SMART CONTRACT AUDIT REPORT

for

AAVE

Prepared By: Shuxiao Wang

Hangzhou, China

August 5, 2020

Document Properties

Client	Aave
Title	Smart Contract Audit Report
Target	CreditDelegationVault
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	August 5, 2020	Xuxian Jiang	Final Release
1.0-rc1	August 2, 2020	Xuxian Jiang	Additional Findings
0.1	July 31, 2020	Huaguo Shi	Initial Draft

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 **CreditDelegationVault** smart contract source code, we in the report outline 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 contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About CreditDelegationVault

Aave is a decentralized non-custodial money market protocol where users can participate as depositors or borrowers. Depositors provide liquidity to the market to earn a passive income, while borrowers are able to borrow in an overcollateralized (perpetually) or undercollateralized (one-block flashloan) fashion. As the name indicates, the smart contract CreditDelegationVault can be used to create so-called credit delegation vault. With the vault, an Aave depositor could delegate his credit line to a third party to withdraw the credit. In the meantime, the Aave depositor can set different kind of parameters on the vault, such as currency, which rate mode can draw (variable or stable), and what currency can be drawn.

The basic information of CreditDelegationVault is as follows:

Table 1.1: Basic Information of CreditDelegationVault

Item	Description
Issuer	Aave
Website	https://aave.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 5, 2020

In the following, we show the repository of reviewed code (verified in etherscan.io) used in this audit. We need to point out that CreditDelegationVault re-uses the same trusted oracles in Aave with timely market price feeds and the oracles themselves are not part of this audit.

- https://etherscan.io/address/0x22fad18e5c1a8c483aca2132f6725c7da6cfb799#code

1.2 About PeckShield

PeckShield Inc. [17] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of 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 OWASP Risk Rating Methodology [12]:

- 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.

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

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 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) [11], 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 an 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 Logics	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 CreditDelegationVault implementation. During the first phase of our audit, we studied the smart contract source code and ran 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	3	■ ■ ■
Low	3	■ ■ ■
Informational	2	■ ■
Total	9	

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 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 vulnerabilities, 3 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Info.	External Declaration of Only-Externally-Invoked Functions	Coding Practices	Fixed
PVE-002	Medium	Mixed Spending Limit Denominations	Business Logics	Fixed
PVE-003	Low	Lack of Access Control in activate()	Security Features	Fixed
PVE-004	Info.	Code Simplification in deployVault()	Coding Practices	Fixed
PVE-005	Low	Improved Sanity Checks in borrow()	Security Features	Fixed
PVE-006	Low	Avoidance of Duplicate Reserves in activate()	Business Logics	Fixed
PVE-007	Medium	Incompatibility With Deflationary Tokens	Time and State	Confirmed
PVE-008	Medium	No Return of Possible User Overpayment	Business Logics	Confirmed
PVE-009	High	Unsupported ETH Borrow And Repay in Vaults	Business Logics	Confirmed

Please refer to Section 3 for details.

3 | Detailed Results

3.1 External Declaration of Only-Externally-Invoked Functions

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: iCollateralVaultProxy
- Category: Coding Practices [9]
- CWE subcategory: CWE-287 [3]

Description

The CreditDelegationVault contracts provide a number of interface functions that are designed to be called only for external users. Many of these functions are defined as public. In public functions, Solidity immediately copies array arguments to memory, while external functions can read directly from calldata. Note that memory allocation can be expensive, whereas reading from calldata is not. So when these functions are not used within the contract, it's always suggested to define them as external instead of public. After analyzing the code, we recommend changing the following functions from public to external:

```
    function limit(address vault, address spender) public view returns (uint)
    function borrowers(address vault) public view returns (address[] memory)
    function borrowerVaults(address spender) public view returns (address[] memory)
    function increaseLimit(address vault, address spender, uint addedValue) public
    function decreaseLimit(address vault, address spender, uint subtractedValue) public
    function setModel(iCollateralVault vault, uint model) public
    function setBorrow(iCollateralVault vault, address borrow) public
    function repay(iCollateralVault vault, address reserve, uint amount) public
    function deployVault() public returns (address)
```

Listing 3.1: iCollateralVaultProxy

```
    function setModel(uint _model) public onlyOwner
    function setBorrow(address _asset) public onlyOwner
    function getReserves() public view returns (address[] memory)
```

Listing 3.2: iCollateralVault

Recommendation Revise the affected functions from being `public` to `external`.

