
Armors Labs
Scape Forum
Smart Contract Audit

Scape Forum Audit Summary

Project name : Scape Forum Contract

Project address: None

Code URL : https://github.com/blocklords/seascape-smartcontracts/blob/nft-burning/contracts/game_4/NftBurning.sol

Commit : 067a01f8b63257ffcf49119ae9782c5a10f69224

Project target : Scape Forum Contract Audit

Blockchain : Binance Smart Chain（BSC）

Test result : PASSED

Audit Info

Audit NO : 0X202106230019

Audit Team : Armors Labs

Audit Proofreading: https://armors.io/#project-cases

Scape Forum Audit

The Scape Forum team asked us to review and audit their Scape Forum contract. We looked at the code and now publish our results.

Here is our assessment and recommendations, in order of importance.

Document information

Name	Auditor	Version	Date
Scape Forum Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2021-06-23

Audit results

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the Scape Forum contract. The above should not be construed as investment advice.

Based on the widely recognized security status of the current underlying blockchain and smart contract, this audit report is valid for 3 months from the date of output.

(Statement: Armors Labs reports only on facts that have occurred or existed before this report is issued and assumes corresponding responsibilities. Armors Labs is not able to determine the security of its smart contracts and is not responsible for any subsequent or existing facts after this report is issued. The security audit analysis and other content of this report are only based on the documents and information provided by the information provider to Armors Labs at the time of issuance of this report (" information provided " for short). Armors Labs postulates that the information provided is not missing, tampered, deleted or hidden. If the information provided is missing, tampered,

deleted, hidden or reflected in a way that is not consistent with the actual situation, Armors Labs shall not be responsible for the losses and adverse effects caused.)

Audited target file

file	md5
NftBurning.sol	709924718568a4cf36ecb4f1f157478e

Vulnerability analysis

Vulnerability distribution

vulnerability level	number
Critical severity	0
High severity	0
Medium severity	0
Low severity	0

Summary of audit results

Vulnerability	status
Re-Entrancy	safe
Arithmetic Over/Under Flows	safe
Unexpected Blockchain Currency	safe
Delegatecall	safe
Default Visibilities	safe
Entropy Illusion	safe
External Contract Referencing	safe
Short Address/Parameter Attack	safe
Unchecked CALL Return Values	safe
Race Conditions / Front Running	safe
Denial Of Service (DOS)	safe
Block Timestamp Manipulation	safe
Constructors with Care	safe
Unintialised Storage Pointers	safe
Floating Points and Numerical Precision	safe
tx.origin Authentication	safe

