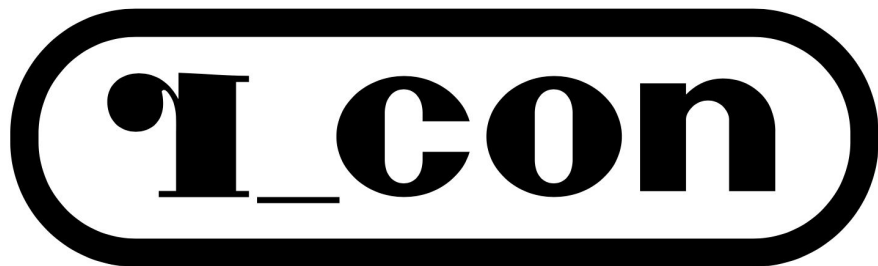# Developing Internals

r2con2019 @ pancake

Summary

- Introduction to radare2
- C Programming Language Basics
- Good coding practices
- Making tests
- How Code is Structured
- Extensibility and Plugins
- Integration with other programming languages
- How to contribute, add tests, improve stability

What We Will Learn (not in this specific order)

- Good coding practices
- Understand how code is structured in r2
- Writing plugins
- Write a regression test
- Identify bugs from reproducer to implementation
- Report issues
- Fix them in a pull request
- Fuzz for more

Background Requirements

- C programming language
- Unix shell and libc basics

Examples and exercises are here:
https://github.com/condret/r2-dev-training-exercises

- Javascript / Python is welcome

POLL: Raise Your Hand!

- Do you use (or know) r2
- Do you know C?
- Python?
- Javascript?

POLL: Why are you here?

- Learn about r2 internals
- Write your own plugins
- Fix bugs in r2
- Find bugs in r2
- Improve radare2 by implementing new features
- Learn about software development, maintenance and security.

Introduction to radare2

What is Radare2?

Free/libre open-source reverse-engineering framework written in C with UNIX concepts in mind.

- Supports static and dynamic analysis
- Support for debugging and emulation
- Visual mode, web interface, cli interface
- Support for scripting in various languages
- Multi platform and multi architecture

Tools

- Radare2
- Rabin2
- Rax2
- Rasm2
- Rahash2
- Rafind2
- Ragent2
- ...

Commands

They are known to be cryptic and may look hard to learn at first sight, but once you understand the basic logic behind each char you'll catch the rest quickly.

- All the features exposed by the r2 libraries can be accessed not just by API and the system shell, but also, as commands inside r2.

Command Syntax Rules

PREFIX

- . to run a command and interpret the output as commands
    - You can also use the |. suffix
- ! to escape to the shell (and dump/execute the cmd history)
    - Use !! to escape without capturing the output
    - !!! to manage autocompletion
- number to repeat a command
- # for comments
- \ is the same as ' and the same as =! (prefix for IO plugins)

Command Syntax Rules

SUFFIX

- j - output in json
- * - output in r2 commands
- = - ascii art columns
- l - long output (verbose) (afll)
- t - (new) table API

Command Syntax Rules

MODIFIERS

- @ - temporal seek - see ?@? For more
- @@ - foreach operator
- @@@ - block based foreach operator
- | - pipe to system command (|H for HTML filtering)
- > - redirect output to a file (>> to append)
- ~ internal grep (~.. for less, ~... for hud)

Command Implementation

- All the commands are implemented in libr/core/cmd*

There are plans to make a syntax parser and make a better parsing in a structured way… but for now we're fine with the ugly switch/if approach.

- Autocompletion can be tweaked, changed or added at runtime with the !!! command.

Adding new commands in RCore

- You can make an RCore plugin, which handles only 1 fcn callback

```
RCorePlugin r_core_plugin_a2f = {
        .name = "pluginName",
        .desc = "plugin Descriptor",
        .license = "LGPL3",
        .call = r_cmd_call,
};

R_API RLibStruct radare_plugin = {
        .type = R_LIB_TYPE_CORE,
        .data = &r_core_plugin_a2f,
        .version = R2_VERSION
};
```

```
static int r_cmd_anal_call(void *user, const char *input) {
    RCore *core = (RCore *) user;
    return 0;
}
```

Exercise: Making an RCore plugin

See libr/core/p/* or find another one in r2e with git grep RCorePlugin

How is the Code Structured in r2

API Summary

- **IO**
  - Descriptors and Maps
- **RBin**
  - Parsing binary headers
- **RAsm**
  - Assembler and Disassemblers
- **RAnal**
  - Improve jump table analysis or tweak the main loop?
  - Maybe good excuse to fix the anal.trim
- **RCore**
  - Show how r2pipe.native works

File Namings

As long as each linker have different limitations I decided to have unique names across all the modules in order to be able to create a single archive library containing all the objects.

