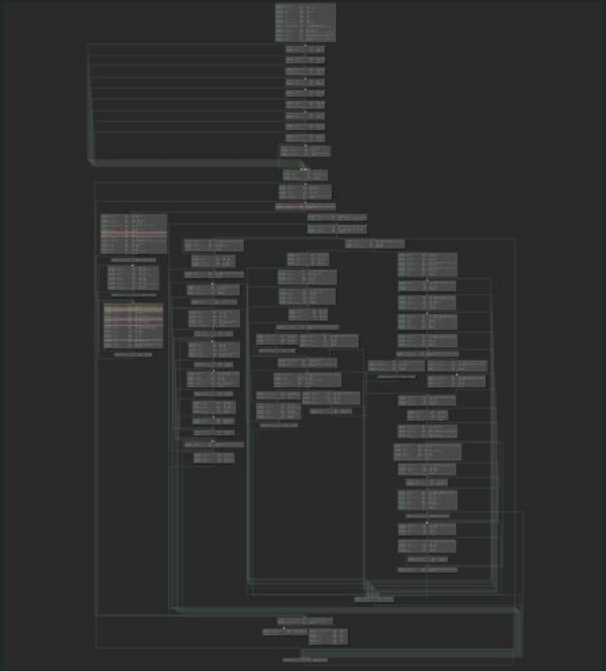
[ripr] Motivation

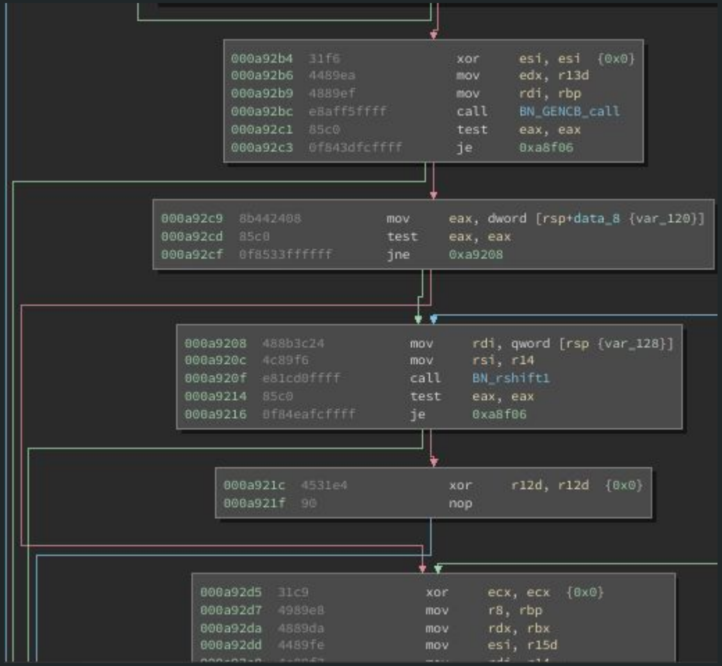
Motivation

Binary Reverse Engineering is the art of mapping a multitude of low level details into abstracted, high-level meaning.

Motivation

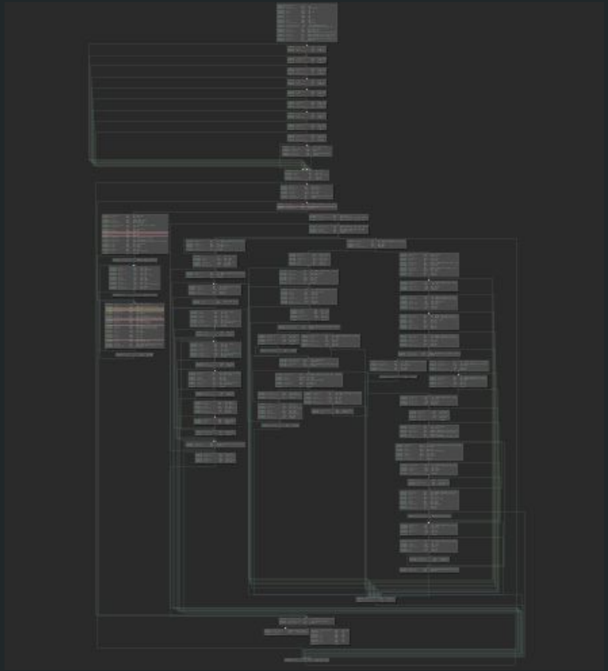
Binary Reverse Engineering is the art of mapping a multitude of low level details into abstracted, high-level meaning.

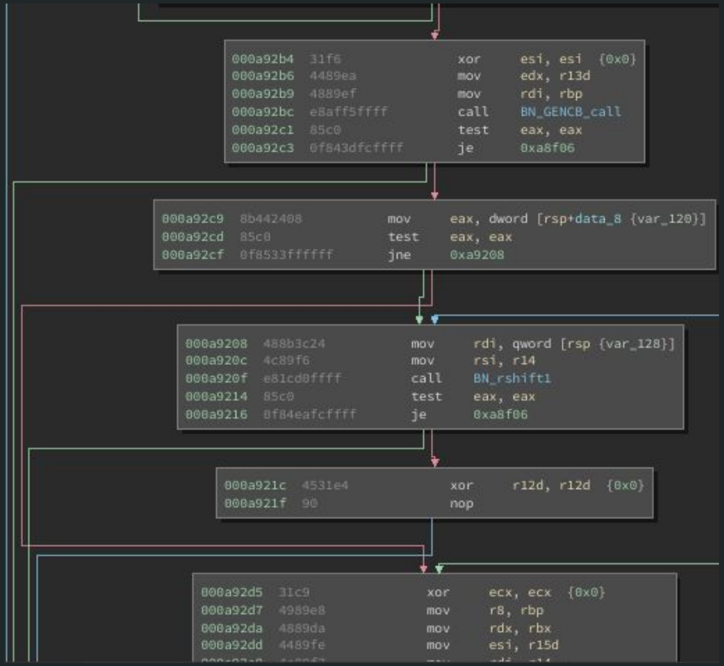




Motivation

libcrypto -- BN_generate_prime_ex





Motivation

We do this for - some - purpose:

- Vulnerability Research / Exploit Development
  - Is my code vulnerable? Can I exploit this code?
- Malware Analysis
  - Is this software Malicious? What is it doing? Can I find a unique signature?
- Interoperation
  - Can I make my “thing” work with or interact with this other, closed-source, “thing”

I want to understand *exactly* what code is running and what it is doing.

