



# SMART CONTRACT AUDIT REPORT

for

## BORINGDAO



Prepared By: Shuxiao Wang

Hangzhou, China  
Dec. 28, 2020

## Document Properties

Client	BoringDAO
Title	Smart Contract Audit Report
Target	BoringDAO
Version	1.0
Author	Xudong Shao
Auditors	Xudong Shao, Chiachih Wu, Huaguo Shi
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

## Version Info

Version	Date	Author(s)	Description
1.0	Dec. 28, 2020	Xudong Shao	Final Release

## Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

## Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	About BoringDAO	5
1.2	About PeckShield	6
1.3	Methodology	6
1.4	Disclaimer	8
<b>2</b>	<b>Findings</b>	<b>10</b>
2.1	Summary	10
2.2	Key Findings	11
<b>3</b>	<b>Detailed Results</b>	<b>12</b>
3.1	Business Logic Error in <code>unpauseSatellitePool()</code>	12
3.2	Missed Sanity Checks in <code>Liquidation:pause()</code>	13
3.3	Unsafe Ownership Transition in <code>SatellitePool</code>	14
3.4	Unused Interfaces	15
3.5	Weak Randomness in <code>Tunnel::burn()</code>	16
<b>4</b>	<b>Conclusion</b>	<b>18</b>
<b>5</b>	<b>Appendix</b>	<b>19</b>
5.1	Basic Coding Bugs	19
5.1.1	Constructor Mismatch	19
5.1.2	Ownership Takeover	19
5.1.3	Redundant Fallback Function	19
5.1.4	Overflows & Underflows	19
5.1.5	Reentrancy	20
5.1.6	Money-Giving Bug	20
5.1.7	Blackhole	20
5.1.8	Unauthorized Self-Destruct	20
5.1.9	Revert DoS	20

---

5.1.10	Unchecked External Call	21
5.1.11	Gasless Send	21
5.1.12	Send Instead Of Transfer	21
5.1.13	Costly Loop	21
5.1.14	(Unsafe) Use Of Untrusted Libraries	21
5.1.15	(Unsafe) Use Of Predictable Variables	22
5.1.16	Transaction Ordering Dependence	22
5.1.17	Deprecated Uses	22
5.2	Semantic Consistency Checks	22
5.3	Additional Recommendations	22
5.3.1	Avoid Use of Variadic Byte Array	22
5.3.2	Make Visibility Level Explicit	23
5.3.3	Make Type Inference Explicit	23
5.3.4	Adhere To Function Declaration Strictly	23
References		24



# 1 | Introduction

Given the opportunity to review the **BoringDAO** design document and related smart contract source code, we outline in this report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

## 1.1 About BoringDAO

BoringDAO is a decentralized bridge that connects multiple blockchains, and it offers users a way to transfer crypto tokens across different blockchains. Therefore, BoringDAO could maximize the utilization rate of various crypto assets, such as BTC, XRP, BCH, etc, and bring these tokens to the DeFi applications on Ethereum.

The basic information of the BoringDAO is as follows:

Table 1.1: Basic Information of BoringDAO

Item	Description
Issuer	BoringDAO
Website	<a href="https://boringdao.com/">https://boringdao.com/</a>
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Dec. 28, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- <https://github.com/BoringDAO/boringDAO-contract> (8b381c3)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- <https://github.com/BoringDAO/boringDAO-contract> (f806935)

## 1.2 About PeckShield

PeckShield Inc. [12] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (<https://t.me/peckshield>), Twitter (<http://twitter.com/peckshield>), or Email ([contact@peckshield.com](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [7]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [6], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

## 1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.






Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
<b>Error Conditions, Return Values, Status Codes</b>	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logic</b>	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

## 2 | Findings

### 2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the BoringDAO protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	
Low	1	
Informational	3	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in [Section 3](#).

## 2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 1 low-severity vulnerability, and 3 informational recommendations.

