June 1, 2018

Audit Report

CRYPTOFIGHTS ERC-721 TOKEN, PAYMENT AND REFERRAL CONTRACTS

AUTHOR: STEFAN BEYER – CRYPTRONICS.IO

Consutling Cryptronics Oü, Sepapaja 6, Tallinn 15551, Estonia, Registry code: 14481983
VAT number: EE102067569

Cryptronics

TABLE OF CONTENTS

Cryptronics

DISCLAIMER

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

Cryptronics

INTRODUCTION

PURPOSE OF THIS REPORT

The author of this report has been engaged to perform an audit of the ERC-721 token, payout and reward smart contracts for the CryptoFights game (http://cryptofights.io/).

The objectives of the audit are as follows:

1. Determine correct functioning of the contract, in accordance with the ERC-721 specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine contract bugs, which might lead to unexpected behavior.
4. Analyze, whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents the summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

CODEBASE SUBMITTED TO THE AUDIT

The smart contract code has been provided by the developers in form of access to the projects private source code repository:

https://github.com/CryptoLibertus/CryptoFights_Contracts

Commit number `f34a458ef025217a9b2d0cbe28670e7e40382cfc` was the latest version and has been analyzed in this audit.

METHODOLOGY

The audit has been performed in the following steps:

1. Gaining an understanding of the contract's intended purpose by reading the available documentation.
2. Automated scanning of the contract with static code analysis tools for security vulnerabilities and use of best practice guidelines.
3. Manual line by line analysis of the contracts source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    - Reentrancy analysis
    - Race condition analysis
    - Front-running issues and transaction order dependencies

- Time dependencies
- Under- / overflow issues
- Function visibility Issues
- Possible denial of service attacks
- Storage Layout Vulnerabilities

4. Report preparation

SMART CONTRACT OVERVIEW

HERO TOKEN

The submitted token contract is a non-fungible token following the ERC-721 specification standardized in EIP-721. The aim of the contract is to represent the game characters of the CryptoFights game.

The token contract extends the functionality of the ERC-721 standard by adding the following additional functionality:

- Permissioned access to some functionality (see below)
- Token contract ownership
- A token sales price
- A token sales facility

The token implementation uses the Open Zeppelin ERC-721 reference implementation.

PERMISSION SYSTEM

The contracts include *Managed* contract which implements varies modifier permissioned access to extending tokens' functionalities.

A register of permissioned addresses is maintained in a related *Management* contract.

The *Management* contract extends Open Zeppelin's *Pausable* lifecycle contract. This is used by the *Managed* contract to provide a modifier allowing certain functionalities of the extending functions to be also paused. This is used by the ERC-721 Hero token to pause creating new heroes.

PAYMENT AND REFERRAL SYSTEM

The payment system allows users to acquire a CryptoFight Hero token. This is closely linked to a referral system, which allows assigning a percentage of the sales to a referrer.

The combined payment and referral system consist in the following contracts:

- a Cashier contract which is called from the Hero Token to process purchases
- a Referral contract calculating payouts as a percentage of referred amounts

The payment system also makes use of the permission system by extending *Managed*.

REWARD SYSTEM

The reward systems is an ERC-721 token implementing a reward chest. Each individual reward consists in a certain amount of units of a particular ERC-20 token (representing items).

Again, Open Zeppelin's ERC-721 reference implementation is for the ERC-721 token.

Furthermore, the ERC-20 rewards are designed to use Open Zeppelin's ERC-20 implementation.

SUMMARY OF FINDINGS

The contracts provided for this audit are of exceptional quality.

Analysis with the aid of static code analysis tools has found no issues.

Community audited code has been reused whenever possible. A safe math library is used throughput the code to prevent overflow and underflow issues (one potential overflow has been identified, see low severity issues).

No reentrancy attack vectors have been found and precautions have been taken to avoid uninitialized storage pointers that may lead to overwriting storage.

For payouts, a pull pattern is used, and best practice guidelines have been followed throughout the code.

No attack vectors for Denial of Service attacks have been found and there are no potential block gas limit issues.