A common Problem

"I just need to implement this in my code and we'll be done"

```
generate_key:
00010464  04b02de5  str   r11, [sp,  #-0x4]!
00010468  00b08de2  add   r11, sp, #0 {var_4}
0001046c  1cd04de2  sub   sp, sp, #0x1c
00010470  10000be5  str   r0, [r11,  #-0x10] {var_14}
00010474  14100be5  str   r1, [r11,  #-0x14] {var_18}
00010478  18200be5  str   r2, [r11,  #-0x18] {var_1c}
0001047c  10201be5  ldr   r2, [r11,  #-0x10] {var_14}
00010480  14301be5  ldr   r3, [r11,  #-0x14] {var_18}
00010484  033082e0  add   r3, r2, r3
00010488  0c300be5  str   r3, [r11,  #-0xc] {var_10}
0001048c  14301be5  ldr   r3, [r11,  #-0x14] {var_18}
00010490  18201be5  ldr   r2, [r11,  #-0x18] {var_1c}
00010494  920301e0  mul   r1, r2, r3
00010498  10201be5  ldr   r2, [r11,  #-0x10] {var_14}
0001049c  0230a0e1  mov   r3, r2
000104a0  8332a0e1  lsl   r3, r3, #0x5
000104a4  023083e0  add   r3, r3, r2
000104a8  8330a0e1  lsl   r3, r3, #0x1
000104ac  023083e0  add   r3, r3, r2
000104b0  8331a0e1  lsl   r3, r3, #0x3
000104b4  023083e0  add   r3, r3, r2
000104b8  8322a0e1  lsl   r2, r3, #0x5
000104bc  023083e0  add   r3, r3, r2
000104c0  033081e0  add   r3, r1, r3
000104c4  08300be5  str   r3, [r11,  #-0x8] {var_c}
000104c8  18301be5  ldr   r3, [r11,  #-0x18] {var_1c}
000104cc  18201be5  ldr   r2, [r11,  #-0x18] {var_1c}
000104d0  930202e0  mul   r2, r3, r2
000104d4  08301be5  ldr   r3, [r11,  #-0x8] {var_c}
000104d8  033082e0  add   r3, r2, r3
000104dc  0c300be5  str   r3, [r11,  #-0xc] {var_10}
000104e0  0c301be5  ldr   r3, [r11,  #-0xc] {var_10}
000104e4  0c201be5  ldr   r2, [r11,  #-0xc] {var_10}
000104e8  920303e0  mul   r3, r2, r3
000104ec  0c201be5  ldr   r2, [r11,  #-0xc] {var_10}
000104f0  920303e0  mul   r3, r2, r3
000104f4  0300a0e1  mov   r0, r3
000104f8  00d04be2  sub   sp, r11, #0
000104fc  04b09de4  ldr   r11, [sp],  #0x4 {var_4}
00010500  1eff2fe1  bx    lr
```

A common Problem

“ I just need to implement this in my code and we’ll be done ”

“ … 5 more minutes …”

“Check every value against gdb”

“This should work”

```
generate_key:
00010464  04b02de5  str   r11, [sp,  #-0x4]!
00010468  00b08de2  add   r11, sp, #0 {var_4}
0001046c  1cd04de2  sub   sp, sp, #0x1c
00010470  10000be5  str   r0, [r11,  #-0x10] {var_14}
00010474  14100be5  str   r1, [r11,  #-0x14] {var_18}
00010478  18200be5  str   r2, [r11,  #-0x18] {var_1c}
0001047c  10201be5  ldr   r2, [r11,  #-0x10] {var_14}
00010480  14301be5  ldr   r3, [r11,  #-0x14] {var_18}
00010484  033082e0  add   r3, r2, r3
00010488  0c300be5  str   r3, [r11,  #-0xc] {var_10}
0001048c  14301be5  ldr   r3, [r11,  #-0x14] {var_18}
00010490  18201be5  ldr   r2, [r11,  #-0x18] {var_1c}
00010494  920301e0  mul   r1, r2, r3
00010498  10201be5  ldr   r2, [r11,  #-0x10] {var_14}
0001049c  0230a0e1  mov   r3, r2
000104a0  8332a0e1  lsl   r3, r3, #0x5
000104a4  023083e0  add   r3, r3, r2
000104a8  8330a0e1  lsl   r3, r3, #0x1
000104ac  023083e0  add   r3, r3, r2
000104b0  8331a0e1  lsl   r3, r3, #0x3
000104b4  023083e0  add   r3, r3, r2
000104b8  8322a0e1  lsl   r2, r3, #0x5
000104bc  023083e0  add   r3, r3, r2
000104c0  033081e0  add   r3, r1, r3
000104c4  08300be5  str   r3, [r11,  #-0x8] {var_c}
000104c8  18301be5  ldr   r3, [r11,  #-0x18] {var_1c}
000104cc  18201be5  ldr   r2, [r11,  #-0x18] {var_1c}
000104d0  930202e0  mul   r2, r3, r2
000104d4  08301be5  ldr   r3, [r11,  #-0x8] {var_c}
000104d8  033082e0  add   r3, r2, r3
000104dc  0c300be5  str   r3, [r11,  #-0xc] {var_10}
000104e0  0c301be5  ldr   r3, [r11,  #-0xc] {var_10}
000104e4  0c201be5  ldr   r2, [r11,  #-0xc] {var_10}
000104e8  920303e0  mul   r3, r2, r3
000104ec  0c201be5  ldr   r2, [r11,  #-0xc] {var_10}
000104f0  920303e0  mul   r3, r2, r3
000104f4  0300a0e1  mov   r0, r3
000104f8  00d04be2  sub   sp, r11, #0
000104fc  04b09de4  ldr   r11, [sp],  #0x4 {var_4}
00010500  1eff2fe1  bx    lr
```

“wut”

“This works… like… half the time….”

“Let’s start from scratch”