Table 2.1: Key Audit Findings of BoringDAO

ID	Severity	Title	Category	Status
PVE-001	Medium	Business Logic Error in <code>unpauseSatellitePool()</code>	Business Logic	Fixed
PVE-002	Info.	Missed Sanity Checks in <code>Liquidation:pause()</code>	Business Logic	Fixed
PVE-003	Info.	Unsafe Ownership Transition in <code>SatellitePool</code>	Business Logic	Fixed
PVE-004	Info.	Unused Interfaces	Coding Practices	Fixed
PVE-005	Low	Weak Randomness in <code>Tunnel::burn()</code>	Business Logic	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

## 3 | Detailed Results

### 3.1 Business Logic Error in `unpauseSatellitePool()`

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: liquidation
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

#### Description

As a contingency plan, the dev team and trustees could pause the BoringDAO system when there is an emergency through the `pause()` function in the `Liquidation` contract. Typically, pausing the whole system comes with paused Satellite pools. If there are more than  $\frac{2}{3}$  trustees reach an agreement with each others, the `unpauseSatellitePool()` could be used to unpause a paused Satellite pool. Specifically, as shown in the following code snippet, line 101 increments the `unpausePoolConfirmCount` [pool] whenever a trustee invokes `unpauseSatellitePool()`. Later on, in line 104, the pool is paused when the `unpausePoolConfirmCount`[pool] reaches the threshold.

```

95     function unpauseSatellitePool(address pool) public onlyTrustee {
96         require(systemPause == true, "Liquidation::unpauseSatellitePool:systemPause
           should paused when call unpause()");
97         require(isSatellitePool[pool] == true, "Liquidation::unpauseSatellitePool:Not
           SatellitePool");
98         if(unpauseConfirm[msg.sender][pool] == false) {
99             unpauseConfirm[msg.sender][pool] = true;
100         }
101         unpausePoolConfirmCount[pool] = unpausePoolConfirmCount[pool].add(1);
102         uint trusteeCount = IHasRole(addressReso.requireAndKey2Address(BORING_DAO, "
           Liquidation::withdraw:boringDAO contract not exist")).getRoleMemberCount(
           TRUSTEE_ROLE);
103         uint threshold = trusteeCount.mod(3) == 0 ? trusteeCount.mul(2).div(3) :
           trusteeCount.mul(2).div(3).add(1);
104         if (unpausePoolConfirmCount[pool] >= threshold) {
105             IPause(pool).unpause();

```

```

106     }
107 }

```

Listing 3.1: liquidation.sol

However, the current implementation fails to check if the trustee has called `unpauseSatellitePool()` with the specific `pool` already. Since line 101 increments the count without checking `unpauseConfirm[msg.sender][pool]`, a malicious trustee could call `unpauseSatellitePool()` multiple times to unpause any `pool`. In addition, line 99 has a typo (i.e., a duplicate `=`) such that `unpauseConfirm[msg.sender][pool]` would never be set to `true`.

**Recommendation** Increment `unpausePoolConfirmCount[pool]` only if `unpauseConfirm[msg.sender][pool]` is `false`. In addition, fix the typo in line 99.

**Status** This issue has been fixed in the commit: [8f3dc2e74c0099435a16e6e22055f205c9b96c20](#).

## 3.2 Missed Sanity Checks in Liquidation:pause()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: BoringDAO
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

In the Liquidation contract, the `pause()` function allows the dev and the trustee to pause the system with a multisig-like mechanism. As shown in the code snippet below, when the `coreDev` or one of the trustees invokes the `pause()` function, one of the flags, `shouldPauseDev` or `shouldPauseTrustee`, would be set. When the other flag is set as well, the BoringDAO contract's `pause()` handler would be called. In addition, each address in the input `pools[]` array would be checked and `pause()` if that address is a Satellite pool.

```

62     function pause(address[] memory pools) public onlyPauser {
63         if (msg.sender == coreDev) {
64             shouldPauseDev = true;
65         } else {
66             shouldPauseTrustee = true;
67         }
68         if (shouldPauseDev && shouldPauseTrustee) {
69             systemPause = true;
70             // pause the system
71             boringDAO().pause();

```

```

72         // pause satellitemapool
73         for(uint i=0; i < pools.length; i++) {
74             if(isSatellitePool[pools[i]] == true) {
75                 IPause(pools[i]).pause();
76             }
77         }
78     }
79 }

```

Listing 3.2: liquidation .sol

However, the current implementation doesn't check whether the dev and trustee want to pause the same list of `pools`. Instead, the later caller decides which `pools` would be literally paused. Besides, if the second caller fails to pass in the complete list of Satellite pools, those missed pools could not be paused anymore. The reason is that `boringDAO().pause()` invokes the `_pause()` function which would revert if it has been called before. This leads to a back-running issue. If there's a compromised trustee, she could `pause()` with an empty `pools` array right after the successful `pause()` call done by the `coreDev`. Therefore, no Satellite pool is paused.