Gas usage is very reasonable for this type of contract.

AUDIT FINDING

REENTRANCY AND RACE CONDITIONS RESISTANCE

DESCRIPTION

Reentrancy vulnerabilities consist in unexpected behavior, if a function is called various times before execution has completed.

Let's look at the following function, which can be used to withdraw the total balance of the caller from a contract:

```
1. mapping(address => uint) private balances;
2.
3. function payOut() {

4.     require(msg.sender.call.value(balances[msg.sender])());
5.     balances[msg.sender] = 0;

6. }
```

The *call.value()* invocation causes contract external code to be executed. If the caller is another contract, this means that the contracts fallback method is executed. This may call *payOut()* again, before the balance is set to 0, thereby obtaining more funds than available.

AUDIT RESULT

No reentrancy issues have been found in the contract. The *transfer()* function is used for all ether transfers, imposing a gas limit and preventing recursive calls, and care has been taken in the order of calls.

UNDER-/OVERFLOW PROTECTION

DESCRIPTION

Balances are usually represented by unsigned integers, typically 256-bit numbers in Solidity. When unsigned integers overflow or underflow, their value changes dramatically. Let's look at the following example of a more common underflow (numbers shortened for readability):

```
  0x0003
- 0x0004
--------
  0xFFFF
```

It's easy to see the issue here. Subtracting 1 more than available balance causes an underflow. The resulting balance is now a large number.

Also note, that in integer arithmetic division is troublesome, due to rounding errors.

AUDIT RESULT

The CryptoFights contracts generally avoid overflow and underflow issues by employing the *SafeMath* library by OpenZeppelin for almost all arithmetic operations. In the few occasions were the library is not used, it is clear that there is no risk of overflows or underflows.

For example, the following code in *RewardChest.sol* is safe despite the potential underflow, because the function is private and only called from contract code that already checks that the reward removed exists. Therefore, *reward.length* will always be greater than 0 when this function is called:

```
146.       function removeReward(uint _id) private {
147.           uint lastRewardId = rewards.length - 1;
148.
149.           if (_id == lastRewardId) {
150.               delete rewards[_id];
151.           } else {
152.       // Keep storage array tightly packed
153.               rewards[_id] = rewards[lastRewardId];
154.
155.               delete rewards[lastRewardId];
156.           }
157.           rewards.length--;
158.       }
```

Omitting the use of safe math in this case is an acceptable gas optimization.

The exception of this is the multiplication in line 89 of *RewardChest.sol*, which introduces a slight chance of inconsistency due to a potential overflow (see low severity security issues).

TRANSACTION ORDERING ASSUMPTIONS

DESCRIPTION

Transactions enter a pool of unconfirmed transactions and maybe included in blocks by miners in any order, depending on the miner's transaction selection criteria, which is probably some algorithm aimed at achieving maximum earnings from transaction fees, but could be anything. Hence, the order of transactions being included can be completely different to the order in which they are generated. Therefore, contract code cannot make any assumptions on transaction order.

Apart from unexpected results in contract execution, there is a possible attack vector in this, as transactions are visible in the mempool and their execution can be predicted. This maybe an issue in trading, where delaying a transaction may be used for personal advantage by a rogue

miner. In fact, simply being aware of certain transactions before they are executed can be used as advantage by anyone, not just miners.

AUDIT RESULT

CryptoFight transactions are kept as simple as possible and care has been taken not to assume a specific order of invocation.

TIMESTAMP DEPENDENCIES

DESCRIPTION

Timestamps are generated by the miners. Therefore, no contract should rely on the block timestamp for critical operations, such as using it as a seed for random number generation. Consensys give a 30 seconds and a 12 minutes rules in their guidelines, which states that it is safe to use *block.timestamp,* if your time depending code can deal with a 30 second or 12 minute time variation, depending on the intended use.

AUDIT RESULT

There are no timestamp dependencies in the CryptoFights contract code.