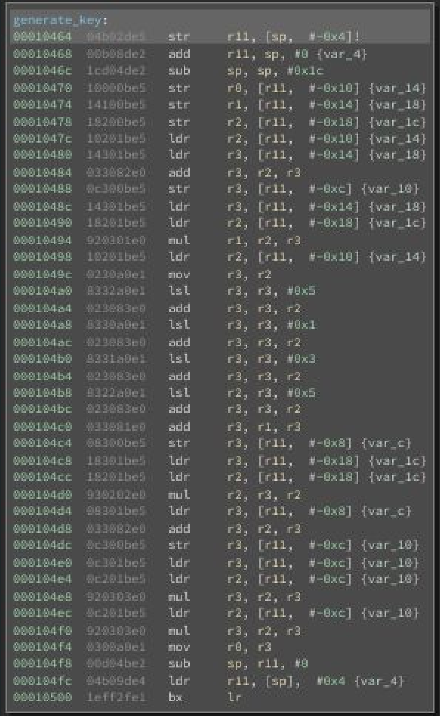
Reimplementing Code can be rough

- Not usually the *fun* or *interesting* part of Reverse Engineering
- Very Bug Prone
    - “Oops I missed that left-shift up there”
- Can turn into a *long, tedious* process

```
generate_key:
00010464  04b02de5  str   r11, [sp,  #-0x4]!
00010468  00b08de2  add   r11, sp, #0 {var_4}
0001046c  1cd04de2  sub   sp, sp, #0x1c
00010470  10000be5  str   r0, [r11,  #-0x10] {var_14}
00010474  14100be5  str   r1, [r11,  #-0x14] {var_18}
00010478  18200be5  str   r2, [r11,  #-0x18] {var_1c}
0001047c  10201be5  ldr   r2, [r11,  #-0x10] {var_14}
00010480  14301be5  ldr   r3, [r11,  #-0x14] {var_18}
00010484  033082e0  add   r3, r2, r3
00010488  0c300be5  str   r3, [r11,  #-0xc] {var_10}
0001048c  14301be5  ldr   r3, [r11,  #-0x14] {var_18}
00010490  18201be5  ldr   r2, [r11,  #-0x18] {var_1c}
00010494  920301e0  mul   r1, r2, r3
00010498  10201be5  ldr   r2, [r11,  #-0x10] {var_14}
0001049c  0230a0e1  mov   r3, r2
000104a0  8332a0e1  lsl   r3, r3, #0x5
000104a4  023083e0  add   r3, r3, r2
000104a8  8330a0e1  lsl   r3, r3, #0x1
000104ac  023083e0  add   r3, r3, r2
000104b0  8331a0e1  lsl   r3, r3, #0x3
000104b4  023083e0  add   r3, r3, r2
000104b8  8322a0e1  lsl   r2, r3, #0x5
000104bc  023083e0  add   r3, r3, r2
000104c0  033081e0  add   r3, r1, r3
000104c4  08300be5  str   r3, [r11,  #-0x8] {var_c}
000104c8  18301be5  ldr   r3, [r11,  #-0x18] {var_1c}
000104cc  18201be5  ldr   r2, [r11,  #-0x18] {var_1c}
000104d0  930202e0  mul   r2, r3, r2
000104d4  08301be5  ldr   r3, [r11,  #-0x8] {var_c}
000104d8  033082e0  add   r3, r2, r3
000104dc  0c300be5  str   r3, [r11,  #-0xc] {var_10}
000104e0  0c301be5  ldr   r3, [r11,  #-0xc] {var_10}
000104e4  0c201be5  ldr   r2, [r11,  #-0xc] {var_10}
000104e8  920303e0  mul   r3, r2, r3
000104ec  0c201be5  ldr   r2, [r11,  #-0xc] {var_10}
000104f0  920303e0  mul   r3, r2, r3
000104f4  0300a0e1  mov   r0, r3
000104f8  00d04be2  sub   sp, r11, #0
000104fc  04b09de4  ldr   r11, [sp],  #0x4 {var_4}
00010500  1eff2fe1  bx    lr
```

It’d be great if we could …

- Right click, “Package Function”
- Save the output
- Call “my_func.run()” from Python
- Get the *exact functionality* regardless of host platform



It’d be great if we could …

- Right click, “Package Function”
- Save the output
- Call “my_func.run()” from Python
- Get the *exact functionality* regardless of host platform

```
generate_key:
00010464  04b02de5  str   r11, [sp,  #-0x4]!
00010468  00b08de2  add   r11, sp, #0 {var_4}
0001046c  1cd04de2  sub   sp, sp, #0x1c
00010470  10000be5  str   r0, [r11,  #-0x10] {var_14}
00010474  14100be5  str   r1, [r11,  #-0x14] {var_18}
00010478  18200be5  str   r2, [r11,  #-0x18] {var_1c}
0001047c  10201be5  ldr   r2, [r11,  #-0x10] {var_14}
00010480  14301be5  ldr   r3, [r11,  #-0x14] {var_18}
00010484  033082e0  add   r3, r2, r3
00010488  0c300be5  str   r3, [r11,  #-0xc] {var_10}
0001048c  14301be5  ldr   r3, [r11,  #-0x14] {var_18}
00010490  18201be5  ldr   r2, [r11,  #-0x18] {var_1c}
00010494  920301e0  mul   r1, r2, r3
00010498  10201be5  ldr   r2, [r11,  #-0x10] {var_14}
0001049c  0230a0e1  mov   r3, r2
000104a0  8332a0e1  lsl   r3, r3, #0x5
000104a4  023083e0  add   r3, r3, r2
000104a8  8330a0e1  lsl   r3, r3, #0x1
000104ac  023083e0  add   r3, r3, r2
000104b0  8331a0e1  lsl   r3, r3, #0x3
000104b4  023083e0  add   r3, r3, r2
000104b8  8322a0e1  lsl   r2, r3, #0x5
000104bc  023083e0  add   r3, r3, r2
000104c0  033081e0  add   r3, r1, r3
000104c4  08300be5  str   r3, [r11,  #-0x8] {var_c}
000104c8  18301be5  ldr   r3, [r11,  #-0x18] {var_1c}
000104cc  18201be5  ldr   r2, [r11,  #-0x18] {var_1c}
000104d0  930202e0  mul   r2, r3, r2
000104d4  08301be5  ldr   r3, [r11,  #-0x8] {var_c}
000104d8  033082e0  add   r3, r2, r3
000104dc  0c300be5  str   r3, [r11,  #-0xc] {var_10}
000104e0  0c301be5  ldr   r3, [r11,  #-0xc] {var_10}
000104e4  0c201be5  ldr   r2, [r11,  #-0xc] {var_10}
000104e8  920303e0  mul   r3, r2, r3
000104ec  0c201be5  ldr   r2, [r11,  #-0xc] {var_10}
000104f0  920303e0  mul   r3, r2, r3
000104f4  0300a0e1  mov   r0, r3
000104f8  00d04be2  sub   sp, r11, #0
000104fc  04b09de4  ldr   r11, [sp],  #0x4 {var_4}
00010500  1eff2fe1  bx    lr
```

The “dream” for ripr is to make this happen

When is this relevant?

This is a common scenario during the reverse engineering process

- Vulnerability Research
  - Can I get register “x” to be equal to “y” under condition “z” with input “I”
  - Can I run this code without setting up a full testbed environment?
- Malware Analysis
  - Can I quickly confirm this function does “x”
  - Can I avoid re-writing this obfuscation technique? Encoding/Decoding Technique? Etc…
- “Exactness”
  - I need my code to have the same quirks and bugs as the original

[ripr] From *what* to *how*

What do we need to accomplish this?

- A way to actually run code from a wide variety of architectures
- A way to figure out what data or other code our target depends on

What do we need to accomplish this?

