Manticore API

Author: Josselin Feist, josselin@trailofbits.com

The aim of this document is to show how to use Manticore to automatically find bugs in smart contracts.

my_token.py is an example of code using most of the API of this document.

Need help? Slack: https://empireslacking.herokuapp.com/ #manticore

The following details how to manipulate a smart contract through the Manticore API:

- How to create accounts
- How to execute transactions
- How to generate test-case
- How to read Manticore results
- How to access state information
- How to add constraints

Creating Accounts

The first thing to do on a script is to initiate a new blockchain:

```
from manticore.ethereum import ManticoreEVM

m = ManticoreEVM()
```

A non-contract account is created using m.create_account:

```
user_account = m.create_account(balance=1000)
```

A Solidity contract can be deployed using m.solidity_create_contract:

```
with open('token.sol') as f:
   source_code = f.read()
# Initiate the contract
contract_account = m.solidity_create_contract(source_code,
owner=user_account)
```

Summary

- **You can create user and contract accounts**

Executing transactions

Manticore supports two types of transaction:

- Raw transaction: all the functions are explored
- Named transaction: only one function is explored

Raw transaction

A raw transaction is executed using m.transaction:

```
m.transaction(caller=user_account,
              address=contract_account,
              data=data,
              value=value)
```

The caller, the address, the data, or the value of the transaction can be either concrete or symbolic:

- `m.make_symbolic_value` creates a symbolic value.
- `m.make_symbolic_buffer(size)` creates a symbolic byte array.

For example:

```
symbolic_value = m.make_symbolic_value()
symbolic_data = m.make_symbolic_buffer(320)
m.transaction(caller=user_account,
              address=contract_account,
              data=symbolic_data,
              value=symbolic_value)
```

If the data is symbolic, Manticore will explore all the functions of the contract during the transaction execution (see the Fallback Function section of the Hands on the Ethernaut CTF article to understand how the function selection works)

Named transaction

Functions can be executed through their name.
To execute `f(uint var)` with a symbolic value, from `user_account`, and with 0 ether, use:

```
symbolic_var = m.make_symbolic_value()
contract_account.f(symbolic_var, caller=user_account, value=0)
```

If `value` of the transaction is not specified, it is 0 by default.

Summary

- **Arguments of a transaction can be concrete or symbolic**
- **A raw transaction will explore all the functions**
- **Function can be called by their name**

Generating test-case

`m.generate_testcase(state, name)` generates a test-case from a state:

```
m.generate_testcase(state, 'NameTestCase')
```

Summary

- **`M.generate_testcase` generate the testcase of a state**

Reading the results

`m.workspace` is the directory used as output directory for all the files generated:

```
print("Results are in {}".format(m.workspace))
```

The workspace directory contains:

- “test_XXXXX.tx”: detailed list of transactions per test-case
- “global.summary”: coverage and compiler warnings
- “test_XXXXX.summary”: coverage, last instruction, account balances per test case

Summary

- **In the workspace directory, test_XXXXX.tx files are the test-case files.**

Accessing state Information

Each path executed has its state of the blockchain. The list of all the states can be iterated using [m.all_states]:

```
for state in m.all_states:
    # do something with m
```

You can access state information, for example:

- `state.platform.get_balance(account.address):` the balance of the account
- `state.platform.transactions`: the list of transactions
- `state.platform.transactions[-1].return_data:` the data returned by the last transaction

The data returned by the last transaction is an array, which can be converted to a value with `ABI.deserialize`, for example:

```
data = state.platform.transactions[0].return_data
data = ABI.deserialize("uint", data)
```

Summary

- **You can iterate over the state with `m.all_states`**
- **`state.platform.get_balance(account.address)` returns the account's balance**
- **`state.platform.transactions` returns the list of transactions**
- **`transaction.return_data` is the data returned**

Adding Constraints

Arbitrary constraints can be added to a state.

Operators

The [Operators](#) module facilitates the manipulation of constraints, among other it provides:

- Operators.AND,
- Operators.OR,
- Operators.UGT (unsigned greater than),
- Operators.UGE (unsigned greater than or equal to),
- Operators.ULT (unsigned greater than),
- Operators.ULE (unsigned greater than or equal to).

The module is imported with :

```
from manticore.core.smtlib import Operators
```

State Constraint

[state.constrain(constraint)](#) will constrain the state with the boolean constraint.

Checking Constraint

`state.is_feasible()` checks if the constraints of the state are feasible.

For example, the following will constraint `symbolic_value` to be different from 65 and check if the state is still feasible:

```
state.constrain(symbolic_var != 65)
if state.is_feasible():
    # state is feasible
```

Summary

- **`state.constraint` add arbitrary constraint**
- **`state.is_feasible()` checks if the constraint are feasible**
- **The `Operators` module facilitates writing constraint**