DENIAL OF SERVICE ATTACK PREVENTION

DESCRIPTION

Denial of Service attacks can occur when a transaction depends on the outcome of an external call. A typical example of this some activity to be carried out after an Ether transfer. If the receiver is another contract, it can reject the transfer causing the whole transaction to fail.

AUDIT RESULT

CryptoFights avoids DoS attacks of this type, using a pull payment pattern and isolating Ether transfers into their own withdrawal transactions, such as the following function in *Cashier.sol*:

```
47. function claim() public {
48.     uint balance = payoutBalances[msg.sender];
49.
50.     require(balance > 0);
51.
52.     allocatedEther = allocatedEther.sub(balance);
53.     payoutBalances[msg.sender] = 0;
54.     msg.sender.transfer(balance);
55.
56.     emit Claim(msg.sender, balance);
57. }
```

BLOCK GAS LIMIT

DESCRIPTION

Contract transactions can sometimes be forced to always fails by making them exceed the maximum amount of gas that can be included in a block. The classic example can be found in this explanation. Forcing the contract to pay out many small amounts, will bump up the gas used and, if this exceeds the block gas limit, the whole transaction will fail.

The solution to this problem is avoiding situations in which many transaction calls can be caused by the same function invocation, especially if the number of calls can be influenced externally.

AUDIT RESULT

The CryptoFight contracts use pull payment patterns and, in general, avoid looping over variable-sized arrays.

The only instance of loop of this kind is the following code in *RewardChest.sol*:

```
50. for (uint i = 0; i < rewards.length; i++) {
51.     selectedReward = i;
52.
53.     if (accumulator + rewards[i].balance > rndReward) {
54.         break;
55.     }
56.
57.     accumulator = accumulator.add(rewards[i].balance);
58. }
```

Since, the length of the reward array is controlled by the CryptoFights software (probably the off-chain part) it is very unlikely for this array grows too large. Furthermore, the code has a very low gas usage, meaning it would need a very large array to cause the block gas limit to be exceeded.

STORAGE ALLOCATION PROTECTION

DESCRIPTION

Storage management in Solidity can be complicated. Declarations of structs inside the scope of a function default to storage pointers. It is therefore easy to end up with and uninitialized storage pointer, pointing to address 0, instead of declaring a new struct.

Writing to this pointer then causes storage to be overwritten unintentionally.

AUDIT RESULT

The CryptoFight contracts avoid storage allocation issues by declaring storage types explicitly. No issues related to his have been found during the audit.

COMMUNITY AUDITED CODE

DESCRIPTION

It always best to re-use community audited code when available, such as the code provided by Open Zeppelin.

AUDIT RESULT

The CryptoFight contracts are based on Open Zeppelin whenever possible. The ERC-721 tokens, ERC-20 tokens, facilities to pause contracts and the Ownable contract functionalities are re-used.

GAS USAGE ANALYSIS

DESCRIPTION

Gas usage of smart contracts is very important. Gas is charged for each operation that alters state, i.e. a write transaction. In contrast, read-only queries can be processed by local nodes and therefore do not have an associated cost.

Excessive gas usage may make contracts unusable in practice, in particular in times of network congestion when the gas price has to be increased to incentivize miners to prioritize transactions.

Furthermore, issues with excessive gas usage can lead to exceeding the block gas limit preventing transactions from completing. This is particularly dangerous in the case of executing code in unbounded loops, for example iterating over a variable size array. If the size of the array can be influenced by a public contract call, this can be used to create Denial of Service Attacks.

For these reasons, the present smart contract audit includes a gas usage analysis performed in two steps:

1. The code has been analyzed using automated gas estimation tools that return a relatively accurate estimate of the gas usage of each function.
2. As automated, gas estimation has its limits, a manual line by line analysis for gas related issues has also been performed.