- A way to actually run code from a wide variety of architectures
- A way to figure out what data or other code our target depends on

Basically:

- We need a diverse, flexible emulator
- We need a good disassembler

[Unicorn Engine] Introduction

“ *Unicorn is a lightweight multi-platform, multi-architecture CPU emulator framework.* ”

Has support for:

- ARM/ARM64
- x86/x86_64
- Power PC
- SPARC
- MIPS
- M68k

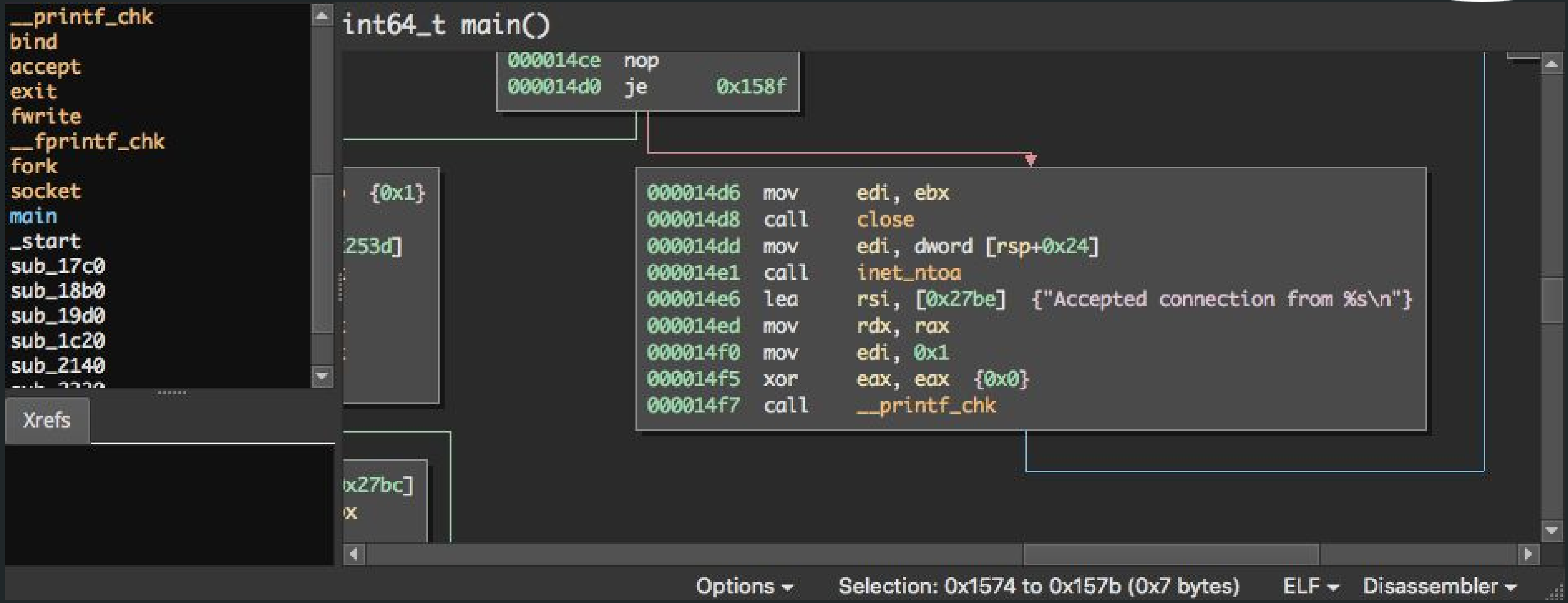
Runs on:

- Windows
- Linux
- OSX

Essentially: A diverse, fast, emulator we can script

[Binary Ninja] Introduction

Binary Ninja is a modern reverse-engineering framework.



[Binary Ninja] Introduction

Binary Ninja is a modern reverse-engineering framework.

Key Features:

- Disassembler
- Clean UI
- “Universal” Intermediate Language
  - Low-Level IL or LLIL
- Powerful, clean API
- Actively Developed

Essentially: A modern, highly extensible static analysis platform

Design and Implementation

Overarching Goal / "*The Dream*": Right-Click on Code → Use it from Python: As quickly and accurately *as possible*.

High Level Strategy:

- Create basic environment (usually, map the code, create a stack)
- Identify dependencies of the target code
    - Code and Data that the target code needs to run correctly
- Put these into the environment
- Optional: Create a convenient interface to the emulated environment

Design and Implementation

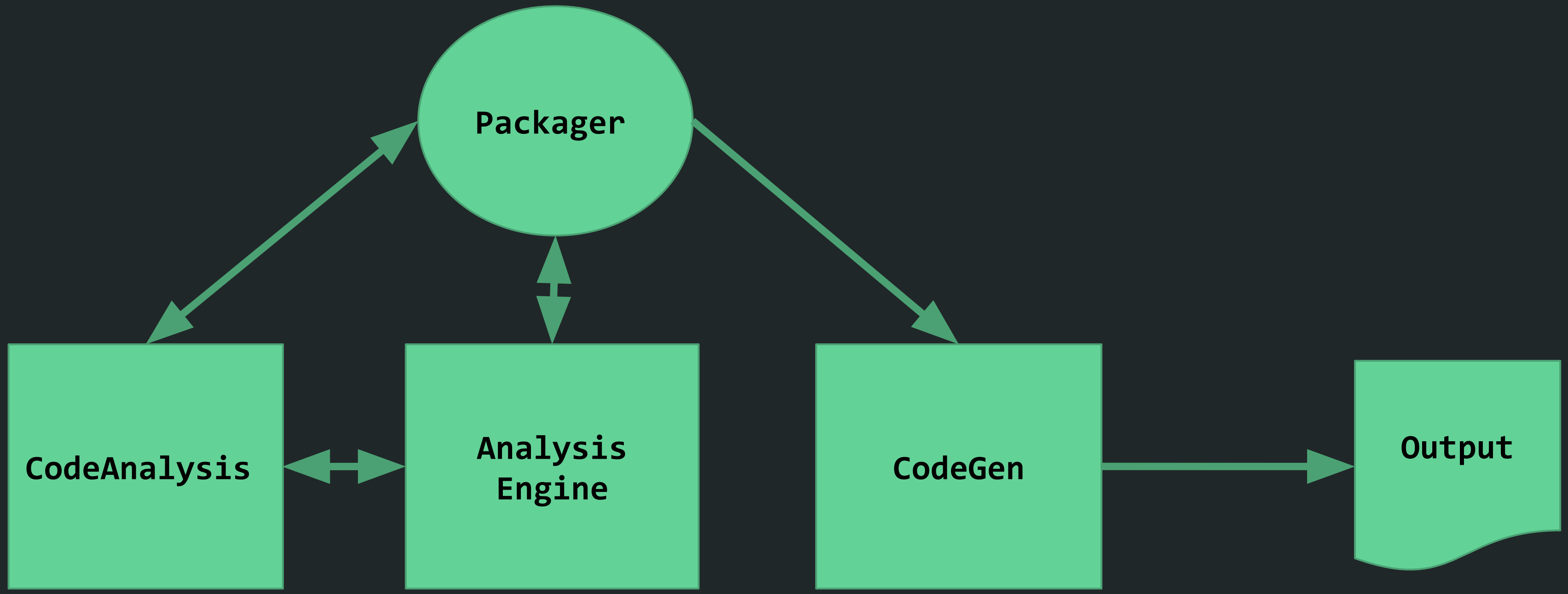
Overarching Goal / "*The Dream*": Right-Click on Code ➜ Use it from Python: As quickly and accurately *as possible*.