Bear in mind that archive (.a) files can contain multiple copies of the same filename and they are treated as different files, which may be a problem when uncompressing them.
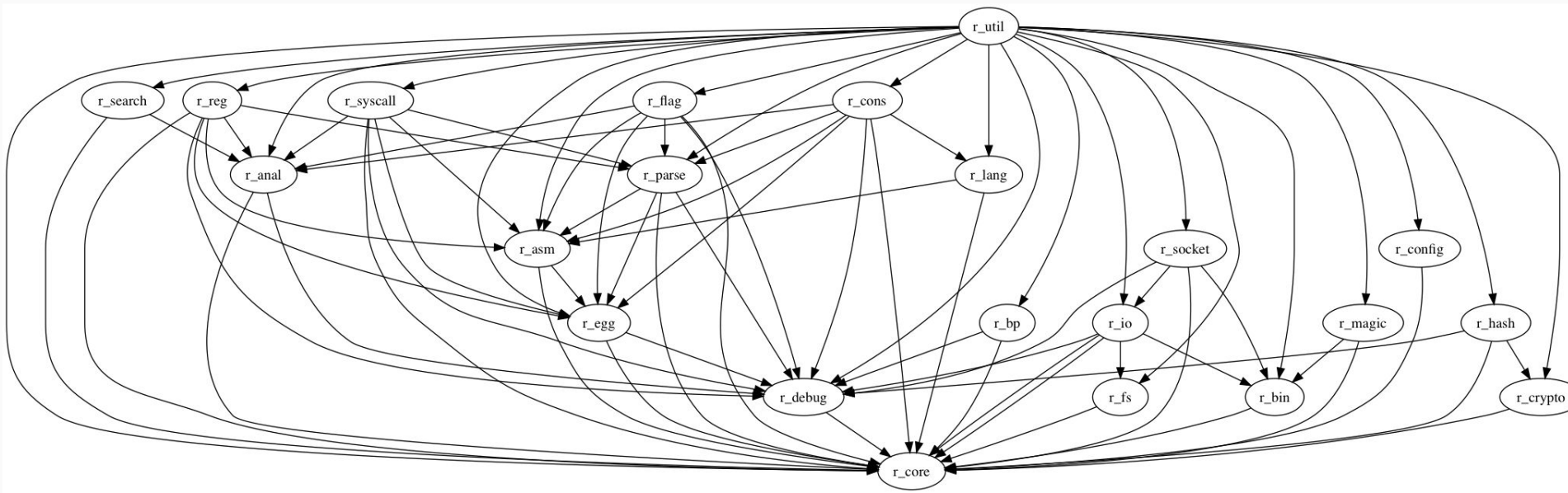
API Summary

But there are more!

- bp/
- config/
- cons/
- crypto/
- debug/
- egg/
- flag/
- fs/

- hash/
- lang/
- magic/
- parse/
- reg/
- search/
- socket/
- syscall/
- util/

How are they related to each other?

- `make depgraph.png`

Module Dependencies

- Dependencies are important
    - You cannot use RCore apis from RUtil.
- Forces us to find cleaner and more modular designs.
- RBind instances to solve this problem
- Possibility to replace each part of r2 with a different impl. (Rust, Go, ..)

See RBindIO, RBindCore, ...

External Dependencies

There are 0 mandatory dependencies in r2.

- Some external deps can be linked after configure
    - Capstone (4 or 5)
    - Zip/zlib
    - OpenSSL
    - Libmagic
    - LibUV

Quizz: Have you been paying attention? :D

- Where are the commands implemented in r2?
- Where is the source of the plugins?
- Which directory contains 3rd party projects?
- How many programs are shipped with r2?
- How to list all the plugins instantiated
- Which directories is r2 looking for plugins?

Releases

Making a release

- Update version in configure.acr and rerun autogen.sh
- Commit it and add a git tag with the version number
- Github will do the tarballs but you can do make dist

All the steps are implemented in a shell-script in the radare2-release repository.

