HERC CONTRACT SECURITY ASSESSMENT

SECURITY VULNERABILITY FINDINGS AND RECOMMENDATIONS

Confidentially prepared for:

HERC.ONE

Logan Ryan Golema
Chief Technology officer
AGLD || HERC
+15126059663
logan@herc.one
herc.one
agld.com

Prepared by:

TLS Consultants

JUNE 20, 2018

TABLE OF CONTENTS

SECTION A: ENGAGEMENT INFORMATION

#	Fields	Engagement Classification Data
1	Organization	HERC
3	Sponsor	Logan Ryan Golema
4	Assessment Name	HERC Contract Security Assessment
6	Consultant	Trusted Ledger Solutions
7	Assessment Start	February 7th, 2018
8	Assessment End	June 20th, 2018
#	Status	Engagement Objectives
1	Completed - 02/12/2018	Interview Devs. Obtain development environment and baseline
2	Complete	Review .sol files
3	Complete	Compilation
4	Complete	Manual Code flow dissection
5	Complete	Manual vulnerability checking

SECTION B: EXECUTIVE SUMMARY

HERC.ONE engaged TRUSTED LEDGER SOLUTIONS to perform a Contract Security Assessment of the HERC.ONE contracts to identify weaknesses in the contracts. The Assessment was conducted between February 7th 2018, and June 20th, 2018. TRUSTED LEDGER SOLUTIONS assessed the application using code provided over multiple drops. Multiple iterations of code assessments were performed. The final drop was delivered June 1st. A final manual review of the code was performed and is reported herein.

TRUSTED LEDGER SOLUTIONS determined that HERC.ONE's solidity contract code is at a moderate to high level of risk based on potential vulnerabilities and lack of test cases. TRUSTED LEDGER SOLUTIONS identified high and moderate vulnerabilities within the contracts. TRUSTED LEDGER SOLUTIONS recommends remediation of the high-risk vulnerability, review of current development processes to reduce the likeliness of deploying high-risk application-layer vulnerabilities.

Contract Review Process

This Contract Review is comprised of a manual review of the application code base. The final review did not include dynamic tools of the contract based on scoping provided by the customer. The assessment uses a hybrid approach consisting of manual review and automated testing with the goal of identifying any weaknesses in the source code of the application that can be abused by an attacker. A Contract Review is not going to identify all weaknesses that may exist but is an important part of the overall application security review.

The following methodology was used:

- Interview Devs
- Review .sol files
- Compilation via local test chain
- Code flow dissection
- Automated analysis*
- Manual verification of vulnerabilities
- Manual vulnerability checking.

* - previous versions of the HERC token

Scope

The specified scope for this report was defined as follows: Looking for Reentrancy, Uint Tests, as well as any Gas deficiencies and deployment success.

	Application	Checksum
1	HERCToken.sol	MD5 (Addresses.sol) = 38c237e9ca972848c91470a80e249ece
2	HERCDistribution.sol	MD5 (Crowdsale.sol) = 70ccdc96ac2d219a0dee9469667fccbf
3	Crowdsale.sol	MD5 (HERCDistribution.sol) = 274802e3e1305e73a84c3a10f571e94e
4	Addresses.sol	MD5 (HERCToken.sol) = 7ccf32e04b53dc506308e1ae7e2276ab
5	assetCoreFunctions.sol	MD5 (AssetCoreFunctions.sol) = c49683da736e7ba3b75a3ca245fcb7b3
6	assetFees.sol	MD5 (AssetFees.sol) = 813e37cc73e09b5a49c278e6e5974931
7	Playerscore.sol	MD5 (PlayerScore.sol) = 7efc86a6e5683f1bdcd603ad1af67b81

Looking at < 600 lines of Solidity due to formatting and commenting throughout this whole process and most of the contracts have precompiled test scripts for the audit.

Findings Summary

1. TRUSTED LEDGER SOLUTIONS observed code that did not follow the **Check-Effects Interaction Pattern** - which may lead to Reentrancy. This is a ***HIGH*** security risk.

2. TRUSTED LEDGER SOLUTIONS identified insufficient test cases that do not test for edge cases or transactions involving ERC223 contracts. This is a ***HIGH*** security risk.

3. TRUSTED LEDGER SOLUTIONS identified areas where month is being used as time dependence. This is a ***MODERATE*** security risk as it may produce unexpected behavior.

Note: in testing for unintended Gas Exhaustion, it was noted that there was no use of arrays or loops except in the game, where the array is limited to of size 5. This is decreases risk.

