SMART CONTRACT AUDIT REPORT

for

COVER PROTOCOL

Prepared By: Shuxiao Wang

Hangzhou, China
Nov. 20, 2020

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	Nov. 20, 2020	Chiachih Wu	Final Release
1.0-rc	Nov. 11, 2020	Chiachih Wu	Release Candidate

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

1 Introduction

Given the opportunity to review the **COVER Protocol** design document and related smart contract source code, we outline in the 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 COVER Protocol

COVER Protocol allows DeFi users to be protected against smart contract risk. It stabilizes the dynamic and even turbulent DeFi space by instilling confidence and trust between protocols and their users. By bridging the gap between decentralized finance and traditional finance, COVER Protocol aims to open the doors of DeFi to all investors. Technically, it is designed to allow all smart contracts to be covered up to the Total Value Locked (TVL) in the covered smart contract, and also allows the market to set coverage prices as opposed to a bonding curve.

The basic information of the COVER Protocol is as follows:

Table 1.1: Basic Information of COVER Protocol

Item	Description
Issuer	COVER Protocol
Website	https://www.coverprotocol.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Nov. 20, 2020

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

- https://github.com/COVERProtocol/cover-core.git (fa58c73)
- https://github.com/COVERProtocol/cover-claim-management.git (ed354d8)

1.2 About PeckShield

PeckShield Inc. [16] 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).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [11]:

- 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) [10], 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
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	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.
Error Conditions, Return Values, Status Codes	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.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	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.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	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 `COVER` 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	1	■
Medium	2	■ ■
Low	4	■ ■ ■ ■
Informational	1	■
Total	8	

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 high-severity vulnerability 2 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings of The COVER Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Missed Error Handling on transfer()/transferFrom() Calls	Business Logic	Fixed
PVE-002	Medium	Reentrancy Risk in Protocol::addCover()	Business Logic	Fixed
PVE-003	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Fixed
PVE-004	Info.	Missed Sanity Checks in Cover::redeemCollateral()	Coding Practices	Fixed
PVE-005	Low	approve()/transferFrom() Race Condition in Cover-ERC20	Business Logic	Fixed
PVE-006	Low	Unsafe Ownership Transition	Business Logic	Fixed
PVE-007	High	Front-Running Risk in Protocol::enactClaim()	Coding Practices	Fixed
PVE-008	Low	Wrong ClaimAccepted() Event Emitted in Protocol::enactClaim()	Security Features	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 Missed Error Handling on transfer()/transferFrom() Calls

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Protocol`, `Cover`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In COVER Protocol, the collateral tokens are transferred into the `Cover` contract when users add funds into a cover through `addCover()`. On the other hand, the `_payCollateral()` routine is used to transfer out collateral tokens to users and the treasury address. The token transfers are conducted via ERC20-compatible `transferFrom()` and `transfer()` calls. However, while calling `collateral.transferFrom()`, the `Protocol` contract fails to check the return value as shown in line 173 below.

```
139 function addCover(address _collateral, uint48 _timestamp, uint256 _amount)
140   external override onlyActive returns (bool)
141 {
142   require(_amount > 0, "COVER: amount <= 0");
143   require(collateralStatusMap[_collateral] == 1, "COVER: invalid collateral");
144   require(block.timestamp < _timestamp && expirationTimestampMap[_timestamp].status ==
        1, "COVER: invalid expiration date");
145 
146   // Validate sender collateral balance is > amount
147   IERC20 collateral = IERC20(_collateral);
148   require(collateral.balanceOf(msg.sender) >= _amount, "COVER: amount > collateral
        balance");
149 
150   address addr = coverMap[_collateral][_timestamp];
151 
152   // Deploy new cover contract if not exist or if claim accepted
153   if (addr == address(0) || ICover(addr).claimNonce() != claimNonce) {
154     string memory coverName = _generateCoverName(_timestamp, collateral.symbol());
155 
```

```
    bytes memory bytecode = type(InitializableAdminUpgradeabilityProxy).creationCode;
    bytes32 salt = keccak256(abi.encodePacked(name, _timestamp, _collateral, claimNonce)
        );
    addr = Create2.deploy(0, salt, bytecode);

    bytes memory initData = abi.encodeWithSelector(COVER_INIT_SIGNITURE, coverName,
        _timestamp, _collateral, claimNonce);
    address coverImplementation = IProtocolFactory(owner()).coverImplementation();
    InitializableAdminUpgradeabilityProxy(payable(addr)).initialize(
      coverImplementation,
      IProtocolFactory(owner()).governance(),
      initData
    );

    activeCovers.push(addr);
    coverMap[_collateral][_timestamp] = addr;
  }

  // move collateral to the cover contract and mint CovTokens to sender
  collateral.transferFrom(msg.sender, addr, _amount);
  ICover(addr).mint(_amount, msg.sender);
  return true;
}
```