- https://github.com/radare/radare2-release
- sys/release-notes.sh

Release Schedules

That's how we roll!

- Rolling releases every 6 weeks.
- Bumping +1.0 after every r2con
- Bumping +0.1 every 6 weeks

Building from Git

$ git clone --depth=1 https://github.com/radare/radare2

$ cd radare2

$ sys/install.sh

.. sys/rebuild.sh gdb # if you want to rebuild part of r2 to solve the problems introduced by the lack of revdeps in the makefiles for stuff in shlr/

With Meson

- $ git clone --depth=1 https://github.com/radare/radare2
- $ sys/meson.sh

Which is the same as:

- Meson build
- Cd build
- Ninja

Using Git

How to use Git

Git is the version control system we use. So it's important to know how to operate with it in order to make branches, resolve conflicts, squash commits, etc..

- Many of those words may sounds strange to you.
- Please install tig

Last week, the git devs decided to change the `git checkout` behaviour

Commit

A group of changes in files associated to a person and a time, and referenced to the parent commit, in order to have a hierarchy of patches.

$ git commit -a   # commit all changes (see git add, rm …)

$ git commit -p # pick change one by one

$ git commit libr/ # commit only changes inside libr

Branch

Master is the default branch, but you can have other branches following different changes, a branch is a name for a specific commit which is considered HEAD for that branch.

$ git checkout -b branchname # create new branch

$ git checkout branchname # switch to it

$ git reset --hard # reset all changes in files

$ git checkout -d branchname # delete a branch

Rebase

Rebasing is moving the heading commits in a branch rebasing them on top of the changes of another branch. This is necessary to avoid doing branch merges which can introduce many issues. Rebasing is cleaner.

$ git rebase master

Cherry Pick

If rebasing is not working you probably want to do this process by hand, this is done by using git cherry-pick to include a commit into your current branch.

$ git reset --hard @^^

$ git pull . master

$ git cherry-pick <hash1>

$ git cherry-pick <hash2>

Squash

Squashing is merging, aka combining N commits into one. This line will open vim and ask us what to do on each file, select all the lines, except the first one and change the first word of each line by the letter 's' (for squash)

$ git rebase -i @~4

Pull Request

A pull request is a process in github where the users can publish a branch of another repository asking the owners of the repo to be able to review, comment and accept. (supports merge, rebase and squash).

- Easiest way to do a pullreq is by pressing the pencil button.
- Otherwise, press the fork button, clone in local, make the branch, push -f
- In gitlab-slang it's called 'merge-request'

GitHub

- Pull requests
- Issues
- Commit message rules
- Branches
- How to use git

GitHub Issues

Labelled for cathegories (kind of tags)

- FEEDBACK WANTED
- Good first issue
- IMPORTANT
- RELEASE
- R2CON2019
- R2WARS
- RSOC

There are also labels for Arch/OS specific issues

- Windows
- macOS
- Linux
- ARM
- X86
- PowerPC
- Mips
- ...

Reporting issues

- Post version of r2
  - We only fix bugs that are in Git, no backports or so
- Operating System
- Architectures
- Crash Log
  - Backtrace
  - Register state
  - ASAN log is preferibly
  - Valgrind logs are helpful

Building with ASAN vs Valgrind

ASAN may catch many issues at runtime and execution is nearly native, output crashlogs are really verbose and useful.

$ sys/asan.sh **or** sys/meson.py --asan

Valgrind works with asan-free builds, execution is slower, and works mainly in Linux, Mac support is sometimes not complete and Windows is not supported.

Both support debugger integration

Testing Framework

Tests

Testing is

- Boring
- Important
- Impossible to cover all cases
- Time consuming

Tests

- r2's testsuite is in the radare2-regressions repository

We test many things:

- Commands
- Fuzzed files (loading and analyzing them)
- Assemble/disassemble instructions
- Debugger
- APIs (C)

Travis

There are many online services that can be used to build each commit of r2 and run the whole testsuite and notify in case of failure. This eases the maintenance and testing because we can be sure that the PR will not be breaking anything

- Jenkins is an opensource alternative that we used in the past but I didn't wanted to spend time with things i shouldn't care like maintaining servers, therefore using Travis is probably the best option.
- But fails randomly sometimes, as well as not being constant in time, so not usable for benchmarks.

Travis

Travis is right now running for Linux and Mac, and also enabling the ASAN build. So we can run the whole testsuite. (even the debugger tests).