Design Goals:

- Abstract away Backend Dependencies
- Clear boundaries on functionality between components
- Expose internal state to users in an *intuitive* and *visual* way when possible
- Make output correct, but nevertheless easily "tweakable"

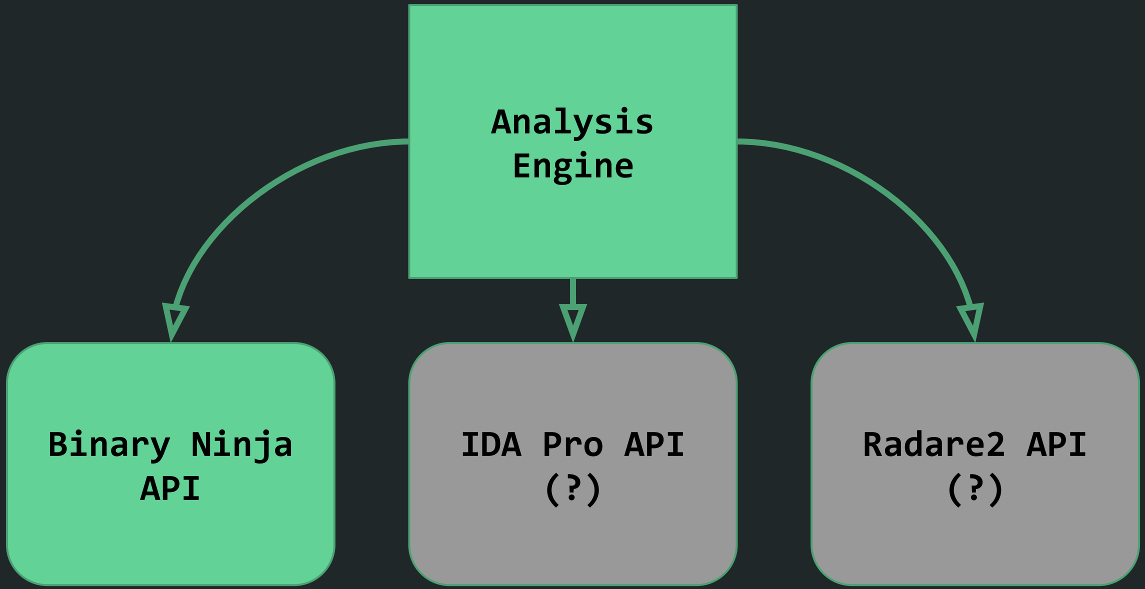
Design and Implementation

Four Conceptual Components

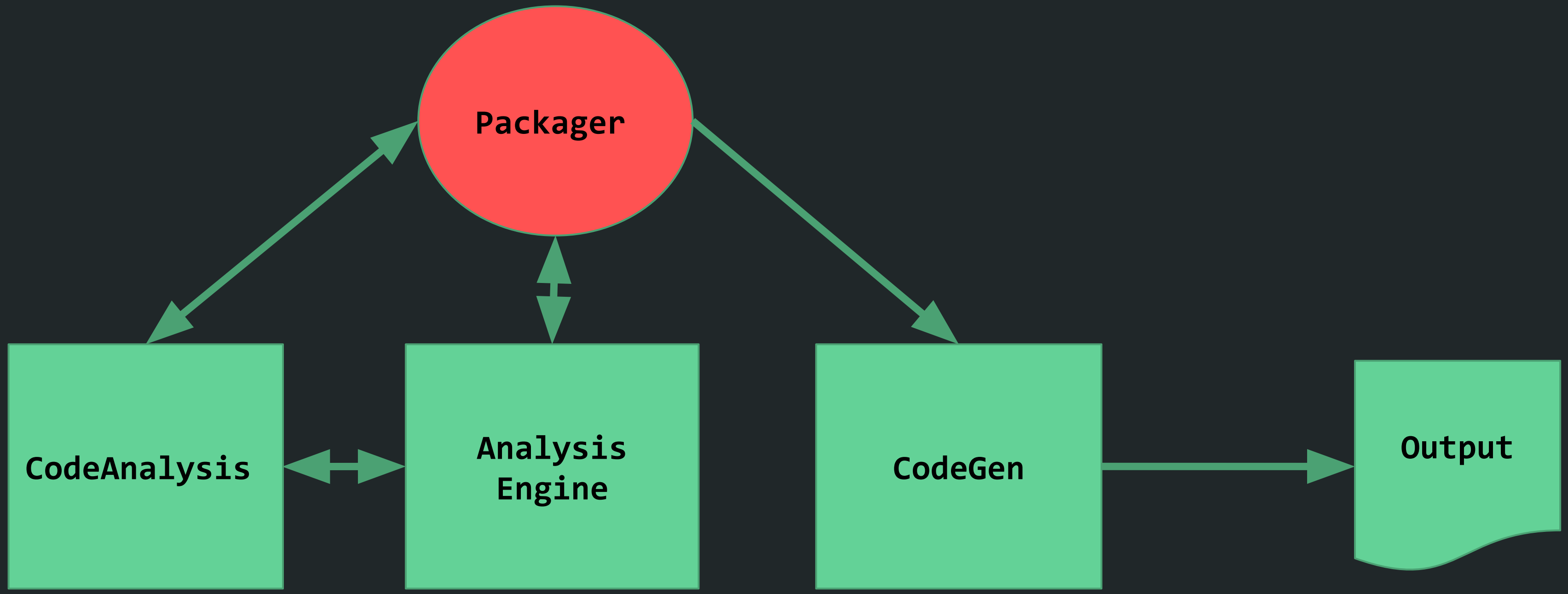


Analysis Engine

Analysis Engine is an abstraction over a static analysis backend.