AUDIT RESULT

AUTOMATED ANALYSIS

The following is the output of the automated gas usage analysis:

```
======= CFConstants.sol:CFConstants =======
Gas estimation:
construction:
   238 + 197800 = 198038
external:
   CAN_ALTER_REWARDS():     206
   CAN_ALTER_STATS():  250
   CAN_ALTER_XP():     316
   CAN_MINT_CHEST():   492
   CAN_RECORD_PURCHASE():   272
   CASHIER():    370
   DWARF():338
   ELF():  470
   HERO():  216
   HERO_PROMO(): 414
   HERO_VALIDATOR():   282
   HUMAN():448
   IS_TRUSTED_TOKEN(): 360
   REFERRAL():   392

======= Cashier.sol:Cashier =======
Gas estimation:
construction:
   82451 + 1243000 = 1325451
external:
   CAN_ALTER_REWARDS():     228
   CAN_ALTER_STATS():  316
   CAN_ALTER_XP():     448
   CAN_MINT_CHEST():   800
   CAN_RECORD_PURCHASE():   360
   CASHIER():    590
   DWARF():470
   ELF():  778
   HERO():  282
   HERO_PROMO(): 700
   HERO_VALIDATOR():   370
   HUMAN():734
   IS_TRUSTED_TOKEN(): 580
   REFERRAL():   678
   allocatedEther():   702
   claim():infinite
   etherHolder():      508
   management(): 882
   owner():904
   payoutBalances(address): 664
   recordPurchase(address,address):     infinite
   referredContracts(address):    1119
   referrers(address): 767
   renounceOwnership():     22425
   setEtherHolder(address): 21114
   setManagementContract(address):      20828
   totalReferred(address):  532
   transferOwnership(address):    23181
   withdrawBalance():  infinite

======= CryptoFightsHero.sol:CryptoFightsHero =======
Gas estimation:
construction:
   infinite + 2924400 = infinite
```

```
external:
   CAN_ALTER_REWARDS():     294
   CAN_ALTER_STATS():  426
   CAN_ALTER_XP():     558
   CAN_MINT_CHEST():   1064
   CAN_RECORD_PURCHASE():   470
   CASHIER():    722
   DWARF():580
   ELF():  1020
   HERO():  392
   HERO_PROMO(): 964
   HERO_VALIDATOR():   480
   HUMAN():998
   IS_TRUSTED_TOKEN(): 712
   REFERRAL():   920
   approve(address,uint256):23887
   balanceOf(address): 1065
   createHero(string,string,uint16[4],address):    infinite
   exists(uint256):    931
   getApproved(uint256):    611
   getHero(uint256):   2300
   heroes(uint256):    2813
   isApprovedForAll(address,address):   1568
   management(): 992
   name():  infinite
   owner():1014
   ownerOf(uint256):   1064
   price():1032
   renounceOwnership():     22557
   safeTransferFrom(address,address,uint256):infinite
   safeTransferFrom(address,address,uint256,bytes):     infinite
   setApprovalForAll(address,bool):     23227
   setManagementContract(address):      20894
   setPrice(uint256):  21075
   symbol():     infinite
   tokenByIndex(uint256):   1351
   tokenOfOwnerByIndex(address,uint256):     1456
   tokenURI(uint256):  infinite
   totalSupply():      476
   transferFrom(address,address,uint256):    infinite
   transferOwnership(address):    23445
   updateStats(uint256,uint16,uint16,uint16):infinite
   updateXP(uint256,uint64):infinite

======= HeroBase.sol:HeroBase =======
Gas estimation:
construction:
   190 + 142600 = 142790
external:
   getHero(uint256):   2168
   heroes(uint256):    2153

======= HeroValidator.sol:HeroValidator =======
Gas estimation:
construction:
   244966 + 475000 = 719966
external:
   CAN_ALTER_REWARDS():     228
   CAN_ALTER_STATS():  272
   CAN_ALTER_XP():     360
   CAN_MINT_CHEST():   558
   CAN_RECORD_PURCHASE():   316
   CASHIER():    414
   DWARF():382
```

```
   ELF():  536
   HERO(): 238
   HERO_PROMO(): 480
   HERO_VALIDATOR():   326
   HUMAN():514
   IS_TRUSTED_TOKEN(): 404
   REFERRAL():   458
   abilityScores(uint16):   1672
   calculateScoreWeights(uint16,uint16,uint16):    1813
   isHeroValid(uint16,uint16,uint16,uint16): infinite

======= Managed.sol:Managed =======
Gas estimation:

======= Management.sol:Management =======
Gas estimation:
construction:
   81722 + 519400 = 601122
external:
   contractRegistry(uint256):     673
   owner():530
   pause():21881
   paused():     514
   permissions(address,uint256):  821
   registerContract(uint256,address):   22403