Other CI platforms

We also use other CI platforms:

- Appveyor - Windows
- SourceHut - BSD
- Coverity - Source Issues
- ...

Creating a test

- Clone the repository

`$ git clone https://github.com/radare/radare2-regressions`

- Writing tests
- Running tests
- Finding the offending commit

If your commit in r2 is breaking any test you can update it by creating a new branch of your r2r repo fork

C Programming Language Basics

Syntax Rules

This is basically described in the DEVELOPERS.md file

- C99?
- Space before (and around +-/*%
- See other files as examples/inspiration

Space before parenthesis

Function definitions have no space before (

- Function calls need that.

This rule makes it easy to find function signatures

- Git grep 'r_core_cmd0('

static

Static make things private to the object

We use 3 levels of symbol visibility

- Static (private to the object, can be safely stripped at linking time)
- Internal (can be used from inside the same module)
- Public (function names must start with r_${module_name})

Typedef struct

- In r2land all the public structs use typedef definitions in order to provide camelcased versions.

        typedef struct foo { … } Foo;

- Foo foo;
- Struct foo foo;

Include Guards

- To avoid including the same file twice
- Follow some standards to avoid overriding system includes

```
#ifndef R2_CORE_H
#define R2_CORE_H
#endif
```

CamelCasings

During 2019 we did some changes in the coding style so we are slowly refactoring and cleaninig up everything to follow the new rules.

- Static methods (local to file) should start by a double lower-dash
- We use snake-case for all public methods
- CamelCase is fine for internal use only

R_API / R_IPI

R_API: It's a define to force public visibility.

- All exported functions must use this keyword.

R_IPI: Same, but for private visibility

r_types.h

All the basic types we use in r2 are redefined in this file.

This is because some types were only available in c99 and not all the compilers at the time was supporting this standard when the project started.

This is ut32 instead of uint32_t or unsigned int

etc...

Basic types

- Ut32
- St32
- Ut64
- St64
- Ut128
- Ut16
- Ut8
- …

Bool vs Int

Use the smallest possible data type for each use. Unix mixed data and errors in the same variable which is good for memory footprint but not for a well design because it makes some errors easier to happen when not checking for the value returned by a function.to be valid as -1, 0 or 1? Or maybe something else? To simplify as a convention we want to boolify as much apis as possible

- Split data vs errors
- Clarify the possible values
  - True = valid, false = failed

Enums

We don't use enums too much

- Some compilers make them 32 other 64, sometimes variable
- Not friendly with bitmasks (1<<1, 1<<2, ...)
- Hard to redefine or check if the exist over different versions
  - (better use #ifdef)

Indent examples

There are few ways to automatically indent C code:

None of them work as expected, so i wrote sys/install.sh

- Uses clang-format or uncrustify
- Readjusts somehow the code with perl regexpes
- See clang-format-diff.py

Development Tricks

Development Environment

- Which is your favourite editor?
  - I don't care which one you use, but i'll be using Vim.
  - But there's people using Visual Studio or Emacs to develop for r2
  - Obviously… not NANO
  - Even the pencil button in github is fine for some simple changes
- Compiler setup:
  - POSIX Shell && GNU Make
  - Meson + Ninja
  - Visual Studio Compiler in Docker

Development Tricks

- r_sys_backtrace();
- r_sys_breakpoint();

- Grep fcnname' ('
- What us R_API

Ctags

CTags, RTags ...are command line tools that can be integrated with vim and other editors in order to autocomplete function names, signatures, find references, jump to type definition, etc.

- But I personally just use grep. In fact git grep.

$ ctags -R .

nnoremap <leader>. :CtrlPTag<cr>

Compiler basics

Each compiler have its own checks and code generation issues/benefits, it is always good to use them all in o rder tog et as much feedback at compile time as possible

- CLANG (Easiest to use, build and availability)
- GCC (best errors out there, kind of Rust)
- VS (just windows things)
- TCC (Tiny!)

Don't let any warning appear in your Pull Requests!

Asserts

I was not really in the mood to introduce the use of asserts here and there, but the current implementation is configurable at compile and execution time, allowing safe, costless execution times as well as easy to debug.

- export R_ASSERT_DEBUG=1
- R_LOG APIs
  - --with-checks-level   value between 0 and 3 to enable different level of assert (see R_CHECKS_LEVEL) (R_CHECKS_LEVEL=2)

Only check for things that should not happen, not talking about alloc issues.