Packager



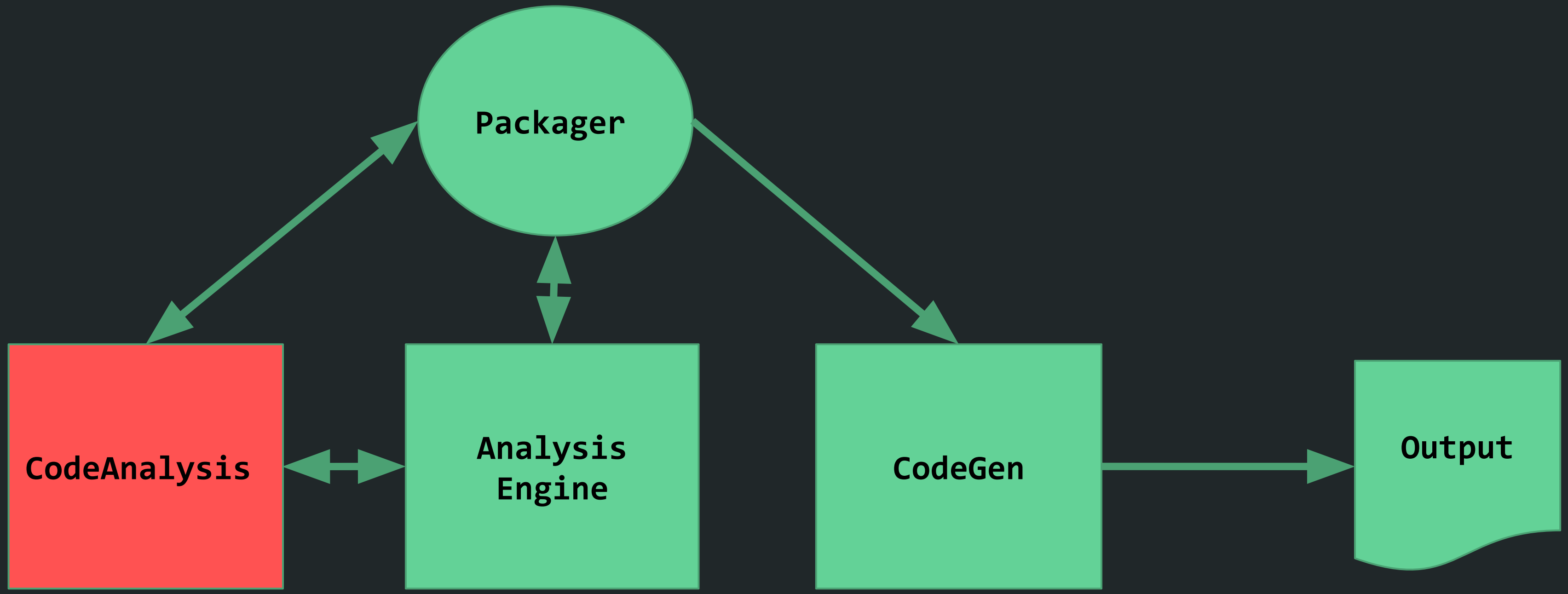
Packager

In any given run, the Packager object is “in the driver’s seat”.

The Packager object is where the high-level behavior of ripr is defined.

Triggers CodeGen when enough information is available to create valid output.

CodeAnalysis



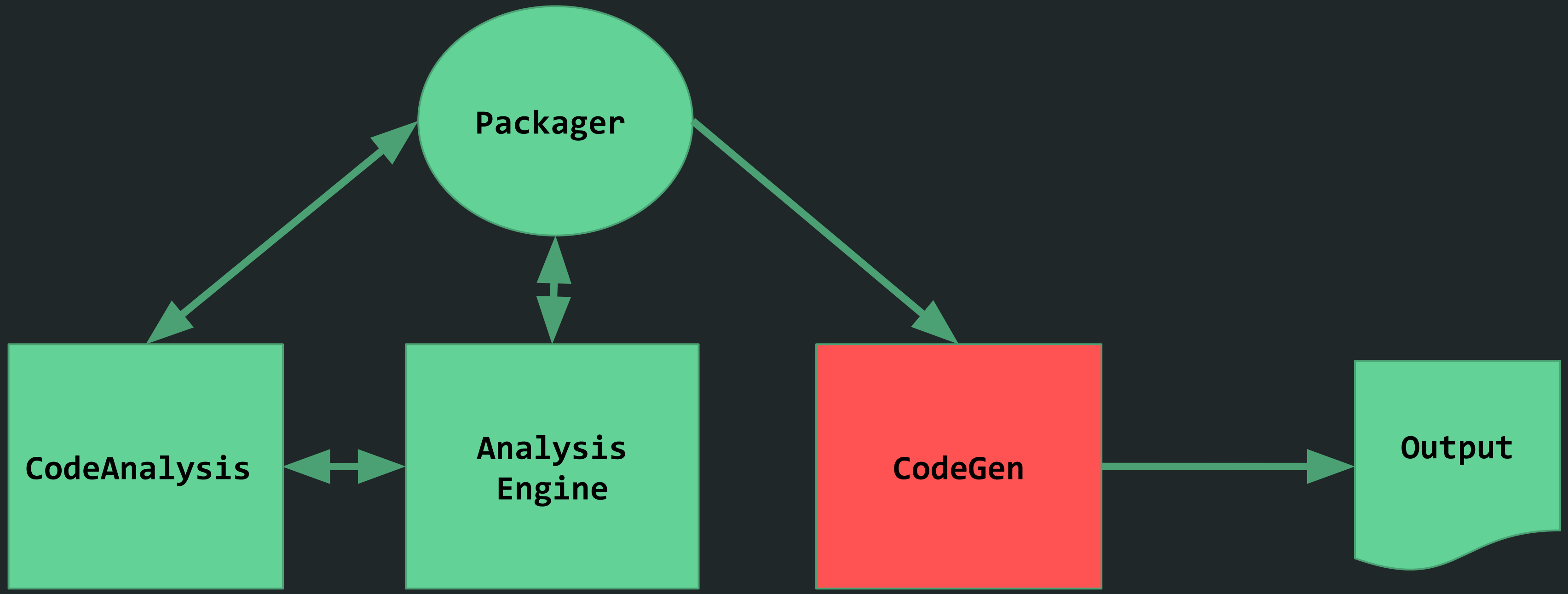
CodeAnalysis

CodeAnalysis is responsible for encapsulating Dependency Resolution and Convenience Passes.

It utilizes the static analysis backend in order to determine what code and data needs to be included for the target code to function properly.

It contains code for deciding if enough information is available to include some convenience methods in our output.

CodeGen



CodeGen

CodeGen is responsible for outputting a valid Python class that encapsulates the selected code using information it gets from Packager.

CodeGen

CodeGen is responsible for outputting a valid Python class that encapsulates the selected code using information it gets from Packager.

Contains:

- Pages to map in the emulator state
- Code and Data to map into the emulator
- Methods for emitting valid Python
- Architecture Specific settings for Unicorn

CodeGen

Segregating this code has the additional benefit of easily porting ripr to support other “host” languages.

Adding { Perl, Rust, Haskell, Ruby, Java, Go, .NET, Delphi/Pascal } support is as “simple” as changing string constants in CodeGen’s emitter methods.

[ripr] Execution Environment Basics

What are the *Bare Bones* things we need?

At absolute minimum, we must include the code we want to emulate, and a stack.

Creating a stack is trivial:

- Map some memory
- Point the target architecture's Stack Pointer within it.

Collecting Target Code

The overall strategy looks like this:

1. Collect Basic Blocks belonging to the target Code
2. “Smash” Contiguous blocks into a “unit”
3. Map these units into the emulator state
4. Mark the “latest” unit with a return as the boundary of the target code.

Return Guards

What happens when you hit the end of a function?

>> retn

We’ve started execution “in the middle of things”. This will crash every time.

Need a way to “signal” that we’ve hit the end of our function successfully.

Return Guards

Solution:

Set up the environment such that we will crash at a particular "marker" value when things go correctly.