Listing 3.1: Protocol.sol

In lines 168 – 169 of `Cover::_payCollateral()`, the return value of `collateralToken.transfer()` is ignored as well.

```
160 function _payCollateral(address _receiver, uint256 _amount) private {
161   IProtocolFactory factory = IProtocolFactory(_factory());
162   uint256 redeemFeeNumerator = factory.redeemFeeNumerator();
163   uint256 redeemFeeDenominator = factory.redeemFeeDenominator();
164   uint256 fee = _amount.mul(redeemFeeNumerator).div(redeemFeeDenominator);
165   address treasury = factory.treasury();
166   IERC20 collateralToken = IERC20(collateral);
167
168   collateralToken.transfer(_receiver, _amount.sub(fee));
169   collateralToken.transfer(treasury, fee);
170 }
```

Listing 3.2: Cover.sol

When the collateral token contract fails to revert for whatever reason, the caller of `transfer()` /`transferFrom()` functions cannot ensure if the tokens are transferred successfully. In addition, certain ERC20 token contracts do not have a return value in `transfer()`/`transferFrom()` functions. To deal with these incompatibility issues, we suggest to use OpenZeppelin's `SafeERC20` library to accommodate various idiosyncrasies in current ERC20 implementations.

Recommendation Use `OpenZeppelin`'s `SafeERC20` routines when interacting with ERC20 contracts.

Status This issue has been addressed by using `SafeERC20` in this commit: ae36588.

3.2 Reentrancy Risk in Protocol::addCover()

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Protocol`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [19] exploit, and the recent `Uniswap/Lendf.Me` hack [17].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. In the `Protocol` contract, the `addCover()` function (see the code snippet below) is provided to purchase or add funds to a cover by externally calling a token contract to transfer assets into the `Cover`. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. Apparently, the interaction with the external contract (line 173) starts before effecting the update on internal states (lines 174), hence violating the principle.

```
139 function addCover(address _collateral, uint48 _timestamp, uint256 _amount)
140   external override onlyActive returns (bool)
141 {
142   require(_amount > 0, "COVER: amount <= 0");
143   require(collateralStatusMap[_collateral] == 1, "COVER: invalid collateral");
144   require(block.timestamp < _timestamp && expirationTimestampMap[_timestamp].status ==
        1, "COVER: invalid expiration date");
145
146   // Validate sender collateral balance is > amount
147   IERC20 collateral = IERC20(_collateral);
148   require(collateral.balanceOf(msg.sender) >= _amount, "COVER: amount > collateral
        balance");
149
150   address addr = coverMap[_collateral][_timestamp];
151
152   // Deploy new cover contract if not exist or if claim accepted
153   if (addr == address(0) || ICover(addr).claimNonce() != claimNonce) {
154     string memory coverName = _generateCoverName(_timestamp, collateral.symbol());
```

```
155
156     bytes memory bytecode = type(InitializableAdminUpgradeabilityProxy).creationCode;
157     bytes32 salt = keccak256(abi.encodePacked(name, _timestamp, _collateral, claimNonce)
            );
158     addr = Create2.deploy(0, salt, bytecode);
159
160     bytes memory initData = abi.encodeWithSelector(COVER_INIT_SIGNITURE, coverName,
            _timestamp, _collateral, claimNonce);
161     address coverImplementation = IProtocolFactory(owner()).coverImplementation();
162     InitializableAdminUpgradeabilityProxy(payable(addr)).initialize(
163       coverImplementation,
164       IProtocolFactory(owner()).governance(),
165       initData
166     );
167
168     activeCovers.push(addr);
169     coverMap[_collateral][_timestamp] = addr;
170   }
171
172   // move collateral to the cover contract and mint CovTokens to sender
173   collateral.transferFrom(msg.sender, addr, _amount);
174   ICover(addr).mint(_amount, msg.sender);
175   return true;
176 }
```

Listing 3.3: Protocol.sol

Specifically, in the case when `Collateral` is an ERC777 token, a bad actor could hijack a `addCover()` call before `collateral.transferFrom()` in line 173 with a callback function. Within the callback function, they could call the `addCover()` function to add funds into the same cover again. Since the collateral tokens are not transferred out yet, the `collateral.balanceOf(msg.sender)>= _amount` check in line 148 would pass again. As mentioned in Section 3.1, if `collateral.transferFrom()` fails to revert when there's not enough token balance to transfer, the bad actor could exploit the reentrancy bug again and again to mint unlimited amount of CLAIM/NOCLAIM covTokens.