Vulnerability	status
Permission restrictions	safe

Contract file

NftBurning.sol

```
pragma solidity 0.6.7;

//declare imports
import "./../openzeppelin/contracts/token/ERC20/IERC20.sol";
import "./../openzeppelin/contracts/token/ERC20/SafeERC20.sol";
import "./../openzeppelin/contracts/access/Ownable.sol";
import "./../openzeppelin/contracts/math/SafeMath.sol";
import "./../openzeppelin/contracts/utils/Counters.sol";
import "./../openzeppelin/contracts/token/ERC721/IERC721Receiver.sol";
import "./../seascape_nft/NftFactory.sol";
import "./../seascape_nft/SeascapeNft.sol";
import "./Crowns.sol";


/// @title Nft Burning contract  mints a higher quality nft in exchange for
/// five lower quality nfts + CWS fee
/// @author Nejc Schneider
contract NftBurning is Crowns, Ownable, IERC721Receiver{
    using SafeMath for uint256;
    using Counters for Counters.Counter;

    NftFactory nftFactory;
    SeascapeNft private nft;
    Counters.Counter private sessionId;

    /// @notice holds session related data. Since event is a solidity keyword, we call them session i
    struct Session {
        uint256 period;       // session duration
        uint256 startTime;    // session start in unixtimestamp
        uint256 generation;       // Seascape Nft generation
        uint256 interval;      // duration between every minting
        uint256 fee;          // amount of CWS token to spend to mint a new nft
        uint256 minStake;     // minimum amount of crowns deposit, only for staking
        uint256 maxStake;     // maximum amount of crowns deposit, only for staking
    }
    /// @notice keeps track of balances for each user
    struct Balance {
          uint256 totalStaked;    // amount of crowns staked
        uint256 mintedTime;     // track minted time per address
    }

    /// @notice Tracking player balance within a game session.
    /// @dev session id =>(wallet address => (Balance struct))
    mapping(uint256 => mapping(address => Balance)) public balances;
    /// @notice each session is a seperate object
    /// @dev session id =>(Session struct)
    mapping(uint256 => Session) public sessions;
    /// session related data
    uint256 public lastSessionId;

    event Minted(
        uint256 indexed sessionId,
        address indexed owner,
        uint256[5] burntNfts,
        uint256 mintedTime,
```

```
        uint256 imgId,
        uint256 mintedNft
    );
    event SessionStarted(
        uint256 indexed sessionId,
        uint256 generation,
        uint256 fee,
        uint256 interval,
        uint256 start_time,
        uint256 end_time,
        uint256 minStake,
        uint256 maxStake
    );
    event Staked(
        uint256 indexed sessionId,
        address indexed owner,
        uint256 amount,
        uint256 totalStaked
    );
    event Withdrawn(
        address indexed owner,
        uint256 indexed sessionId,
        uint256 withdrawnAmount,
        uint256 withdrawnTime
    );
    event FactorySet(address indexed factoryAddress);


    /// @dev set currency addresses
    /// @param _crowns staking currency address
    /// @param _nftFactory nft minting contract address
    /// @param _nft nft fusion contract address
    constructor(address _crowns, address _nftFactory, address _nft)  public {
        require(_nftFactory != address(0), "nftFactory cant be zero address");

        /// @dev set crowns is defined in Crowns.sol
        setCrowns(_crowns);

        sessionId.increment();     // starts at value 1
        nftFactory = NftFactory(_nftFactory);
        nft = SeascapeNft(_nft);
    }

    /// @dev start a new session, during which players are allowed to mint nfts
    /// @param _startTime unix timestamp when session starts
    /// @param _period unix timestamp when session ends. Should be equal to startTime + period
    /// @param _generation generation of newly minted nfts
    /// @param _interval duration between every possible minting
    /// @param _fee amount of CWS token to spend to mint a new nft
    function startSession(
        uint256 _startTime,
        uint256 _period,
        uint256 _generation,
        uint256 _interval,
        uint256 _fee,
        uint256 _minStake,
        uint256 _maxStake
    )
        external
        onlyOwner
    {
        /// cant start new session when another is active
        if (lastSessionId > 0) {
            require(!isActive(lastSessionId), "another session is still active");
        }
        require(_startTime > block.timestamp, "session should start in future");
```

```
        require(_period > 0, "period should be above 0");
        require(_interval > 0 && _interval <= _period,
        "interval should be >0 & <period");
        require(_fee > 0, "fee should be above 0");
        require(_minStake > 0, "minStake should be above 0");
        require(_maxStake > _minStake, "maxStake should be > minStake");

            //--------------------------------------------------------------------
            // updating session related data
            //--------------------------------------------------------------------

        uint256 _sessionId = sessionId.current();
        sessions[_sessionId] = Session(
            _period,
            _startTime+ _period,
            _generation,
            _interval,
            _fee,
            _minStake,
            _maxStake
        );

        sessionId.increment();
        lastSessionId = _sessionId;

        emit SessionStarted(
            _sessionId,
            _generation,
            _fee,
            _interval,
            _startTime,
            _startTime + _period,
            _minStake,
            _maxStake
        );
    }

    /// @notice spend nfts and cws, burn nfts, mint a higher quality nft and send it to player
    /// @param _sessionId id of the active session, during which nfts can be minted
    /// @param _nfts users nfts which will be burned
    /// @param _quality  of the new minting nft