E.g Set Return Address to 0x1, check if we crash there at the end.

Are we done?

These steps build the “core” of the emulator state.

However, we would be limited to emulating only “purely logical” functions which do not depend on *any* external data or code.

[ripr] Dependency Resolution

What are “Dependencies” in this context?

Dependencies are all the code and data that our target code may access or execute during a normal execution.

What are “Dependencies” in this context?

Dependencies are all the code and data that our target code may access or execute during a normal execution.

- Global Variables
- .data, .rodata, etc variables
- Jump or Call Targets not in the original scope

Missing these will result in incorrect behavior.

Dependency Resolution Strategies

There are three strategies in ripr used to identify these dependencies.

simpleDataScan: Relies purely on information from the static analysis backend

dataScan: Uses information from LLIL to augment dependency resolution.

branchScan: Use LLIL to look for unmapped Call/JMP Targets

Data Mapping Strategies

The two “data” strategies result in a list of dataDependency objects.

However, it can be impossible to know exactly *what* data is being used.

Oftentimes, only a base, or starting, address can be identified.

Data Mapping Strategies

There are two strategies currently used to address this:

1. Section Mapping Mode
2. Page Mapping Mode

Section Mapping Mode

Main Idea: If a section is accessed, just map and copy that section.

Page Mapping Mode

Main Idea: Mark pages within the binary that contain a potential dependency. Map this page in the emulator, and copy *all* of its contents from the binary.

```
pages = set()
for dep in dependencies:
    pages.append(get_page_of_addr(dep.address))

for page in pages:
    emu.map(page)
    copy_data_to_emu(page)
```

Data Mapping Strategies

The default in ripr is to use Section Mapping Mode.

Provides “pretty good” reliability in practice.

Won’t accidentally map GOT/PLT/etc (This can lead to issues later on)

branchScan

Main Idea: Use LLIL to check if we need to map additional call or jump targets.

For every Jump, Call, or GOTO, check if the target is in our mapped range.

In Range: Great! Keep going.

Out of Range: Add to codeDependency List

branchScan

For all the codeDependencies:

1. Map and Copy them into the emulator state
2. Run dependency-finding code against them
3. Repeat for any new codeDependencies

[ripr] Convenience Passes

Analysis we don't *need*

Convenience Passes are methods in CodeAnalysis that are non-essential.

Specifically, these methods are **not** concerned with identifying dependencies of the target code.

Return Value Extraction

Typically, functions in x86_64 Linux code use the rax register to return a value.

If applicable, we can structure our package such that it’s “run” method automatically queries the emulator and returns whatever is in rax at the end of execution.

Return Value Extraction

This allows packaged code to be used more "naturally".

```
x = my_package.run()

>> x = 2
```

```
my_package.run()
x = my_package.emu_state.read_reg(REG_X86_RAX)

>> x = 2
```

Return Value Extraction

This allows packaged code to be used more “naturally”.

```
x = my_package.run()

>> x = 2
```

```
my_package.run()
x = my_package.emu_state.read_reg(REG_X86_RAX)

>> x = 2
```

Unimplemented: Use available type information to automatically grab relevant data.

Ex. my_package returns a char *, automatically dereference and return contents from emulator.

[ripr] Limitations

Limitations

Most of the Limitations of ripr are due to one, main principle: We cannot emulate what we do not have access to.

Some common examples:

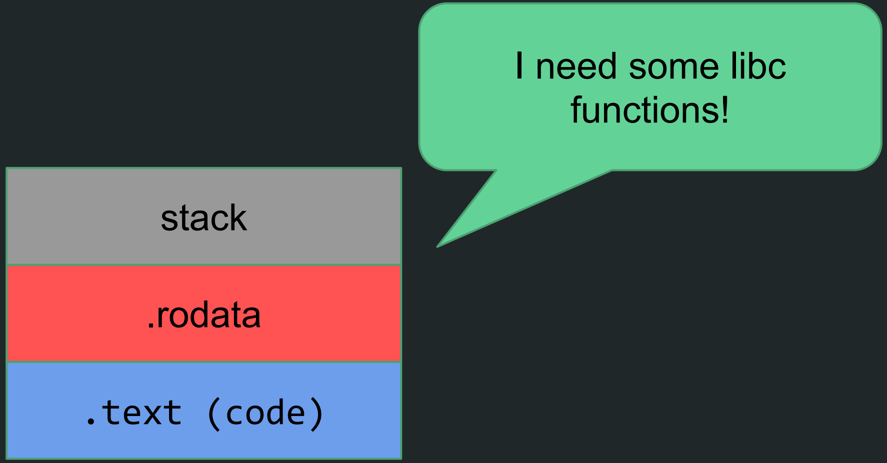
- Imported Functions / Imported Data
- Syscalls

Imported Functions

Imported Functions are functions which are loaded into the address space at runtime. This most commonly happens during program startup.

Imported Functions

Imported Functions are functions which are loaded into the address space at runtime. This most commonly happens during program startup.



Imported Functions

Imported Functions are functions which are loaded into the address space at runtime. This most commonly happens during program startup.

Imported Functions

This implies that, most of the time, we won’t even have access to the library where these functions are imported from.

There are two strategies present to deal with this:

- Hook-and-replace
- NOP-ing out calls

Hook and Replace

Basic Idea: Let calls to imported functions “crash” the emulator. Pass execution to a python “hook”. Restore context and resume emulation.

Hook and Replace

Basic Idea: Let calls to imported functions “crash” the emulator. Pass execution to a python “hook”. Restore context and resume emulation.

```
main:
00400560  55           push  rbp
00400561  4889e5       mov   rbp, rsp {var_8}
00400564  4883ec10     sub   rsp, 0x10
00400568  897dfc       mov   dword [rbp-0x4 {var_c}], edi
0040056b  488975f0     mov   qword [rbp-0x10 {var_18}], rsi
0040056f  bf20064000   mov   edi, 0x400620  {"In Main"}
00400574  e887feffff   call  puts  // [ripr] Imported Call !!
00400579  b800000000   mov   eax, 0x0
```

In more detail

1. Keep a record of “Expected Return Address : Imported Function”
   a. Build this statically during packaging
2. When the emulator “crashes”, recover the return address
   a. x86/x64: Look at the stack
   b. arm: Look at LR
3. Check against the addresses we’ve identified in 1.
4. Call the corresponding python hook
5. Do any clean-up required, and start emulating again at the expected return address.

What can you do from these hooks?

Anything you need or want to!