**Recommendation** Keep all Satellite pools in an array and pause all of them in the `pause()` function without passing in the `pools` array.

**Status** This issue has been fixed in the commit: [0daf09f09fbd0b6ff4ea1f5faa9c08abe8bf3da5](#).

### 3.3 Unsafe Ownership Transition in SatellitePool

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: SatellitePool
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

#### Description

While reviewing the implementation of the `SatellitePool` contract, we notice that there's a privileged function `liquidate()` which allows the `liquidation` address to transfer all `stakingToken` out to an arbitrary account. As shown in the code snippets below, the `liquidation` address could be set by the owner with the `setLiquidation()` public function.

```

43     function liquidate(address account) public override onlyLiquidation {
44         stakingToken.safeTransfer(account, stakingToken.balanceOf(address(this)));
45     }

```

Listing 3.3: SatellitePool .sol

```

65     function setLiquidation(address liqui) public onlyOwner {
66         liquidation = liqui;
67     }

```

Listing 3.4: SatellitePool .sol

In addition, the `transferOwnership()` function allows the current owner to set a `newOwner`.

```

79     function transferOwnership(address newOwner) public onlyOwner {
80         require(newOwner != address(0), "Ownable: new owner is the zero address");
81         owner = newOwner;
82     }

```

Listing 3.5: SatellitePool .sol

However, if the `newOwner` is not the exact address of the new owner (e.g., due to a typo), nobody could own that contract anymore.

**Recommendation** Implement a two-step ownership transfer mechanism that allows the new owner to claim the ownership by signing a transaction. In addition, set the owner address to a timelock or multisig contract to prevent a compromised owner from invoking the `setLiquidation()` function and transferring all `stakingToken` out through `liquidate()`.

**Status** This issue has been fixed in the commit: [f05453eb364606d6fcfe9890ed76b1bf5e07dfa6](#).

## 3.4 Unused Interfaces

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: GovernorAlpha
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [2]

### Description

By declaring interfaces in a smart contract, we could simply call an external function of a callee contract. For example, in the GovernorAlpha contract, `interface TimelockInterface timelock` is declared to interact with the Timelock contract. In particular, `timelock.delay()` (line 298) could be used to retrieve the public variable, `delay`, defined in the Timelock contract from the GovernorAlpha contract.

```

297 interface TimelockInterface {
298     function delay() external view returns (uint);
299     function GRACE_PERIOD() external view returns (uint);
300     function acceptAdmin() external;
301     function queuedTransactions(bytes32 hash) external view returns (bool);
302     function queueTransaction(address target, uint value, string calldata signature,
        bytes calldata data, uint eta) external returns (bytes32);

```

```

303     function cancelTransaction(address target, uint value, string calldata signature,
304                               bytes calldata data, uint eta) external;
305     function executeTransaction(address target, uint value, string calldata signature,
306                                bytes calldata data, uint eta) external payable returns (bytes memory);
307 }

```

Listing 3.6: gov/GovernorAlpha.sol

However, while reviewing the declared interfaces, we identified that the `acceptAdmin()` is not used throughout the `GovernorAlpha` contract.

**Recommendation** Remove the unused interface.

**Status** This issue has been fixed in the commit: [f05453eb364606d6fcfe9890ed76b1bf5e07dfa6](https://github.com/PeckShield/audits/commit/f05453eb364606d6fcfe9890ed76b1bf5e07dfa6).

### 3.5 Weak Randomness in Tunnel::burn()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BoringDAO, Tunnel
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

#### Description

In the `Tunnel` contract, the `burn()` function allows the `BoringDAO` contract to burn `otoken` and emit a `BurnOToken` event (line 303) with a random trustee as the proposer. With the event, the random selected trustee would be notified to process the event.

```

272     function burn(address account, uint256 amount, string memory assetAddress) external
273         override onlyBoringDAO{
274         require(amount >= burnMiniLimit, "Tunnel::burn: the amount too small");
275         uint256 burnFeeAmountBToken = amount.multiplyDecimal(getRate(BURN_FEE));
276         // convert to bor amount
277         uint burnFeeAmount = oracle().getPrice(tunnelKey).multiplyDecimal(
278             burnFeeAmountBToken).divideDecimal(oracle().getPrice(BOR));
279
280         // insurance apart
281         address insurancePoolAddress = addrResolver.key2address(INSURANCE_POOL);
282         uint256 burnFeeAmountInsurance = burnFeeAmount.multiplyDecimal(
283             getRate(BURN_FEE_INSURANCE)
284         );
285
286         // pledger apart
287         uint256 burnFeeAmountPledger = burnFeeAmount.multiplyDecimal(
288             getRate(BURN_FEE_PLEDGER)
289         );

