Manticore Training

Presenter: Josselin Feist, josselin@trailofbits.com

Requirements:

- Basic Python knowledge
- git clone https://github.com/trailofbits/trufflecon-2019

Each exercise corresponds to automatically find a vulnerability with a Manticore script. Once the vulnerability is found, Manticore can be applied to a fixed version of the contract, to demonstrate that the bug is effectively fixed.

The solution presented during the workshop will be based on the proposed scenario of each exercise, but other solutions are possible. Each scenario is composed of three steps: the initialization, the exploration and the check of properties.

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

Installation

Manticore requires >= python 3.6. It can be installed through pip or using docker.

Manticore through docker

```
docker pull trailofbits/eth-security-toolbox
docker run -it -v "$PWD":/home/trufflecon trailofbits/eth-security-toolbox
```

The last command runs eth-security-toolbox in a docker that has access to your current directory. You can change the files from your host, and run the tools on the files from the docker

Inside docker, run:

```
solc-select 0.5.3
cd /home/trufflecon/
```

To run a python script with python 3:

```
python3 script.py
```

Manticore through pip

```
pip install --user manticore
```

solc 0.5.3 is recommended.

Manticore: API Basics

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:

```
# Initiate the contract
with open('token.sol') as f:
      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**

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
```

`m.ready_states` will return the list of states that are alive (they did not execute a REVERT/INVALID)

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.

Generating test-case

`m.generate_testcase(state, name, condition)` generates a test-case from a state, if the `condition` is true:

```
m.generate_testcase(state, 'NameTestCase', only_if=condition)
```

Summary

- **`state.constraint` add arbitrary constraint**
- **`state.is_feasible()` checks if the constraint are feasible**
- **The `Operators` module facilitates writing constraint**
- **`m.generate_testcase` generate the testcase of a state**

Example: Incorrect token transfer

This scenario is given as an example. You can follow its structure to solve the following exercises.

my_token.py uses manticore to find for an attacker to generate tokens during a transfer on `Token` (my_token.sol).

Proposed scenario

Initialization:

- Create one user account
- Create the contract account

Exploration:

- Call `balances` on the user account
- Call transfer with symbolic destination and value
- Call `balances` on the user account

Property:

- Check if the user can have more token after the transfer than before.

Exercice 1 : Arithmetic

Use Manticore to find an input allowing an attacker to generate free tokens.
Propose a fix of the contract, and test your fix using your Manticore script.

Proposed scenario

Initialization:

- Create one account
- Create the contract account

Exploration:

- Call is_valid_buy with two symbolic values for `tokens_amount` and `wei_amount`

Property:

- An attack is found if on a state alive:
  - `wei_amount == 0`
  - `tokens_amount >= 1`

```
/// @notice Check if a buy is valid
   /// @param tokens_amount tokens amount
   /// @param wei_amount wei amount
   function is_valid_buy(uint tokens_amount, uint wei_amount) external
view returns(bool){
       _valid_buy(tokens_amount, wei_amount);
       return true;
   }


   /// @notice Mint tokens
   /// @param addr The address holding the new token
   /// @param value The amount of token to be minted
   /// @dev This function performed no check on the caller. Must stay
internal
```

```
    function _mint(address addr, uint value) internal{
        balances[addr] = safeAdd(balances[addr], value);
        emit Mint(addr, value);
    }


    /// @notice Compute the amount of token to be minted. 1 ether = 10
tokens
    /// @param desired_tokens The number of tokens to buy
    /// @param wei_sent The ether value to be converted into token
    function _valid_buy(uint desired_tokens, uint wei_sent) internal view{
        uint required_wei_sent = (desired_tokens / 10) * decimals;
        require(wei_sent >= required_wei_sent);
    }
```

Figure 1: [token.sol]()

Hints:

- `m.ready_states` returns the list of state alive
- `Operators.AND(a, b)` allows to create and AND condition

Exercice 2 : Arithmetic on multi transactions (Bonus)

Use Manticore to find if an overflow is possible in Overflow.add. Propose a fix of the contract, and test your fix using your Manticore script.

Proposed scenario

Initialization

- Create one user account
- Create the contract account

Exploration

- Call two times add with two symbolic values
- Call sellerBalance()

Property:

- Check if it is possible for the value returned by sellerBalance() to be lower than the first input.

```
pragma solidity^0.4.24;
contract Overflow {
```

```
    uint public sellerBalance=0;

    function add(uint value) public returns (bool){
        sellerBalance += value; // complicated math, possible overflow
    }
}
```

Figure 2: [overflow.sol]()

Hints:
The value returned by the last transaction can be accessed through:

```
state.platform.transactions[-1].return_data
```

The data returned needs to be deserialized:

```
data = ABI.deserialize("uint", data)
```