    /// @param _v part of signature of message
    /// @param _r part of signature of message
    /// @param _s part of signature of message
    function mint(
        uint256 _sessionId,
        uint256[5] calldata _nfts,
        uint8 _quality,
        uint256 _imgId,
        uint8 _v,
        bytes32 _r,
        bytes32 _s
    )
        external
    {
        Session storage _session = sessions[_sessionId];
        Balance storage _balance = balances[_sessionId][msg.sender];

        require(_sessionId > 0, "Session has not started yet");
        require(_nfts.length == 5, "Need to deposit 5 nfts");
        require(_quality >= 1 && _quality <= 5, "Quality value should range 1 - 5");
        require(isActive(_sessionId), "Session not active");
        require(_balance.mintedTime == 0 ||
            (_balance.mintedTime.add(_session.interval) < block.timestamp),
            "Still in cooldown, try later");
        require(crowns.balanceOf(msg.sender) >= _session.fee, "Not enough CWS in your wallet");
```

```
        //---------------------------------------------------------------------
        // spend crowns, burn nfts, mint new nft
        //---------------------------------------------------------------------

        /// @dev make sure that signature of nft matches with the address of the contract deployer
        bytes32 _messageNoPrefix = keccak256(abi.encodePacked(
            _nfts[0],
            _nfts[1],
            _nfts[2],
            _nfts[3],
            _nfts[4],
            _balance.totalStaked,
            _imgId,
            _quality
        ));
        bytes32 _message = keccak256(abi.encodePacked(
            "\x19Ethereum Signed Message:\n32", _messageNoPrefix));
        address _recover = ecrecover(_message, _v, _r, _s);
        require(_recover == owner(),  "Verification failed");

        /// @dev verify nfts ids and ownership
        for (uint _index=0; _index < 5; _index++) {
            require(_nfts[_index] > 0, "Nft id must be greater than 0");
            require(nft.ownerOf(_nfts[_index]) == msg.sender, "Nft is not owned by caller");
        }
        /// @dev spend crowns
        crowns.spendFrom(msg.sender, _session.fee);
        /// @dev burn nfts
        for (uint _index=0; _index < 5; _index++) {
            nft.burn(_nfts[_index]);
        }
        /// @dev mint new nft
        uint256 mintedNftId = nftFactory.mintQuality(msg.sender, _session.generation, _quality);
        require(mintedNftId > 0, "Failed to mint a token");
        _balance.mintedTime = block.timestamp;
        emit Minted(_sessionId, msg.sender, _nfts, _balance.mintedTime, _imgId, mintedNftId);
    }

    /// @notice stake crowns
    /// @param _sessionId id of active session
    /// @param _amount amount of cws to stake
    function stake(uint256 _sessionId, uint256 _amount) external {
        Session storage _session = sessions[_sessionId];
        Balance storage _balance = balances[_sessionId][msg.sender];

        require(_sessionId > 0, "No active session");
        require(isActive(_sessionId), "Session not active");
        require(_amount > 0, "Should stake more than 0");
        require(_balance.totalStaked.add(_amount) <= _session.maxStake,
            "Cant stake more than maxStake");
        require(_balance.totalStaked.add(_amount) >= _session.minStake,
            "Cant stake less than minStake");
        require(crowns.balanceOf(msg.sender) >= _amount, "Not enough CWS in your wallet");
        crowns.transferFrom(msg.sender, address(this), _amount);

        /// @dev update balance
        balances[_sessionId][msg.sender].totalStaked = balances[_sessionId][msg.sender]
            .totalStaked.add(_amount);

        emit Staked(_sessionId, msg.sender, _amount,  _balance.totalStaked);
    }

    /// @notice withdraw callers totalStaked crowns
    /// @param _sessionId id of past session
    function withdraw(uint256 _sessionId) external {
```

```
        require(!isActive(_sessionId), "Session should be inactive");
        require(balances[_sessionId][msg.sender].totalStaked > 0, "Total staked amount is 0");

        /// update balance first to avoid reentrancy
        uint256 withdrawnAmount = balances[_sessionId][msg.sender].totalStaked;
        delete balances[_sessionId][msg.sender].totalStaked;

        /// transfer crowns second
        crowns.transfer(msg.sender, withdrawnAmount);

        emit Withdrawn(msg.sender, _sessionId, withdrawnAmount, block.timestamp);
    }

    /// @notice return amount of coins staked by _owner
    /// @param _sessionId id of active or past session
    /// @param _owner owner of staked coins
    /// @return token amount
    function totalStakedBalanceOf(
        uint256 _sessionId,
        address _owner
    )
        external
        view
        returns(uint256)
    {
        return balances[_sessionId][_owner].totalStaked;
    }

    /// @dev sets a smartcontract that mints tokens.
    /// @dev the nft factory should give a permission on it's own side to this contract too.
    /// @param _address nftFactory's new address
    function setNftFactory(address _address) external onlyOwner {
        require(_address != address(0), "nftFactory address cant be zero");
        nftFactory = NftFactory(_address);

        emit FactorySet(_address);
    }

    /// @notice check whether session is active or not
    /// @param _sessionId id of session to verify
    /// @return true if session is active
    function isActive(uint256 _sessionId) internal view returns(bool) {
        if (now > sessions[_sessionId].startTime + sessions[_sessionId].period) {
            return false;
          }
        return true;
    }

    /// @dev encrypt token data
    /// @return encrypted data
    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    )
        external
        override
        returns (bytes4)
    {
        return bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"));
    }

}
```