Unicorn State is accessible from these hooks

- Can make any necessary modifications to program state
- Can access any data from program state
- Add or remove “Unicorn Hooks”

NOP-ing out Calls

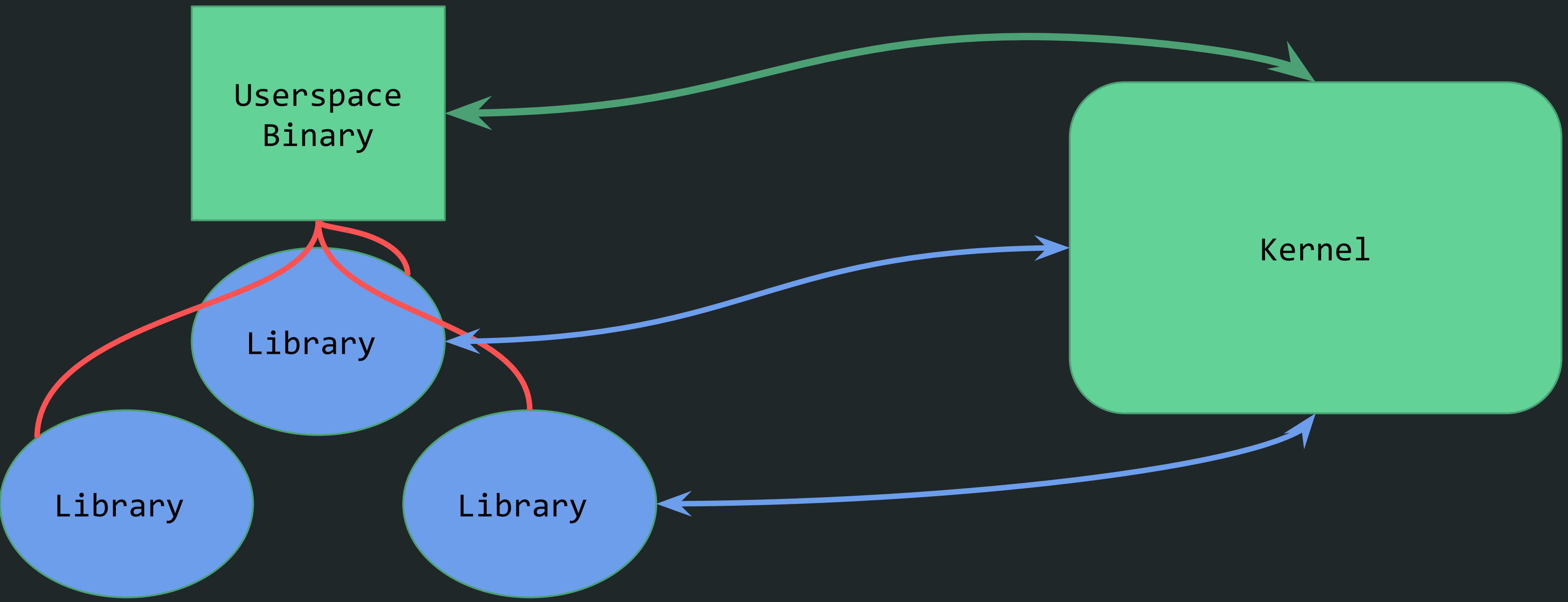
A series of NOPs (no-op) can be used to overwrite the imported function call.

In many cases, this is a harmless and useful resolution:

- Printing Functions
- Logging Functions
- Sleeps

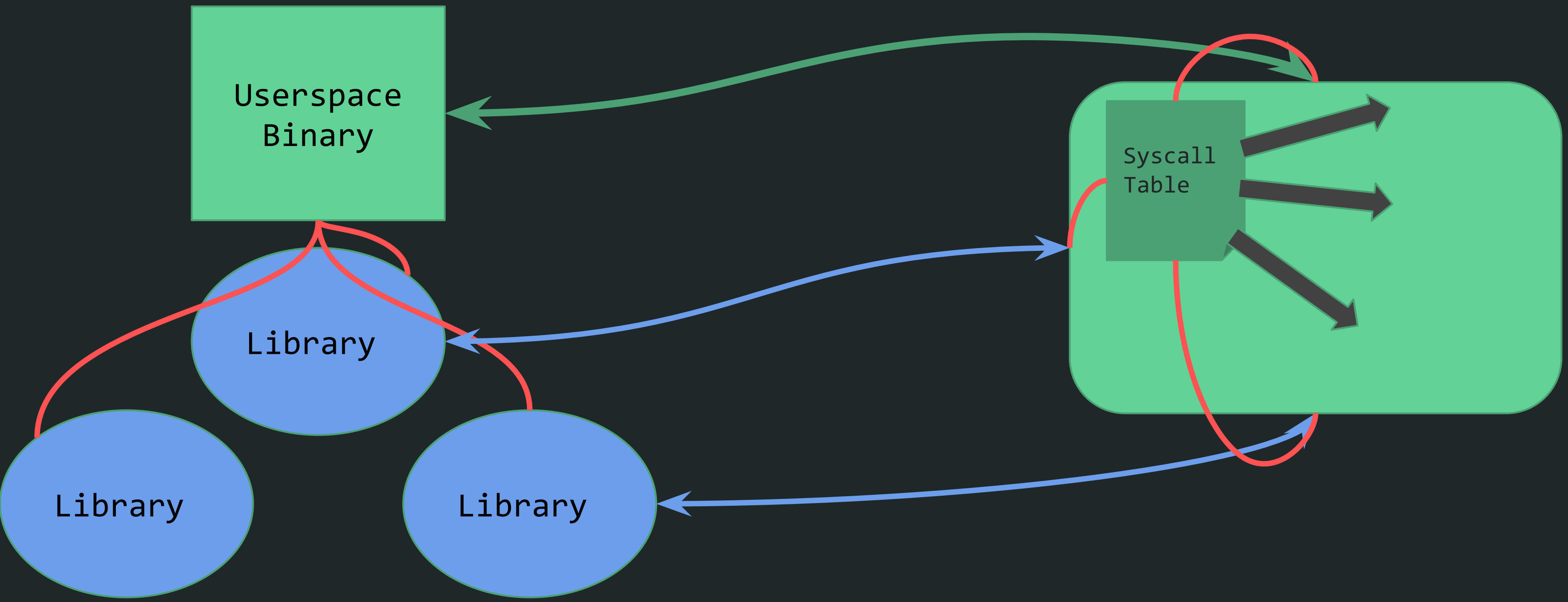
System Calls

Properly emulating System Calls is frequently impossible or extremely inefficient.



System Calls

Syscalls can trigger a vast amount of kernel code.



System Calls

The implication is that we would have to emulate a large portion (all?) of the Kernel.

Additionally, things syscalls *do* frequently do not make sense in this emulated environment.

- Read/Write
  - No Files or File Descriptors!
- Create a socket
  - No network hardware
- Execute another program
  - No FileSystem, No scheduler, No loader, etc...

Demo Time

Acknowledgements / Shout outs

Professor Yener

RPISEC

Shmoocon

The End

Questions? Comments? Complaints?

irc: Unix_Dude -- freenode, irc.rpis.ec

Binary Ninja slack: pbiernat

Twitter: @PatrickBiernat

In a few minutes/hours:

https://github.com/pbiernat/ripr