Recommendations Summary

TRUSTED LEDGER SOLUTIONS recommends that HERC.ONE prioritize remediation efforts based on the risk level of the vulnerabilities identified in the assessment. The roadmap provided below should help HERC.ONE better prioritize next steps:

1. Universally implement the Check-Effects Interaction Pattern (***HIGH PRIORITY***)
2. Increase the robustness of test cases (***HIGH PRIORITY***)
3. Ensure time dependencies are enforceable (***MODERATE PRIORIT Y***)

SECTION C: TECHNICAL FINDINGS

Potential Reentrancy	TRUSTED LEDGER SOLUTIONS observed code that did not follow the Check-Effects Interaction Pattern - which may lead to Reentrancy

FINDINGS

TLS identifed several sections of code that failed to implmeent the Checks-Effects-Interaction pattern

Line 82 Crowdsale.sol

```
function buyFor(address beneficiary)
    public
    available
    valid(beneficiary, msg.value)
    payable {
    uint amount = msg.value.div(_price);
    _token.transfer(beneficiary, amount);
    _available = _available.sub(amount);
    _limits[beneficiary] = _limits[beneficiary].add(amount);
    Buy(beneficiary, amount);
}
```

Line 56 and 82 of HercToken.sol

```
function transfer(address _to, uint _value, bytes _data)
    public
    returns (bool) {
    if (_value > 0 &&
        _value <= _balanceOf[msg.sender]) {

        if (_to.isContract()) {
          ERC223ReceivingContract _contract = ERC223ReceivingContract(_to);
          _contract.tokenFallback(msg.sender, _value, _data);
        }

        _balanceOf[msg.sender] = _balanceOf[msg.sender].sub(_value);
        _balanceOf[_to] = _balanceOf[_to].add(_value);

        return true;
    }
    return false;
}
```

```
function transferFrom(address _from, address _to, uint _value, bytes _data)
    public
    returns (bool) {
    if (_allowances[_from][msg.sender] > 0 &&
        _value > 0 &&
        _allowances[_from][msg.sender] >= _value &&
        _balanceOf[_from] >= _value) {

          _allowances[_from][msg.sender] -= _value;

          if (_to.isContract()) {
            ERC223ReceivingContract _contract = ERC223ReceivingContract(_to);
            _contract.tokenFallback(msg.sender, _value, _data);
          }

        _balanceOf[_from] = _balanceOf[_from].sub(_value);
        _balanceOf[_to] = _balanceOf[_to].add(_value);

        Transfer(_from, _to, _value);

        return true;
    }
    return false;
}
```

Line 145 of HercDistribution.sol

```
function transferFrom(address _from, address _to, uint _value, bytes _data)
    public
    returns (bool) {
    if (_allowances[_from][msg.sender] > 0 &&
        _value > 0 &&
        _allowances[_from][msg.sender] >= _value &&
        _balanceOf[_from] >= _value) {

          _allowances[_from][msg.sender] -= _value;

          if (_to.isContract()) {
            ERC223ReceivingContract _contract = ERC223ReceivingContract(_to);
            _contract.tokenFallback(msg.sender, _value, _data);
          }

        _balanceOf[_from] = _balanceOf[_from].sub(_value);
        _balanceOf[_to] = _balanceOf[_to].add(_value);

        Transfer(_from, _to, _value);

        return true;
    }
    return false;
}
```

Line 40 and 65 of Mytoken.sol

```
function transfer(address _to, uint _value, bytes _data)
    public
    returns (bool) {
    if (_value > 0 &&
        _value <= _balanceOf[msg.sender]) {

        if (_to.isContract()) {
          ERC223ReceivingContract _contract = ERC223ReceivingContract(_to);
          _contract.tokenFallback(msg.sender, _value, _data);
        }

        _balanceOf[msg.sender] = _balanceOf[msg.sender].sub(_value);
        _balanceOf[_to] = _balanceOf[_to].add(_value);

        return true;
    }
    return false;
}
```

```
function transferFrom(address _from, address _to, uint _value, bytes _data)
    public
    returns (bool) {
    if (_allowances[_from][msg.sender] > 0 &&
        _value > 0 &&
        _allowances[_from][msg.sender] >= _value &&
        _balanceOf[_from] >= _value) {

          _allowances[_from][msg.sender] -= _value;

          if (_to.isContract()) {
            ERC223ReceivingContract _contract = ERC223ReceivingContract(_to);
            _contract.tokenFallback(msg.sender, _value, _data);
          }

        _balanceOf[_from] = _balanceOf[_from].sub(_value);
        _balanceOf[_to] = _balanceOf[_to].add(_value);

        Transfer(_from, _to, _value);

        return true;
    }
    return false;
}
```

