Manticore Documentation

Release 0.1.0

Trail of Bits

Aug 24, 2018

Contents:

1 API Reference **3**
- 1.1 Helpers . . . 3
- 1.2 Manticore . . . 3
- 1.3 State . . . 3
- 1.4 SLinux . . . 3
- 1.5 Cpu . . . 3
- 1.6 Models . . . 3
- 1.7 EVM . . . 3

2 Symbolic Input **5**
- 2.1 Wildcard byte . . . 5
- 2.2 Symbolic arguments/environment . . . 5
- 2.3 Symbolic stdin . . . 5
- 2.4 Symbolic file input . . . 6
- 2.5 Symbolic sockets . . . 6

3 Function Models **7**

4 Gotchas **9**
- 4.1 Mutable context entries . . . 9
- 4.2 Context locking . . . 9
- 4.3 “Random” Policy . . . 10

5 Indices and tables **11**

Manticore is a symbolic execution tool for analysis of binaries and smart contracts.

CHAPTER 1

API Reference

This API is under active development, and should be considered unstable.

1.1 Helpers

1.2 Manticore

1.3 State

1.4 SLinux

Symbolic Linux

1.5 Cpu

1.6 Models

1.7 EVM

CHAPTER 2

Symbolic Input

Manticore allows you to execute programs with symbolic input, which represents a range of possible inputs. You can do this in a variety of manners.

2.1 Wildcard byte

Throughout these various interfaces, the '+' character is defined to designate a byte of input as symbolic. This allows the user to make input that mixes symbolic and concrete bytes (e.g. known file magic bytes).:

For example: "concretedata++++++++moreconcretedata++++++++++"

2.2 Symbolic arguments/environment

To provide a symbolic argument or environment variable on the command line, use the wildcard byte where arguments and environment are specified.:

```
$ manticore ./binary +++++ +++++
$ manticore ./binary --env VAR1=+++++ --env VAR2=++++++
```

For API use, use the `argv` and `envp` arguments to the `manticore.Manticore.linux()` classmethod.:

```
Manticore.linux('./binary', ['++++++', '++++++'], dict(VAR1='+++++', VAR2='++++++'))
```

2.3 Symbolic stdin

Manticore by default is configured with 256 bytes of symbolic stdin data, after an optional concrete data prefix, which can be provided with the `concrete_start` kwarg of `manticore.Manticore.linux()`.

2.4 Symbolic file input

To provide symbolic input from a file, first create the files that will be opened by the analyzed program, and fill them with wildcard bytes where you would like symbolic data to be.

For command line use, invoke Manticore with the `--file` argument.:

```
$ manticore ./binary --file my_symbolic_file1.txt --file my_symbolic_file2.txt
```

For API use, use the `add_symbolic_file()` interface to customize the initial execution state from an `init()` hook.:

```
@m.init
def init(initial_state):
    initial_state.platform.add_symbolic_file('my_symbolic_file1.txt')
```

2.5 Symbolic sockets

Manticore's socket support is experimental! Sockets are configured to contain 64 bytes of symbolic input.

CHAPTER 3

Function Models

The Manticore function modeling API can be used to override a certain function in the target program with a custom implementation in Python. This can greatly increase performance.

Manticore comes with implementations of function models for some common library routines (core models), and also offers a user API for defining user-defined models.

To use a core model, use the `invoke_model()` API. The available core models are documented in the API Reference:

```
from manticore.models import strcmp
addr_of_strcmp = 0x400510
@m.hook(addr_of_strcmp)
def strcmp_model(state):
    state.invoke_model(strcmp)
```

To implement a user-defined model, implement your model as a Python function, and pass it to `invoke_model()`. See the `invoke_model()` documentation for more. The core models are also good examples to look at and use the same external user API.

CHAPTER 4

Gotchas

Manticore has a number of "gotchas": quirks or little things you need to do in a certain way otherwise you'll have crashes and other unexpected results.

4.1 Mutable context entries

Something like `m.context['flag'].append('a')` inside a hook will not work. You need to (unfortunately, for now) do `m.context['flag'] += ['a']`. This is related to Manticore's built in support for parallel analysis and use of the *multiprocessing* library. This gotcha is specifically related to this note from the Python documentation :

> "Note: Modifications to mutable values or items in dict and list proxies will not be propagated through the manager, because the proxy has no way of knowing when its values or items are modified. To modify such an item, you can re-assign the modified object to the container proxy"

4.2 Context locking

Manticore natively supports parallel analysis; if this is activated, client code should always be careful to properly lock the global context when accessing it.

An example of a global context race condition, when modifying two context entries.:

```
m.context['flag1'] += ['a']
--- interrupted by other worker
m.context['flag2] += ['b']
```

Client code should use the `locked_context()` API:

```
with m.locked_context() as global_context:
    global_context['flag1'] += ['a']
    global_context['flag2'] += ['b']
```

4.3 "Random" Policy

The *random* policy, which is the manticore default, is not actually random and is instead deterministically seeded. This means that running the same analysis twice should return the same results (and get stuck in the same places).

CHAPTER 5

Indices and tables

- genindex
- modindex
- search