Multiple R2 builds

I use to have a single build of r2… but when you are in the middle of a patch and you get a binary to have a look at you will probably miss having a working version of r2.

- Git clone radare2 r2
- Just make symstall to switch between then

Quizzzzzz

- How to show a backtrace without a debugger?
- Function signature definitions contain a space or not before the (
- Shall we use asserts to check malloc failures?

Data Structures

Data Structures in r2land

- Double Linked Lists (SdbList, RList)
  - Can be used as stacks and queues
- Hashtables (SdbHt, RHashtable)
- Vectors (Arrays of structs)
- OIDStorage - filedescriptors
- SDB - key=value database
- RBTree
- …

RList

- double linked list
- can be used as a stack

Operations:

- split, merge, sort, iterate and join

Example: RList

(tiny example here, that puts n strings in a rlist)

Exercise: RList

Modify example0:

- use r_list_newf instead of r_list_new
- use r_list_foreach instead of while-loop

Exercise: RList

extend previous exercise:

- sort the strings in alphabetical order
  - Use r_list_sort

Exercise: RList

modify previous exercise:

- iterate over list and remove elements from it, if their strlen is odd
  - use r_list_delete_iter and r_list_foreach_safe
- store elements with odd strlen in another list
  - Print that list too

Exercise: RList

modify previous exercise:

- create a free()-wrapper and use it via r_list_newf
  - print the string
  - free it

RQueue

Operations:

enqueue and dequeue

RStack

Operations:

push and pop

RHashTable / SdbHt

Sdb

SDB it is a simple key=value database that can only store strings and works in memory or disk.

It is used inside r2 in different places, but not everywhere, because it doesn't fits for some use cases and the payload of strings processing and hashtable sometimes is too heavy and sometimes pays off.

- Accessible using the `k` command in r2.

What is stored in Sdb inside r2?

- Syscalls database
- Xrefs
- Metadata
- Headers information
- ROP gadgets
- Zignatures
- Analysis Hints
- Call Conventions
- ...

How to operate with a keyvalue database?

- Everything can be simplified to k=v
- Several data structures and encodings can be implemented on top of a string hashtable. Linked lists, binary containers, formatted structures, etc.
- Schema-free, initialization, translation must be handled by hand.
- Non-relational database, this must be done in code.
- Easy to map from disk to memory
- Requires duplication of data for multiple column lookups
- We can set a timeout on each key.
- See sdb.h

Why use it?

- It's simple
- It's fast
- Can be nested into namespace and inspectable from the shell at runtime
- Easy to dump and restore from disk
- Supports arrays, numbers, booleans and structs as string frontends.

RIDStorage

- RIDPool
  - counter
  - RQueue
- dynamic array

Operations:

add, set, get, delete, take and foreach

What is stored inside RIDStorage in r2?

Used to store RIODesc (filehandle):

```
R_API RIODesc* r_io_desc_get(RIO* io, int fd) {
        if (!io || !io->files) {
                return NULL;
        }
        return (RIODesc*) r_id_storage_get (io->files, fd);
}
```

Why use RIDStorage?

- Faster than SDB
- ids can be considered as safer pointers
  - also work for scripting

ROIDStorage

- RIDStorage
- counter
- dynamic array

Operations:

add, set, get, delete, take, insert, sort, find, to_rear, to_front, first, last and foreach

Exercise: ROIDStorage

Implement a sort callback, to sort the strings in alphabetical order

Strings

- Rstrbuf
- Rstr api
- ...

Other

- vectors
- Rbtree
- ...

RNum

RNum

The api used to convert a string into an integer and viceversa

- Supports parenthesis and complex chained expressions
- Supports name resolutions (for flags)
-

Rax2

Do some base number conversions

Practice with rahash2 and rax2

- Convert from
- Encode in base64
-

ESIL

What is ESIL

- Intermediate language
- Intended for emulation and detailed analysis
  - Every code analysis is abstract emulation (halting problem)
- Extendable

How does ESIL work?