Recommendation Apply the `checks-effects-interactions` design pattern or add the reentrancy guard modifier.

Status This issue has been fixed by adding the reentrancy guard modifier in this commit: 651617f.

3.3 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Protocol`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In COVER Protocol, the `Protocol` contract is designed to be the main entry point for interaction with users. In particular, one entry routine, i.e., `addCover()`, accepts asset transfer-in and mints the corresponding `covToken` to represent the cover that the user purchased. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of COVER Protocol. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
function addCover(address _collateral, uint48 _timestamp, uint256 _amount)
  external override onlyActive returns (bool)
{
  require(_amount > 0, "COVER: amount <= 0");
  require(collateralStatusMap[_collateral] == 1, "COVER: invalid collateral");
  require(block.timestamp < _timestamp && expirationTimestampMap[_timestamp].status ==
      1, "COVER: invalid expiration date");

  // Validate sender collateral balance is > amount
  IERC20 collateral = IERC20(_collateral);
  require(collateral.balanceOf(msg.sender) >= _amount, "COVER: amount > collateral
      balance");

  address addr = coverMap[_collateral][_timestamp];

  // Deploy new cover contract if not exist or if claim accepted
  if (addr == address(0)   ICover(addr).claimNonce() != claimNonce) {
    string memory coverName = _generateCoverName(_timestamp, collateral.symbol());

    bytes memory bytecode = type(InitializableAdminUpgradeabilityProxy).creationCode;
    bytes32 salt = keccak256(abi.encodePacked(name, _timestamp, _collateral, claimNonce)
      );
    addr = Create2.deploy(0, salt, bytecode);

    bytes memory initData = abi.encodeWithSelector(COVER_INIT_SIGNITURE, coverName,
        _timestamp, _collateral, claimNonce);
    address coverImplementation = IProtocolFactory(owner()).coverImplementation();
    InitializableAdminUpgradeabilityProxy(payable(addr)).initialize(
      coverImplementation,
```

```
            IProtocolFactory(owner()).governance(),
            initData
        );

        activeCovers.push(addr);
        coverMap[_collateral][_timestamp] = addr;
    }

    // move collateral to the cover contract and mint CovTokens to sender
    collateral.transferFrom(msg.sender, addr, _amount);
    ICover(addr).mint(_amount, msg.sender);
    return true;
}
```

Listing 3.4: Protocol.sol

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `addCover()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the `Cover` contract before and after the `transferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted to be the collateral tokens. In fact, COVER Protocol is indeed in the position to effectively regulate the set of assets that can be used as collaterals. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()`/`transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status This issue has been addressed by checking the balance before and after the `transferFrom()` call in this commit: 651617f.

3.4 Missed Sanity Checks in Cover::redeemCollateral()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Cover`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [3]

Description

In the `Cover` contract, the `redeemCollateral()` function allows covToken holders to get collateral tokens back by burning CLAIM/NOCLAIM covTokens. However, the current implementation fails to check the given argument in `_amount`. As a result, covToken holders could `redeemCollateral(0)` and burn/transfer zero tokens, which is a waste of gas.

```
function redeemCollateral(uint256 _amount) external override onlyNotExpired {
  _noClaimAcceptedCheck(); // save gas than modifier

  ICoverERC20 _claimCovToken = claimCovToken; // save gas
  ICoverERC20 _noclaimCovToken = noclaimCovToken; // save gas

  require(_amount <= _claimCovToken.balanceOf(msg.sender), "COVER: low CLAIM balance");
  require(_amount <= _noclaimCovToken.balanceOf(msg.sender), "COVER: low NOCLAIM balance
      ");

  _claimCovToken.burnByCover(msg.sender, _amount);
  _noclaimCovToken.burnByCover(msg.sender, _amount);
  _payCollateral(msg.sender, _amount);
}
```

Listing 3.5: Cover.sol

Recommendation Ensure `_amount > 0` in `Cover::redeemCollateral()`.

Status This issue has been addressed by `require(_amount > 0, "COVER: amount is 0")` in this commit: 651617f.

3.5 approve()/transferFrom() Race Condition in CoverERC20

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `CoverERC20`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

As an ERC20 token contract, `CoverERC20` implements the `approve()` and `transferFrom()` functions to allow a spender address to spend the owner's tokens, which is an essential feature in DeFi universe. However, one well-known race condition vulnerability is identified in the `CoverERC20` contract [2].