   renounceOwnership():     22117
   setPermission(address,uint256,bool): 22753
   transferOwnership(address):    22741
   unpause():    21812

======= Migrations.sol:Migrations =======
Gas estimation:
construction:
   20462 + 152000 = 172462
external:
   last_completed_migration():    416
   owner():486
   setCompleted(uint256):   20544
   upgrade(address):   infinite

======= Referral.sol:Referral =======
Gas estimation:
construction:
   135 + 85600 = 85735
external:
   calculatePayoutAmount(uint256,uint256):   infinite

======= RewardChest.sol:RewardChest =======
Gas estimation:
construction:
   infinite + 2968800 = infinite
external:
   CAN_ALTER_REWARDS():     294
   CAN_ALTER_STATS():  426
   CAN_ALTER_XP():     580
   CAN_MINT_CHEST():   1064
   CAN_RECORD_PURCHASE():   492
   CASHIER():    766
   DWARF():602
   ELF():  1020
   HERO(): 392
   HERO_PROMO(): 964
   HERO_VALIDATOR():   502
   HUMAN():998
```

```
   IS_TRUSTED_TOKEN(): 734
   REFERRAL():   920
   addReward(address,uint256,uint256):  infinite
   addRewardTokens(uint256,address,uint256,uint256):    infinite
   approve(address,uint256): 23887
   balanceOf(address): 1065
   chestCounter():    548
   exists(uint256):   953
   getApproved(uint256):    611
   isApprovedForAll(address,address):   1568
   issue(address):    infinite
   management(): 1036
   name():  infinite
   open(uint256):     infinite
   owner(): 1058
   ownerOf(uint256):   1064
   renounceOwnership():     22557
   rewardCount():     944
   rewardLength():    1114
   rewards(uint256):   2232
   safeTransferFrom(address,address,uint256):infinite
   safeTransferFrom(address,address,uint256,bytes):     infinite
   setApprovalForAll(address,bool):    23227
   setManagementContract(address):     20894
   symbol():     infinite
   tokenByIndex(uint256):   1373
   tokenOfOwnerByIndex(address,uint256):     1456
   tokenURI(uint256):  infinite
   totalSupply():     476
   transferFrom(address,address,uint256):    infinite
   transferOwnership(address):    23489
   withdrawReward(uint256,address,uint128,uint256):     infinite
internal:
   removeReward(uint256):   infinite

======= TestToken.sol:TestToken =======
Gas estimation:
construction:
   62083 + 1187400 = 1249483
external:
   allowance(address,address):    948
   approve(address,uint256): 22353
   balanceOf(address): 691
   decreaseApproval(address,uint256):   infinite
   finishMinting():    22072
   increaseApproval(address,uint256):   infinite
   mint(address,uint256):   infinite
   mintingFinished():  492
   owner(): 640
   renounceOwnership():     22227
   totalSupply():     446
   transfer(address,uint256):     infinite
   transferFrom(address,address,uint256):    infinite
   transferOwnership(address):    22851

======= interfaces/ICashier.sol:ICashier =======
Gas estimation:

======= interfaces/IHeroValidator.sol:IHeroValidator =======
Gas estimation:

======= interfaces/IPromotion.sol:IPromotion =======
Gas estimation:
```

```
======= interfaces/IReferral.sol:IReferral =======
Gas estimation:

======= zeppelin-solidity/contracts/AddressUtils.sol:AddressUtils =======
Gas estimation:
construction:
   116 + 15200 = 15316
internal:
   isContract(address):     infinite

======= zeppelin-solidity/contracts/lifecycle/Pausable.sol:Pausable =======
Gas estimation:
construction:
   40896 + 295200 = 336096
external:
   owner():530
   pause():21881
   paused():     514
   renounceOwnership():     22117
   transferOwnership(address):    22675
   unpause():    21812

======= zeppelin-solidity/contracts/math/SafeMath.sol:SafeMath =======
Gas estimation:
construction:
   116 + 15200 = 15316
internal:
   add(uint256,uint256):    infinite
   div(uint256,uint256):    infinite
   mul(uint256,uint256):    infinite
   sub(uint256,uint256):    infinite

======= zeppelin-solidity/contracts/ownership/Ownable.sol:Ownable =======
Gas estimation:
construction:
   20498 + 189800 = 210298
external:
   owner():464
   renounceOwnership():     22073
   transferOwnership(address):    22609