- Parser
- Operations
- Internal Vars

ESIL: Parser

Parser works like a pushdown automaton/stack machine

ESIL: Parser

A PDA is formally defined as a 7-tuple:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$$ where

- Q is a finite set of *states*
- Σ is a finite set which is called the *input alphabet*
- Γ is a finite set which is called the *stack alphabet*
- δ is a finite subset of $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^*$, the *transition relation*
- $q_0 \in Q$ is the *start state*
- $Z \in \Gamma$ is the *initial stack symbol*
- $F \subseteq Q$ is the set of *accepting states*

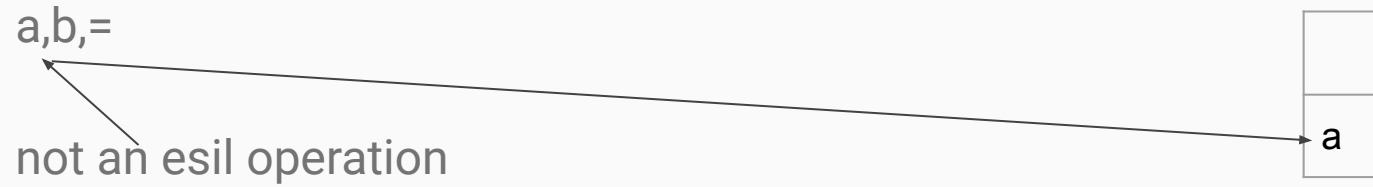# How does ESIL work?

ESIL operations
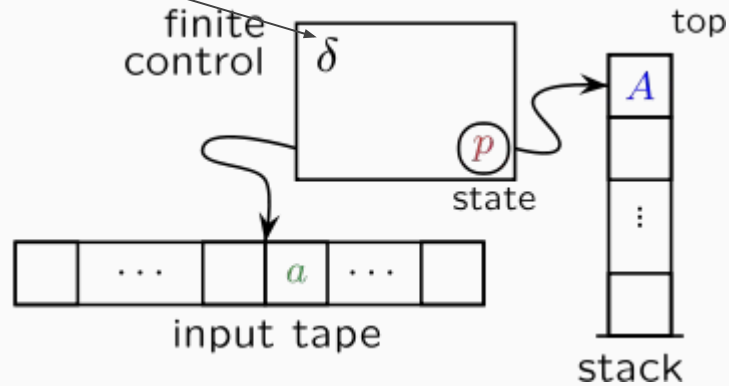
ESIL: Parser

```
while not at end of esil_string {
    cur = get_next_element()
    If cur is esil_operation {
        op = get_esil_operation(cur)
        op ()
    } else {
        push (cur)
}}
```

ESIL: Parser

a,b,=

not an esil operation

a

ESIL: Parser

b,=

b
a

not an esil operation

ESIL: Parser

=

↑

esil operation

b
a

ESIL: Operations (just a few basic ones)

- := and =
- +
- -
- *
- /
- ==
- []
- =[]
- ?{
- }

- }{
- &
- |
- ^
- !
- <<
- >>
- GOTO
- DUP
- NUM

anal-plugins can and should add their own custom operations, if needed

ESIL: Internal vars

- $ is prefix for access
- calculate flags
- updated on every operation, that sets something (==)
  - comparing old and new value of the destination

ESIL: Internal vars

			CY	H	h	Z	CYCL	7 6	5 4 3	2 1 0
CP	s	A—s	*	*	1	*	--	--	–	--

Compares the contents of operand s and register A and sets the flag if they are equal. r, n, and (HL) are used for operand s.

		CYCL	7 6	5 4 3	2 1 0
CP	r	1	10	111	r
CP	n	2	11	111	110
			← n →		
CP	(HL)	2	10	111	110

Examples: When A= 0x3C, B = 0x2F, and (HL) = 0x40,
CP B   ; Z ← 0, H ← 1, N ← 1, CY ← 0
CP 0x3C ; Z ← 1, H ← 0, N ← 1, CY ← 0
CP (HL) ; Z ← 0, H ← 0, N ← 1, CY ← 1

cp b:
b,a,==,$z,Z,=,$b4,H,=,$b8,C,=,1,N,=

cp 0x3c:
60,a,==,$z,Z,=,$b4,H,=,$b8,C,=,1,N,=

cp [hl]:
hl,[1],a,==,$z,Z,=,$b4,H,=,$b8,C,=,1,N,=

Where it is used?

- Emulation
- Search
- Assisted debugging
- Analysis
-

Testing ESIL expressions

Test for functionality, not strcmp

How to improve or fix

- Try not to add new operations to the standard set of operations
    - If really needed, create custom operations for the plugin
- Modify the esil expressions to fix behaviour
    - ahe
- Create a JIT for faster emulation
    - Main bottlenecks are rreg, rio, …
- Use recursive esil always if possible, avoid analyzing twice by keeping the state
-

Register profiles

Bug Finding

Compiler Warnings

We should know that when a compiler spits a warning is because there's something wrong.