Analysis of audit results

Re-Entrancy

- **Description:**
  One of the features of smart contracts is the ability to call and utilise code of other external contracts. Contracts also typically handle Blockchain Currency, and as such often send Blockchain Currency to various external user addresses. The operation of calling external contracts, or sending Blockchain Currency to an address, requires the contract to submit an external call. These external calls can be hijacked by attackers whereby they force the contract to execute further code (i.e. through a fallback function) , including calls back into itself. Thus the code execution "re-enters" the contract. Attacks of this kind were used in the infamous DAO hack.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Arithmetic Over/Under Flows

- **Description:**
  The Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable, only has a certain range of numbers it can represent. A uint8 for example, can only store numbers in the range [0,255]. Trying to store 256 into a uint8 will result in 0. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unexpected Blockchain Currency

- **Description:**
  Typically when Blockchain Currency is sent to a contract, it must execute either the fallback function, or another function described in the contract. There are two exceptions to this, where Blockchain Currency can exist in a contract without having executed any code. Contracts which rely on code execution for every Blockchain Currency sent to the contract can be vulnerable to attacks where Blockchain Currency is forcibly sent to a contract.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Delegatecall

- **Description:**
  The CALL and DELEGATECALL opcodes are useful in allowing developers to modularise their code. Standard external message calls to contracts are handled by the CALL opcode whereby code is run in the context of the external contract/function. The DELEGATECALL opcode is identical to the standard message call, except that the code executed at the targeted address is run in the context of the calling contract along with the fact that msg.sender and msg.value remain unchanged. This feature enables the implementation of libraries whereby developers can create reusable code for future contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Default Visibilities

- **Description:**
  Functions in Solidity have visibility specifiers which dictate how functions are allowed to be called. The visibility determines whBlockchain Currency a function can be called externally by users, by other derived contracts, only internally or only externally. There are four visibility specifiers, which are described in detail in the Solidity Docs. Functions default to public allowing users to call them externally. Incorrect use of visibility specifiers can lead to some devestating vulernabilities in smart contracts as will be discussed in this section.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Entropy Illusion

- **Description:**
  All transactions on the blockchain are deterministic state transition operations. Meaning that every transaction modifies the global state of the ecosystem and it does so in a calculable way with no uncertainty. This ultimately means that inside the blockchain ecosystem there is no source of entropy or randomness. There is no rand() function in Solidity. Achieving decentralised entropy (randomness) is a well established problem and many ideas have been proposed to address this (see for example, RandDAO or using a chain of Hashes as described by Vitalik in this post).
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

External Contract Referencing

- **Description:**
  One of the benefits of the global computer is the ability to re-use code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts and in general

operation use external message calls to interact with these contracts. These external message calls can mask malicious actors intentions in some non-obvious ways, which we will discuss.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unsolved TODO comments

- **Description:**
  Check for Unsolved TODO comments
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Short Address/Parameter Attack

- **Description:**
  This attack is not specifically performed on Solidity contracts themselves but on third party applications that may interact with them. I add this attack for completeness and to be aware of how parameters can be manipulated in contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unchecked CALL Return Values

- **Description:**
  There a number of ways of performing external calls in solidity. Sending Blockchain Currency to external accounts is commonly performed via the transfer() method. However, the send() function can also be used and, for more versatile external calls, the CALL opcode can be directly employed in solidity. The call() and send() functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialised by call() or send()) fails, rather the call() or send() will simply return false. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Race Conditions / Front Running

- **Description:**
  The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users race code execution to obtain unexpected states. Re-Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the blockchain. There is a variety of good posts on this subject, a few are: Wiki - Safety, DASP - Front-Running and the Consensus - Smart Contract Best Practices.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Denial Of Service (DOS)

- **Description:**
  This category is very broad, but fundamentally consists of attacks where users can leave the contract inoperable for a small period of time, or in some cases, permanently. This can trap Blockchain Currency in these contracts forever, as was the case with the Second Parity MultiSig hack
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Block Timestamp Manipulation

- **Description:**
  Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion section for further details), locking funds for periods of time and various state-changing conditional statements that are time-dependent. Miner's have the ability to adjust timestamps slightly which can prove to be quite dangerous if block timestamps are used incorrectly in smart contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Constructors with Care

- **Description:**
  Constructors are special functions which often perform critical, privileged tasks when initialising contracts. Before solidity v0.4.22 constructors were defined as functions that had the same name as the contract that contained them. Thus, when a contract name gets changed in development, if the constructor name isn't changed, it becomes a normal, callable function. As you can imagine, this can (and has) lead to some interesting contract hacks.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unintialised Storage Pointers

- **Description:**
  The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default types for local variables of functions is highly recommended when developing contracts. This is because it is possible to produce vulnerable contracts by inappropriately intialising variables.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Floating Points and Numerical Precision

- **Description:**
  As of this writing (Solidity v0.4.24), fixed point or floating point numbers are not supported. This means that floating point representations must be made with the integer types in Solidity. This can lead to errors/vulnerabilities if not implemented correctly.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

tx.origin Authentication

- **Description:**
  Solidity has a global variable, tx.origin which traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts leaves the contract vulnerable to a phishing-like attack.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Permission restrictions

- **Description:**
  Contract managers who can control liquidity or pledge pools, etc., or impose unreasonable restrictions on other users.

- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

armors.io

contact@armors.io