```
/// @notice Standard ERC20 function
function approve(address spender, uint256 amount) external virtual override returns (
    bool) {
  _approve(msg.sender, spender, amount);
  return true;
}

/// @notice Standard ERC20 function
function transferFrom(address sender, address recipient, uint256 amount)
  external virtual override returns (bool)
{
  _transfer(sender, recipient, amount);
  _approve(sender, msg.sender, allowances[sender][msg.sender].sub(amount, "CoverERC20:
      transfer amount exceeds allowance"));
  return true;
}
```

Listing 3.6: CoverERC20.sol

Specifically, when Bob approves Alice for spending his 100 covToken but re-set the approval to 200 covToken, Alice could front-run the second `approve()` call with a `transferFrom()` call to spend $100 + 200 = 300$ covToken owned by Bob.

Recommendation Ensure that the allowance is 0 while setting a new allowance. An alternative solution is implementing the `increaseAllowance()` and `decreaseAllowance()` functions which increase/decrease the allowance instead of setting the allowance directly.

Status This issue has been addressed by adding the `increaseAllowance()` and `decreaseAllowance()` functions in this commit 651617f.

3.6 Unsafe Ownership Transition

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Ownable`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In COVER Protocol, the `Ownable` contract is widely used for ownership management in contracts such as `Protocol`, `Cover`, etc. When the contract owner needs to transfer the ownership to another address, she could use the `transferOwnership()` function with a `newOwner` address.

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
}
```

Listing 3.7: utils/Ownable.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.

Status This issue has been addressed by adding the `claimOwnership()` function which allows the new owner to claim the ownership in this commit: 651617f.

3.7 Front-Running Risk in Protocol::enactClaim()

- ID: PVE-007
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Protocol`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1099 [4]

Description

In `Protocol` contract, the `enactClaim()` function allows the `claimManager` to decide a claim. Specifically, anyone could file a claim through the `ClaimManagement` contract while privileged users such as the `auditor` could validate the claim, and finally decide the claim through `Protocol::enactClaim()`. When a claim is decided by `enactClaim()`, CLAIM covToken holders could get the collateral tokens by burning CLAIM covTokens. When a claim is not decided and expired, NOCLAIM covToken holders could get the collateral tokens by burning NOCLAIM covTokens. While reviewing the implementation, we identify a front-running bug that a bad actor could get both CLAIM and NOCLAIM corresponding collateral tokens.

```
function enactClaim(
  uint16 _payoutNumerator,
  uint16 _payoutDenominator,
  uint256 _protocolNonce
)
 external override returns (bool)
{
  require(_protocolNonce == claimNonce, "COVER: nonces do not match");
  require(_payoutNumerator <= _payoutDenominator && _payoutNumerator > 0, "COVER: payout % is not in (0%, 100%]");
  require(msg.sender == IProtocolFactory(owner()).claimManager(), "COVER: caller not claimManager");

  claimNonce = claimNonce.add(1);
  delete activeCovers;
  claimDetails.push(ClaimDetails(
    _payoutNumerator,
    _payoutDenominator,
    uint48(block.timestamp)
  ));
  emit ClaimAccepted(claimNonce);
  return true;
}
```

Listing 3.8: Protocol.sol

For each `enactClaim()`'ed claim, CLAIM covToken holders could redeem the claim by calling `redeemClaim()`. With CLAIM covTokens burned, the corresponding collateral tokens are transferred to the CLAIM covToken holders through `_paySender()` (line 76).

```
69 function redeemClaim() external override {
70   IProtocol protocol = IProtocol(owner());
71   require(protocol.claimNonce() > claimNonce, "COVER: no claim accepted");
72
73   (uint16 _payoutNumerator, uint16 _payoutDenominator, uint48 _timestamp) = _claimDetails();
74   require(block.timestamp >= uint256(_timestamp) + protocol.claimRedeemDelay(), "COVER: not ready");
75
76   _paySender(
77     claimCovToken,
78     uint256(_payoutNumerator),
79     uint256(_payoutDenominator)
80   );
```

```
}
```