Lines 56 and 81

```
function transferFrom(address _from, address _to, uint _value, bytes _data)
    public
    returns (bool) {
    if (_allowances[_from][msg.sender] > 0 &&
        _value > 0 &&
        _allowances[_from][msg.sender] >= _value &&
        _balanceOf[_from] >= _value) {

          _allowances[_from][msg.sender] -= _value;

          if (_to.isContract()) {
            ERC223ReceivingContract _contract = ERC223ReceivingContract(_to);
            _contract.tokenFallback(msg.sender, _value, _data);
          }

        _balanceOf[_from] = _balanceOf[_from].sub(_value);
        _balanceOf[_to] = _balanceOf[_to].add(_value);

        Transfer(_from, _to, _value);

        return true;
    }
    return false;
}
```

```
function transfer(address _to, uint _value, bytes _data)
    public
    returns (bool) {
    if (_value > 0 &&
        _value <= _balanceOf[msg.sender]) {

        if (_to.isContract()) {
          ERC223ReceivingContract _contract = ERC223ReceivingContract(_to);
          _contract.tokenFallback(msg.sender, _value, _data);
        }

        _balanceOf[msg.sender] = _balanceOf[msg.sender].sub(_value);
        _balanceOf[_to] = _balanceOf[_to].add(_value);

        return true;
    }
    return false;
}
```

SUPPORT INFORMATION

To implement the Check Effects Interactions pattern, we have to be aware about which parts of our function are the susceptible ones. Once we

identify that the external call with its insecurities regarding the control flow is the potential cause of vulnerability, we can act accordingly. A re-entrancy attack can lead to a function being called again, before its first invocation has been finished, therefore we should not make any changes to state variables, after interacting with external entities, as we cannot rely on the execution of any code coming after the interaction. This leaves us with the only option to update all state variables prior to the external interaction. This method can be described as “optimistic accounting”, because effects are written down as completed, before they actually took place. For example, the balance of a user will be reduced before the money is actually transferred to him. We get the natural ordering of: checks first, after that effects to state variables and interactions last.

```
// This code has not been professionally audited, therefore I cannot make any promises about
// safety or correctness. Use at own risk.
contract ChecksEffectsInteractions {

    mapping(address => uint) balances;

    function deposit() public payable {
        balances[msg.sender] = msg.value;
    }

    function withdraw(uint amount) public {
        require(balances[msg.sender] >= amount);

        balances[msg.sender] -= amount;

        msg.sender.transfer(amount);
    }
}
```

User balances are stored in a mapping in line 3. The deposit() function in line 5 lets the user deposit ether in the contract and stores the respective balances. The actual pattern is implemented in the withdraw() function in line 9, which is provided with the amount requested to withdraw. The first step is conducting all necessary checks. As this is a small example, there is only one condition to check: if the balance of the user is sufficient for the requested amount, which is done with the help of a require statement in line 10. The next step is the application of all effects, of which we again have only one: the adjustment of the users balance in line 12. All external interactions take place in the last step.

In an unsafe implementation of this contract, one disregarding the Checks Effects Interactions pattern, where the order of the effect and interaction in line 12 and 14 are exchanged and not transfer() but the unsafe and low level call.value() is used, a malicious contract could reenter our function. Because the control flow would pass over to the malicious contract, it would be able to call the withdraw() function again, before the first invocation is finished, without being intercepted by our check. This is because the line of code carrying the effect of adjusting the balance would not have been reached yet. Therefore, a second transfer of ether to the attacker would be issued. This circle would keep on draining the contract of ether, until either the transaction runs out of gas or the contracts funds are not sufficient anymore.

RECOMMENDATIONS

- Implement Optimistic Accounting where contract ensures a complete state before interacting with an untrusted contract.

SEVERITY

HIGH RATED RISK

CONCLUSION

The HERC contracts may be susceptible to Reentrancy and should implement Optimistic Accounting to ensure a complete state before interacting with an untrusted contract

Time	TRUSTED LEDGER SOLUTIONS identified areas where month is being used as time dependence.

