Introduction to Smart Contract Analysis with Manticore

Presenters:

- Josselin Feist: josselin@trailofbits.com
- JP Smith: jp@trailofbits.com

Requirements:

- Basic Python knowledge
- Manticore:
    - git clone https://github.com/trailofbits/manticore.git
    - cd manticore
    - git checkout fbf0823cb61aa7f3b417839a4e2e7e3f58352f10
    - pip2 install --user .
- solc in version 0.4.20.

All the material of this workshop are available in https://github.com/trailofbits/workshops/tree/master/Manticore - EthCC 2018 .

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 two exercices 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^0.4.20;
contract Simple {
    function f(uint a) payable public{
        if (a == 65) {
            revert();
        }
    }
}
```

Figure 1: simple.sol

Before starting to manipulate the Python API, you can run manticore directly on the contract:

```
$ manticore simple.sol
```

As shown during the presentation, 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

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^0.4.20;
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 data and the value of the transaction can be either concrete or symbolic. `m.SValue` expresses a symbolic value, and `m.SByte(size)` expresses a symbolic byte array:

```
symbolic_value = m.SValue
symbolic_data = m.SByte(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 a specific function. For example, to execute `f(uint a)` with a symbolic value, from `user_account`, and with 1 ether (10**18 wei), use:

```
contract_account.f(m.Svalue, caller=user_account, value=10**18)
```

If value is not specified, it is symbolic by default.

Workspace

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

```
print "Results are in %s" % m.workspace
```

Terminate the Exploration

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

Summary: Running under Manticore

Putting all the previous steps together, we obtain:

```
from manticore.ethereum import ManticoreEVM
```

```
# initiate the blockchain
m = ManticoreEVM()
source_code = '''
pragma solidity^0.4.20;
contract Simple {
    function f(uint a) payable public {
        if (a == 65) {
            revert();
        }
    }
}
'''

# Initiate the accounts
user_account = m.create_account(balance=1000)
contract_account = m.solidity_create_contract(source_code,
owner=user_account, balance=0)

# Call f(a), with a symbolic value
contract_account.f(m.SValue, caller=user_account)

print "Results are in %s" % m.workspace
m.finalize() # stop the exploration
```

Figure 2: simpleRun.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. In the following exercises we will need:

- State.platform.transactions: list of transactions
- state.platform.get_balance(addr): return the balance of the account addr
- State.platform.last_return_data: the value 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'.

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

# initiate the blockchain
m = ManticoreEVM()
source_code = '''
pragma solidity^0.4.20;
contract Simple {
    function f(uint a) payable public {
        if (a == 65) {
            revert();
        }
    }
}
'''

# Initiate the accounts
user_account = m.create_account(balance=1000)
contract_account = m.solidity_create_contract(source_code,
owner=user_account, balance=0)

# Call f(a), with a symbolic value
contract_account.f(m.SValue, caller=user_account)

     # 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 "Error found in f() execution (see %s)"%m.workspace
             m.generate_testcase(state, 'BugFound')
```

Figure 3: simpleThrow.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.

Inputs

`state.input_symbols` contains the list of symbolic inputs. For example, if `f(uint a)` is called two times with a symbolic a, `state.input_symols` will contain two symbolic inputs.

Recall that the input in EVM holds in its first 4 bytes the function identifier (see the Fallback Function explanation in the Hands on the Ethernaut CTF article for more information).

Operators

Symbolic values can be manipulated easily to generate constraints using `Operators`.
For example, to extract a particular parameter of a transaction input, `Operators.CONCAT` can be used:

```
from manticore.core.smtlib import Operators
# ...
# return the first symbolic input
input0 = state.input_symbols[0]
# skip the function id, and extract the 32 bytes corresponding to the first parameter
input0 = Operators.CONCAT(256, *input0[4:36])
```

To compare a > b on two unsigned integers, use `Operators.UGT(a, b).`

State Constraint

state.constrain(constraint) will constrain the state with the boolean constraint. solver.check(state.constraints) will check if the constraints of a state are still feasible. For example, the following will constraint input0 to be different from 65 and check if the state is still feasible:

```
state.constrain(input0 != 65)
if not solver.check(state.constraints):
    # state not feasible
```

Summary: Adding Constraints

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

m = ManticoreEVM() # initiate the blockchain
source_code = '''
pragma solidity^0.4.20;
contract Simple {
    function f(uint a) payable public {
        if (a == 65) {
            revert();
        }
    }
}
'''

# Initiate the accounts
user_account = m.create_account(balance=1000)
contract_account = m.solidity_create_contract(source_code,
owner=user_account, balance=0)

# Call f(a), with a symbolic value
contract_account.f(m.SValue, caller=user_account)

## 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']:

        # return the first symbolic input
        input0 = state.input_symbols[0]
        # skip the function id, and extract the 32 bytes corresponding
to the first parameter
        input0 = Operators.CONCAT(256, *input0[4:36])


    # we do not consider the path were a == 65
    state.constrain(input0 != 65)
    if not solver.check(state.constraints):
         print "Error found in infeasible path"
         continue

    print "Error found in f() execution (see %s)"%m.workspace
    m.generate_testcase(state, 'BugFound')
```

Figure 4: simpleConstraint.py

4. Exercices

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

Each exercice 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.

Unprotected function

Find with Manticore an input allowing an attacker to steal ethers from the `UnprotectedWallet.` Propose a fix of the contract, and test your fix using your Manticore script.

Proposed scenario:

- Generate two accounts, one for the creator (with 10 ether), one for the attacker (with 0 ether)
- Generate the contract
- Creator calls deposit() with 1 ether

- Attacker calls two functions (explore any function, using raw transactions)
- An attack is found if for one of the path, the attacker’s balance has 1 ether.

The contract:

```
pragma solidity ^0.4.20;
contract UnprotectedWallet{
    address public owner;

    modifier onlyowner {
        require(msg.sender==owner);
        _;
    }
    function UnprotectedWallet() public {
        owner = msg.sender;
    }
    function changeOwner(address _newOwner) public {
       owner = _newOwner;
    }

   function deposit() payable public { }

   function withdraw() onlyowner public {
       msg.sender.transfer(this.balance);
   }
}
```

Figure 4: unprotectedWallet.sol

Hints:

Recall from Section 2 that the balance of a user can be accessed through:

```
state.platform.get_balance(addr)
```

Recall from Section 3, to add a constraint a > b use:

```
state.constrain(a > b)
```

Recall from Section 3, the constraints of state are still feasible if the following returns `True`:

```
solver.check(state.constraints)
```

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

```
pragma solidity^0.4.20;
contract Overflow {
    uint public sellerBalance=0;

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

Figure 5: overflow.sol

Hints:

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

```
state.platform.last_return_data
```

To add constraints between variables, you need to compare variables in the same format; a simple way to convert an array to an int is to use:

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

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

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