Listing 3.9: Cover.sol

If the claim has not been `enactClaim()`'ed yet, covToken holders could also burn the NOCLAIM covTokens and get collateral tokens back when the current time exceeds the expiration time + the `noclaimRedeemDelay` (line 108).

```
88 function redeemNoclaim() external override {
89   IProtocol protocol = IProtocol(owner());
90   if (protocol.claimNonce() > claimNonce) {
91     // protocol has an accepted claim
92
93     (uint16 _payoutNumerator, uint16 _payoutDenominator, uint48 _timestamp) =
         _claimDetails();
94
95     // If claim payout is 100%, nothing is left for NOCLAIM covToken holders
96     require(_payoutNumerator < _payoutDenominator, "COVER: claim payout 100%");
97
98     require(block.timestamp >= uint256(_timestamp) + protocol.claimRedeemDelay(), "COVER
         : not ready");
99     _paySender(
100      noclaimCovToken,
101      uint256(_payoutDenominator).sub(uint256(_payoutNumerator)),
102      uint256(_payoutDenominator)
103    );
104  } else {
105    // protocol has no accepted claim
106
107    require(block.timestamp >= uint256(expirationTimestamp) + protocol.
         noclaimRedeemDelay(), "COVER: not ready");
108    _paySender(noclaimCovToken, 1, 1);
109  }
110 }
```

Listing 3.10: Cover.sol

However, when the covToken holder is ready to redeem the NOCLAIM covToken (i.e., `block.timestamp >= uint256(expirationTimestamp)+ protocol.noclaimRedeemDelay()`), she could wait until someone calls `enactClaim()` and front-run that `enactClaim()` transaction with a `redeemNoclaim()` call. Two days later, she could call the `redeemClaim()` function to get the double pay. This is possible since the `ClaimManagement` contract fails to check if a claim could be filed or decided after `expirationTimestamp + 10 days`.

Recommendation Prevent the claim from being filed and decided after the claim is expired.

Status This issue has been addressed by checking the current time with the allowed file/decide time window while filing/deciding a claim in this commit: 58d6af0.

3.8 Wrong ClaimAccepted() Event Emitted in Protocol::enactClaim()

- ID: PVE-008
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Protocol`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [5]

Description

As mentioned in Section 3.7, the `enactClaim()` function in `Protocol` contract allows the `claimManager` to update the `claimNonce` and add a new entry in the `claimDetails` array. When a claim is decided, the `ClaimAccepted()` event is emitted with the `claimNonce`. However, the `claimNonce` is not exactly the nonce of the claim just decided by the `claimManager`. Instead, the next nonce (i.e., `claimNonce+1`) is emitted with `ClaimAccepted()`, which is not what the event name suggesting.

```
function enactClaim(
  uint16 _payoutNumerator,
  uint16 _payoutDenominator,
  uint256 _protocolNonce
)
 external override returns (bool)
{
  require(_protocolNonce == claimNonce, "COVER: nonces do not match");
  require(_payoutNumerator <= _payoutDenominator && _payoutNumerator > 0, "COVER: payout
      % is not in (0%, 100%]");
  require(msg.sender == IProtocolFactory(owner()).claimManager(), "COVER: caller not
      claimManager");

  claimNonce = claimNonce.add(1);
  delete activeCovers;
  claimDetails.push(ClaimDetails(
    _payoutNumerator,
    _payoutDenominator,
    uint48(block.timestamp)
  ));
  emit ClaimAccepted(claimNonce);
  return true;
}
```

Listing 3.11: Protocol.sol

Recommendation Emit `ClaimAccpeted(_protocolNonce)` in the end of `enactClaim()`.

Status This issue has been addressed by emitting `ClaimAccpeted()` with `_protocolNonce` in this commit: ae36588.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `COVER` protocol, which is designed to provide insurance coverage to blockchain smart contracts. 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 [12, 13, 14, 15, 18].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [20] 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] HaleTom. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack. https://github.com/ethereum/EIPs/issues/738.

[3] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[4] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[5] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[7] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[8] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[9] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[10] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[11] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[12] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[13] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[14] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[15] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

[16] PeckShield. PeckShield Inc. https://www.peckshield.com.

[17] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[18] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[19] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.

[20] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.