======= zeppelin-solidity/contracts/token/ERC20/BasicToken.sol:BasicToken =======
Gas estimation:
construction:
   251 + 209000 = 209251
external:
   balanceOf(address): 581
   totalSupply():     402
   transfer(address,uint256):     infinite

======= zeppelin-solidity/contracts/token/ERC20/ERC20.sol:ERC20 =======
Gas estimation:

======= zeppelin-solidity/contracts/token/ERC20/ERC20Basic.sol:ERC20Basic =======
Gas estimation:

======= zeppelin-solidity/contracts/token/ERC20/MintableToken.sol:MintableToken
=======
Gas estimation:
construction:
   41793 + 1187400 = 1229193
external:
   allowance(address,address):    948
   approve(address,uint256):22353
```

```
   balanceOf(address): 691
   decreaseApproval(address,uint256):   infinite
   finishMinting():    22072
   increaseApproval(address,uint256):   infinite
   mint(address,uint256):    infinite
   mintingFinished():  492
   owner():640
   renounceOwnership():     22227
   totalSupply():      446
   transfer(address,uint256):     infinite
   transferFrom(address,address,uint256):    infinite
   transferOwnership(address):    22851

======= zeppelin-solidity/contracts/token/ERC20/StandardToken.sol:StandardToken
=======
Gas estimation:
construction:
   864 + 830200 = 831064
external:
   allowance(address,address):    838
   approve(address,uint256):22331
   balanceOf(address): 647
   decreaseApproval(address,uint256):   infinite
   increaseApproval(address,uint256):   infinite
   totalSupply():      424
   transfer(address,uint256):     infinite
   transferFrom(address,address,uint256):    infinite

======= zeppelin-solidity/contracts/token/ERC721/ERC721.sol:ERC721 =======
Gas estimation:

======= zeppelin-solidity/contracts/token/ERC721/ERC721.sol:ERC721Enumerable =======
Gas estimation:

======= zeppelin-solidity/contracts/token/ERC721/ERC721.sol:ERC721Metadata =======
Gas estimation:

======= zeppelin-solidity/contracts/token/ERC721/ERC721Basic.sol:ERC721Basic =======
Gas estimation:

======= zeppelin-
solidity/contracts/token/ERC721/ERC721BasicToken.sol:ERC721BasicToken =======
Gas estimation:
construction:
   981 + 946400 = 947381
external:
   approve(address,uint256):23865
   balanceOf(address): 735
   exists(uint256):    711
   getApproved(uint256):    589
   isApprovedForAll(address,address):   930
   ownerOf(uint256):   756
   safeTransferFrom(address,address,uint256):infinite
   safeTransferFrom(address,address,uint256,bytes):      infinite
   setApprovalForAll(address,bool):     22721
   transferFrom(address,address,uint256):    infinite
internal:
   _burn(address,uint256):  infinite
   _mint(address,uint256):  infinite
   addTokenTo(address,uint256):   infinite
   checkAndCallSafeTransfer(address,address,uint256,bytes memory):  infinite
   clearApproval(address,uint256):      infinite
   isApprovedOrOwner(address,uint256):  1364
   removeTokenFrom(address,uint256):    infinite
```

```
======= zeppelin-solidity/contracts/token/ERC721/ERC721Receiver.sol:ERC721Receiver
=======
Gas estimation:

======= zeppelin-solidity/contracts/token/ERC721/ERC721Token.sol:ERC721Token =======
Gas estimation:
construction:
   infinite + 1400400 = infinite
external:
   approve(address,uint256): 23887
   balanceOf(address): 823
   exists(uint256):    777
   getApproved(uint256):    611
   isApprovedForAll(address,address):   1062
   name(): infinite
   ownerOf(uint256):   844
   safeTransferFrom(address,address,uint256): infinite
   safeTransferFrom(address,address,uint256,bytes):     infinite
   setApprovalForAll(address,bool):    22831
   symbol():     infinite
   tokenByIndex(uint256):   1197
   tokenOfOwnerByIndex(address,uint256):     1390
   tokenURI(uint256):  infinite
   totalSupply():     476
   transferFrom(address,address,uint256):    infinite
internal:
   _burn(address,uint256):  infinite
   _mint(address,uint256):  infinite
   _setTokenURI(uint256,string memory): infinite
   addTokenTo(address,uint256):   infinite

   removeTokenFrom(address,uint256): infinite
```