- Maybe unexpected behaviour
- Or undefined values
- Integer overflows

Static source analysis tools

- Coverity
- Clang analyzer
- ...

CVEs

-

Backporting bug fixes policies

- We only fix bug in master, no backporting
- People should be aware of that because distros never handle CVEs or patches properly.

- So we just go forward

Fuzzing

Watch a training about fuzzers

Bug Fixing

Understanding a crash

r2pipe

R2Pipe

R2pipe is the simplest interface to automate r2, this is, taking a command, running it and getting the results.

As a bonus, as long as many commands support JSON output it allows to get a parsed native object.

> import r2pipe

> r2 = r2pipe.open("/bin/ls")

> print(r2.cmd("?V"))

R2Pipe

There are many backends we can use with r2pipe, some of them are local, others remote, some requires native code to be executed, .. we should choose when we need some performance boost.

- Tcp
- Http
- Native
- Popen
- Socket File
- ...

R2pipe.html

This is a simolified usage of the webserver to run a single file by replacing yhr path of the index file.

- Supports running r2 commands from the web browser
- Can render stuff in html/css
- Interactive

EXERCISE

Write an r2pipe script and compare times of execution.

Extensibility and Plugins

Plugins in C

We have seen that all the codebase is written in C, so all the plugins can be written in this language.

But as long as it's a pretty low level language it is very easy to interface with other native languages.

- Rust, Go, ..
- Need to expose the radare_plugin struct symbol

Plugins in Python

It is also possible to write plugins in python, lets see some examples and play with them

- They make startup times of r2 quite high
- Better load them only when needed
- You can use all the apis from python

Scripts

Scripts can be executed with r2 -i or the . Command

- Checks the extension and picks the best way to run them
- May be just plain r2 script file
- Or python, nodejs …

#!pipe

When executing a program via #!pipe (or !*) we get the r2pipe environment loaded for the program to use the current r2 instance

Using C as scripting

The #!c command will compile as shared lib the given .c, load it in memory and run the constructor. This allows us to load code at runtime, which runs fast and requires no bindings.

Building code that links to r2

In order to make things portable you may want to use some conventions and standard tools to get paths and compiler flags

- Use pkg-config
- Use r2 -HH
- Make/meson/cmake
- Package it for r2pm

RIO

What does RIO do?

- abstraction for filehandling (malloc, zip, ptrace ...)
- remapping in virtual address-space
- 2 layers of caches
- masking

How IO works

Well, magic (out of scope for this training) :P

RIO

- fd api
  - no need to use desc api
  - see example
- highlevel api
  - In most cases you only need r_io_read_at

RAsm/RAnal

Writing disassembler/analysis plugins

Right now disasm/asm api is shipped in a separate plugin than analysis.

- Analysis plugins provide basic opcode and arch information to have a single generic analysis loop
- They will be merged at some point, reduce instruction decoding times
- Assembler can do one instruction or a bunch of them
- Disassembler must be used linearly for one instruction at a time because the flow is not defined by it

Write an asm/anal plugin

What about using the Via C3 risc processor?

I think plugin for crvm should be fine for this

There are a bunch of issues in github asking for support for more archs.

That's part of the contribution challenge which have some nice prizes!

RBin

RBin

RBin code structure

Which are the apis exposed by the rbin plugins to work?

Which are the apis exposed by rbin?

What are segments and sections?

RBin strings

rabin2 -z show strings, but there are other ways to get strings

- anal.strings
- Define string constraints
    - Support multiple encodings
    - Prefix string with null byte
    - Checking for pascal strings
    - Nsstring/cfstring..

Packages

What is r2pm

A package manager that is shipped with radare2.

Uses git and shellscript files to define the packages

- Handle dependencies
- Installs in home by default
- Focus on ease of use
- Most packages are r2 plugins

Making it available in r2pm

In order to make the pkg available on r2pm you need to add an extra repository ir just make a pullreq into the official one with your script in there.

- r2pm update
- See r2pm env vars to change the path of the pkg db

Future

- There's a WIP re-implementation in Go
- We can't depend on posix shell for the packages if we aim to be portable and dependency-free.
  - For windows
- Do not depend on posix shell to parse the file probably using yaml would make more sense

Questions