```



```

289     borERC20().transferFrom(
290         account,
291         insurancePoolAddress,
292         burnFeeAmountInsurance
293     );
294     //fee to feepool
295     borERC20().transferFrom(
296         account,
297         address(feePool()),
298         burnFeeAmountPledger
299     );
300     feePool().notifyBORFeeAmount(burnFeeAmountPledger);
301     // otoken burn
302     otokenMintBurn().burn(account, amount);
303     emit BurnOToken(
304         account,
305         amount,
306         boringDAO().getRandomTrustee(),
307         assetAddress
308     );
309 }

```

Listing 3.7: Tunnel.sol

However, randomness on Ethereum is an existing problem with no proper solution except using an oracle. As shown in the following code snippet, the `getRandomTrustee()` function uses the hash of the timestamp and difficulty of the current block to generate the pseudo-random `index`. If a bad actor uses a contract to trigger `Tunnel::burn()`, the `index` could be easily derived. Therefore, the malicious contract could revert when the `index` is not the one she need and always pick up a single trustee to process that event, which totally breaks the design.

```

105     function getRandomTrustee() public override view returns (address) {
106         uint256 trusteeCount = getRoleMemberCount(TRUSTEE_ROLE);
107         uint256 index = uint256(
108             keccak256(abi.encodePacked(now, block.difficulty))
109         )
110             .mod(trusteeCount);
111         address trustee = getRoleMember(TRUSTEE_ROLE, index);
112         return trustee;
113     }

```

Listing 3.8: BoringDAO.sol

**Recommendation** Use an oracle to feed the random seed instead of using Blockchain data.

**Status** This issue has been fixed in the commit: [f05453eb364606d6fcfe9890ed76b1bf5e07dfa6](https://github.com/PeckShield/audits/commit/f05453eb364606d6fcfe9890ed76b1bf5e07dfa6).

## 4 | Conclusion

In this audit, we thoroughly analyzed the design and implementation of the BoringDAO protocol, which is a decentralized bridge that connects multiple blockchains and supports crypto token transfers across different blockchains. During the audit, we notice that the current code base is clearly organized and those identified issues are promptly confirmed and fixed.

As a final precaution, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.



## 5 | Appendix

### 5.1 Basic Coding Bugs

---

#### 5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

#### 5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

#### 5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

#### 5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [8, 9, 10, 11, 13].
- Result: Not found
- Severity: Critical

### 5.1.5 Reentrancy

- Description: Reentrancy [14] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

### 5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

### 5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

### 5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

### 5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected revert.
- Result: Not found
- Severity: Medium

#### 5.1.10 Unchecked External Call

- Description: Whether the contract has any external call without checking the return value.
- Result: Not found
- Severity: Medium

#### 5.1.11 Gasless Send

- Description: Whether the contract is vulnerable to gasless send.
- Result: Not found
- Severity: Medium

#### 5.1.12 Send Instead Of Transfer

- Description: Whether the contract uses send instead of transfer.
- Result: Not found
- Severity: Medium

#### 5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to Out-Of-Gas exception.
- Result: Not found
- Severity: Medium

#### 5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

### 5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

### 5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

### 5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

## 5.2 Semantic Consistency Checks

---

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

## 5.3 Additional Recommendations

---

### 5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

### 5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

### 5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

### 5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low



## References

- [1] axic. Enforcing ABI length checks for return data from calls can be breaking. <https://github.com/ethereum/solidity/issues/4116>.
- [2] MITRE. CWE-1041: Use of Redundant Code. <https://cwe.mitre.org/data/definitions/1041.html>.
- [3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [4] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [5] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [6] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [7] OWASP. Risk Rating Methodology. [https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [8] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). <https://www.peckshield.com/2018/04/22/batchOverflow/>.
- [9] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). <https://www.peckshield.com/2018/05/18/burnOverflow/>.



- [10] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). <https://www.peckshield.com/2018/05/10/multiOverflow/>.
- [11] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). <https://www.peckshield.com/2018/04/25/proxyOverflow/>.
- [12] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [13] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. <https://www.peckshield.com/2018/04/28/transferFlaw/>.
- [14] Solidity. Warnings of Expressions and Control Structures. <http://solidity.readthedocs.io/en/develop/control-structures.html>.

