Introduction to Smart Contract Analysis with Manticore

Presenters:
- Josselin Feist: josselin@trailofbits.com

Requirements:
- Basic Python knowledge
- Manticore:
    - docker pull trailofbits/manticore
    - git clone https://github.com/trailofbits/trufflecon
    - cd trufflecon/manticore
    - docker run -it -v $PWD:/home/manticore/trufflecon trailofbits/manticore
    - cd trufflecon

All the material of this workshop are available in XXXX.
It is recommended to use Manticore v0.2.1.1.

The aim of this document is to show how to use Manticore to automatically find bugs in smart contracts. The goal of the workshop is to solve the exercises proposed in Section 4. Section 1 to 3 introduce the basic features of Manticore needed for this purpose.

Slack: https://empireslacking.herokuapp.com/ #manticore

1. Running under Manticore

We will see how to explore a smart contract with the Manticore API. The target is the following smart contract:

```
pragma solidity^0.4.24;
contract Simple {
   function f(uint a) payable public{
        if (a == 65) {
            revert();
        }
   }
}
```

Figure 1: example.sol

Standalone Exploration

Manticore can be run directly on the smart contract::

```
$ manticore simple.sol
...
... m.ethereum:INFO: Generated testcase No. 0 - STOP
... m.ethereum:INFO: Generated testcase No. 1 - REVERT
... m.ethereum:INFO: Generated testcase No. 2 - RETURN
... m.ethereum:INFO: Generated testcase No. 3 - REVERT
... m.ethereum:INFO: Generated testcase No. 4 - REVERT
... m.ethereum:INFO: Generated testcase No. 5 - STOP
... m.ethereum:INFO: Generated testcase No. 6 - REVERT
... m.ethereum:INFO: Results in /path/examples/mcore_Pj3Aqa
```

Figure 2 : Run of Manticore (the order of the testcases may change)

Without additional information, Manticore will explore the contract with new symbolic transactions until it does not explore new paths on the contract. Manticore does not run new transactions after a failing one (e.g: after a revert).

Manticore will output the information in a mcore_* directory. Among other, you will find in this directory:

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

Here Manticore founds 6 test cases, which correspond to (the filename order may change):

	Transaction 1	**Transaction 2**	**Transaction 3**	**Result**
test_00000000.tx	Contract creation			`RETURN`
test_00000001.tx	Contract creation	f(65)		`REVERT`
test_00000002.tx	Contract creation	fallback function		`REVERT`
test_00000003.tx	Contract creation	f(!=65)		`STOP`

test_00000004.tx	Contract creation	f(!=65)	fallback function	REVERT
test_00000005.tx	Contract creation	f(!=65)	f(!=65)	STOP

Figure 3 : Exploration summary
f(!=65) denotes f called with any value different than 65

As you can notice, Manticore generates an unique test case for every successful transaction.

Blockchain and Accounts

The following details how to manipulate a smart contract through the Manticore Python API. The first thing to do is to initiate a new blockchain:

```
from manticore.ethereum import ManticoreEVM

m = ManticoreEVM()
```

New accounts can be created. 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`:

```
source_code = '''
pragma solidity^0.4.24;
contract Simple {
    function f(uint a) payable public {
        if (a == 65) {
            revert();
        }
    }
}
'''
# Initiate the contract
contract_account = m.solidity_create_contract(source_code,
owner=user_account)
```

Transactions

A raw transaction can be 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, and `m.make_symbolic_buffer(size)` creates a symbolic byte array:

```
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. To understand how the function selection works, see the *Fallback Function* explanation in the Hands on the Ethernaut CTF article.

Manticore also allows the user to execute only one specific function. For example, 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.

Workspace

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

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

Terminate the Exploration

`m.finalize()` stops the exploration. No further transactions should be sent once this method is called and Manticore generates test cases for each of the path explored.

Summary: Running under Manticore

Putting all the previous steps together, we obtain:

```
from manticore.ethereum import ManticoreEVM


m = ManticoreEVM()


with open('example.sol') as f:
   source_code = f.read()


user_account = m.create_account(balance=1000)
contract_account = m.solidity_create_contract(source_code,
                                              owner=user_account)


symbolic_var = m.make_symbolic_value()
contract_account.f(symbolic_var)


print("Results are in {}".format(m.workspace))
m.finalize() # stop the exploration
```

Figure 4: example_run.py

2. Getting Throwing Path

We will now improve the previous example and generate specific inputs for the paths raising an exception in `f()`.

State Information

Each path executed has its state of the blockchain. A state is either alive or it is terminated, meaning that it reaches a THROW or REVERT instruction.
The list of alive states can be accessed through `m.running_states`. The list of terminated states (e.g: states that reached an error) can be accessed using `m.terminated_states`. The list of all the states can be accessed using `m.all_states`:

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

From a specific state, information can be accessed. For example:

- `state.platform.transactions`: the list of transactions
- `state.platform.transactions[-1].return_data`: the data returned by the last transaction

On a transaction, `transaction.result` returns the result of the transaction, which can be `'RETURN'`, `'STOP'`, `'REVERT'`, `'THROW'` or `'SELFDESTRUCT'`. `transaction.data` return the data of the transaction.

Generating testcase

`m.generate_testcase(state, name)` generates a testcase from a state:

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

Summary: Getting Throwing Path

```
from manticore.ethereum import ManticoreEVM


m = ManticoreEVM()


with open('example.sol') as f:
   source_code = f.read()


user_account = m.create_account(balance=1000)
contract_account = m.solidity_create_contract(source_code,
                                              owner=user_account)


symbolic_var = m.make_symbolic_value()
contract_account.f(symbolic_var)


## Check if an execution ends with a REVERT or INVALID
for state in m.terminated_states:
   last_tx = state.platform.transactions[-1]
   if last_tx.result in ['REVERT', 'INVALID']:
       print('Throw found {}'.format(m.workspace))
       m.generate_testcase(state, 'ThrowFound')
```

Figure 5: [example_throw.py]

Note we could have generated a much simpler script, as all the states returned by

`terminated_state` have REVERT or INVALID in their result: this example was only meant to demonstrate how to manipulate the API.

3. Adding Constraints

We will now see how to constrain the exploration. We will make the assumption that the documentation of `f()` states that the function is never called with a == 65, so any bug with a ==65 is not a real bug.

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).

`Operators.CONCAT` is used to concates an array to a value. For example, the `return_data` of a transaction needs to be changed to a value to be checked against another value:

```
last_return = Operators.CONCAT(256, *last_return)
```

The module is imported with :

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

Global constraint

`m.constrain(constraint)` will constraint all the current state with the boolean constraint. For example, you can call a contract from a symbolic address, and restraint this address to be specific values:

```
symbolic_address = m.make_symbolic_value()
m.constraint(Operators.OR(symbolic == 0x41, symbolic_address == 0x42))
m.transaction(caller=user_account,
              address=contract_account,
              data=m.make_symbolic_buffer(320) ,
              value=0)
```

State Constraint

`state.constrain(constraint)` will constrain the state with the boolean `constraint`.
It can be used to constrain the state after its exploration to check some property on it.

Checking Constraint

`solver.check(state.constraints)` will check if the constraints of a state are still 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 solver.check(state.constraints):
    # state is feasible
```

Summary: Adding Constraints

```
from manticore.ethereum import ManticoreEVM
from manticore.core.smtlib import solver


m = ManticoreEVM()


with open('example.sol') as f:
   source_code = f.read()


user_account = m.create_account(balance=1000)
contract_account = m.solidity_create_contract(source_code,
                                              owner=user_account)


symbolic_var = m.make_symbolic_value()
contract_account.f(symbolic_var)


## Check if an execution ends with a REVERT or INVALID
for state in m.terminated_states:
   last_tx = state.platform.transactions[-1]
```

```
    if last_tx.result in ['REVERT', 'INVALID']:
        # we do not consider the path were a == 65
        state.constrain(symbolic_var != 65)
        if solver.check(state.constraints):
            print("Bug found in {}".format(m.workspace))
            m.generate_testcase(state, 'BugFound')
```

Figure 6: example_constraint.py

4. Exercises

The two following exercises are meant to be solved using the features provided in the previous Sections.

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.

Integer overflow

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:

- Generate one user account
- Generate the contract account
- Call two times `add` with two symbolic values
- Call `sellerBalance()`
- 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 7:overflow.sol

Hints:
Recall from Section 2 that the value returned by the last transaction can be accessed through:

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

Recall from Section 3, `return_data` needs to be concatened:

```
return_data = Operators.CONCAT(256, *last_return)
```

Recall from Section 3, to add a constraint `a > b` on two unsigned integers use:

```
state.constrain(Operators.UGT(a, b))
```