FINDINGS

HERCDistribution.sol lines 115 and 118, month is used for time whereas day and year (365 days) are valid time increments.

```
115     allocations[_recipient] = Allocation(uint8(AllocationType.FOUNDER), startTime + 6 months, startTime + 3 years, _totalAllocated,
    0);
116   } else if (_supply == AllocationType.RESERVE) {
117     AVAILABLE_RESERVE_SUPPLY = AVAILABLE_RESERVE_SUPPLY.sub(_totalAllocated);
118     allocations[_recipient] = Allocation(uint8(AllocationType.RESERVE), startTime + 6 months, startTime + 18 months, _totalAllocated
```

SUPPORT INFORMATION

Time Units

Suffixes like `seconds`, `minutes`, `hours`, `days`, `weeks` and `years` after literal numbers can be used to convert between units of time where seconds are the base unit and units are considered naively in the following way:

- `1 == 1 seconds`
- `1 minutes == 60 seconds`
- `1 hours == 60 minutes`
- `1 days == 24 hours`
- `1 weeks == 7 days`
- `1 years == 365 days`

Take care if you perform calendar calculations using these units, because not every year equals 365 days and not even every day has 24 hours because of leap seconds. Due to the fact that leap seconds cannot be predicted, an

exact calendar library has to be updated by an external oracle.

Note

The suffix `years` has been deprecated due to the reasons above.

These suffixes cannot be applied to variables. If you want to interpret some input variable in e.g. days, you can do it in the following way:

```
function f(uint start, uint daysAfter) public {     if (now >= start + daysAfter * 1 days) {       // ...     } }
```

reference: http://solidity.readthedocs.io/en/v0.4.24/units-and-global-variables.html

RECOMMENDATIONS

Ensure that month is a valid time increment. Otherwise, change to 180 days.

SEVERITY

MODERATE RATED RISK

CONCLUSIONS

Use valid time increments

Insufficient Test cases	TRUSTED LEDGER SOLUTIONS identified insufficient test cases that do not test for edge cases or transactions involving ERC223 contracts.

FINDINGS

- **Test cases appear to be written to satisfy the "happy path".**
- **No tests for HercDistribution, AssetCoreFunctions or assetfees, For HercToken, scenarios when receiving address is a ERC223Contract.**
- **Tests for Hipr Game need expanding and comments on what the tests are testing for**
- **Herc.tge.master does not compile**

SUPPORT INFORMATION

Given When Then Structure

```
it("should transfer 16666667 HERC Tokens to the crowdsale balance", () => {
  var hercTokenInstance;
  return HERCToken.deployed().then(instance => {
    hercTokenInstance = instance;
    return hercTokenInstance.transfer(crowdsaleAddress, 16666667, {from: creatorAddress});
  }).then(result => {
    return hercTokenInstance.balanceOf.call(crowdsaleAddress);
  }).then(crowdsaleBalance => {
    assert.equal(crowdsaleBalance.valueOf(), 16666667, "16666667 wasn't in the crowdsale balance");
  });
});
```

This test case tests the "transfer" function. HERCToken is deployed and 16666667 is transferred from the creatorAddress to crowdsaleAddress. Then the balance of the crowdsaleAddress is taken and tested. This also assumes that HERCToken creator comes with 16666667 or more tokens - this assertion should be made before that the creator balance is greater.

Given:
HERCToken is deployed, Creator has 16666667 or more Herc Tokens,

When:
16666667 tokens are transferred from creatorAddress to crowdsaleAddress.

Then:

The 166666667 tokens should be at crowdsaleAddress

What happens when “given” creator has less than 16666667 tokens?

RECOMMENDATIONS

- Expand testing to test edge cases
- Generate tests for scenarios when receiving ERC223Contract address
  - HercDistribution
  - AssetCoreFunctions or assetfees
  - HercToken

SEVERITY

Transaction Ordering Dependence: HIGH RATED RISK

CONCLUSIONS

There are insufficient test cases to ensure the robustness of the HERC token

APPENDIX A: CONTRACTS ANALYZED

	Application	Checksum
1	HERCToken.sol	MD5 (Addresses.sol) = 38c237e9ca972848c91470a80e249ece
2	HERCDistribution.sol	MD5 (Crowdsale.sol) = 70ccdc96ac2d219a0dee9469667fccbf
3	Crowdsale.sol	MD5 (HERCDistribution.sol) = 274802e3e1305e73a84c3a10f571e94e
4	Addresses.sol	MD5 (HERCToken.sol) = 7ccf32e04b53dc506308e1ae7e2276ab
5	assetCoreFunctions.sol	MD5 (AssetCoreFunctions.sol) = c49683da736e7ba3b75a3ca245fcb7b3
6	assetFees.sol	MD5 (AssetFees.sol) = 813e37cc73e09b5a49c278e6e5974931
7	Playerscore.sol	MD5 (PlayerScore.sol) = 7efc86a6e5683f1bdcd603ad1af67b81

APPENDIX B: ADDITIONAL SECURITY RECOMMENDATIONS

See Recommendations for Smart Contract Security in Solidity - Ethereum Smart Contract.pdf attachement