As can be seen, gas usage of all functions for which a numerical result was return is very reasonable.

Infinite gas estimates are due to the limitations of automated gas analysis. These functions have been analyzed manually.

MANUAL ANALYSIS

It is obvious that care has been taken to implement all functions of the CryptoFights contracts as compact and gas efficiently as possible.

Gas usage is very reasonable.

SECURITY ISSUES

HIGH SEVERITY ISSUES

No high severity issues have been found.

MEDIUM SEVERITY ISSUES

No medium severity issues have been found.

LOW SEVERITY ISSUES

POTENTIAL OVERFLOW

The following code in *RewardChest.sol* exposes a small risk of introducing an inconsistency due to arithmetic overflow in line 89:

```
82. function addReward(
83. ERC20 _token,
84. uint _rewardAmount,
85. uint _rewardCount
86. ) public requirePermission(CAN_ALTER_REWARDS) {
87.     require(hasPermission(address(_token), IS_TRUSTED_TOKEN), "NOT_TRUSTED");
88.
89.     _token.transferFrom(msg.sender, address(this), uint(_rewardAmount * _rewar
    dCount));
90.
91.     rewardCount = rewardCount.add(_rewardCount);
92.
93.     uint rewardId = rewards.push(Reward(_token, _rewardAmount, _rewardCount))
    - 1;
94.
95.     emit RewardAdded(rewardId, _token, _rewardAmount, _rewardCount);
96. }
```

Should the values of _rewardAmount and _rewardCount be very high due to an off-chain code error and overflow could occur, leading to an incorrect number of tokens to be transferred.

This is classed as low severity, as the function requires special permissions to be executed. However, an unintentional error in the calling code could cause this issue.

Recommendation: use the already included safe math library for this calculation.

RANDOM NUMBER GENERATION

The following code in *RewardChest.sol* is used provide a random number:

```
46.
uint rndReward = uint(keccak256(_tokenId, blockhash(block.number - 1))) % rewardCount
;
```

This code could be called from another contract. Since all parameters in this calculation are easily obtainable, the calling contract could execute the same calculation in the same transaction before calling the function. The calculation could be repeated several times and the transaction reverted until the randomly chosen reward is of advantage to the user.

However, this is issue is of very low severity because the reward of such an attack would be very minor and probably not result in an advantage.

The team may decide that random number generation in this way is acceptable in this particular case, as alternative solutions would be costly.

ADDITIONAL RECOMMENDATIONS

DEPRECATED USE OF KECCAK256

The latest version of the Solidity complier (version 0.4.24) warns about the usage of the function keccak256 without a single bytes argument. In future versions this will throw an error. The following line in *RewardChest.sol* causes this warning:

```
47.
uint rndReward = uint(keccak256(_tokenId, blockhash(block.number - 1))) % rewardCount
;
```

It is recommended to make this code future proof.