3.2 Mixed Spending Limit Denominations

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `iCollateralVaultProxy`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [6]

Description

In the `iCollateralVaultProxy` contract, the `setborrow()` function is used to set the token intended for borrowing. For the specified borrow token, the associated vault owner can specify who are those legitimate borrowers and what amounts are permitted to borrow. However, when the borrow token has been changed, the related spending limits are not accordingly updated. Notice that the market price of one borrow token is likely not the same as another borrow token. Therefore, the spending limit for one borrow token should not be the same for another borrow token. Otherwise, the difference may be leveraged by a borrower to draw a larger credit line than permitted.

Specifically, the spending limits are defined in the following private member `_limits`. The comment above the member definition shows the spending limits per user are `measured in dollars` (scaled by $1e8$).

```
    // Spending limits per user measured in dollars 1e8
    mapping (address => mapping (address => uint)) private _limits;
```

Listing 3.3: iCollateralVaultProxy.sol

The enforcement of spending limits is performed in `borrow()` (line 398) by calculating the `_borrow` amount and ensuring the amount is within the current limit of the spender. We notice that if the vault's borrow token has not been set, the `_borrow` amount is denominated indeed in dollars. But once the borrow token has been set up, the amount becomes denominated in the borrow token. If the borrow token has been changed to another token, the amount is denominated in the new borrow token. Despite these changes, the spending limits however remain the same, leading to possible exploitation by a borrower to draw a larger credit line than permitted.

```
392     // amount needs to be normalized
393     function borrow(iCollateralVault vault, address reserve, uint amount) external {
394         uint _borrow = amount;
395         if (vault.asset() == address(0)) {
396             _borrow = getReservePriceUSD(reserve).mul(amount);
397         }
```

```
        _approve(address(vault), msg.sender, _limits[address(vault)][msg.sender].sub(
            _borrow, "borrow amount exceeds allowance"));
        vault.borrow(reserve, amount, msg.sender);
        emit Borrow(address(vault), msg.sender, reserve, amount);
    }
```

Listing 3.4: iCollateralVaultProxy.sol

Recommendation Revise the `setborrow()` logic to ensure the spending limits are reset when the borrow token is being changed. An example revision is shown below.

```
    function setBorrow(iCollateralVault vault, address borrow) public {
        require(isVaultOwner(address(vault), msg.sender), "!owner");
        vault.setBorrow(borrow);
        resetLimit(vault);
        emit SetBorrow(address(vault), msg.sender, borrow);
    }

    function resetLimit(address vault) internal {
        for (uint i = 0; i< _borrowers[vault].length-1; i++){
            if(_borrowers[vault][i] != 0){
                _limits[vault][_borrowers[vault][i]] = 0;
            }
        }
    }
```

Listing 3.5: iCollateralVaultProxy.sol (revised)

In the new `resetLimit()` routine, we essentially reset the spending limits back to 0 for all possible vault spenders.

3.3 Lack of Access Control in activate()

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `iCollateralVault`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `iCollateralVault` contract, the `activate()` function allows for any deposit from LPs to be used as collateral. We notice that there is no access control restriction imposed on this particular function and anyone is allowed to invoke it.

```
    // LP deposit, anyone can deposit/topup
    function activate(address reserve) external {
```

```
        _activeReserves.push(reserve);
        Aave(getAave()).setUserUseReserveAsCollateral(reserve, true);
    }
```

Listing 3.6: iCollateralVaultProxy.sol

Our assessment shows that the lack of access control may not cause any damage on the vault asset. However, it does unnecessarily expose the call to `Aave(getAave()).setUserUseReserveAsCollateral (reserve, true)`. The call may be abused to enable any reserve in Aave as collateral (even with tiny dust balance). While it remains to assess the scope of possible impact or further explore any meaningful abuse, it is suggested to add necessary access control for better assurance, just like other routines such as `withdraw()`.

Recommendation Add the necessary access control restriction to the exposed `activate()`.

```
    // LP deposit, anyone can deposit/topup
    function activate(address reserve) external onlyOwner{
        _activeReserves.push(reserve);
        Aave(getAave()).setUserUseReserveAsCollateral(reserve, true);
    }
```

Listing 3.7: iCollateralVaultProxy.sol (revised)

3.4 Code Simplification in deployVault()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `iCollateralVaultProxy`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1041 [2]

Description

In the `iCollateralVaultProxy` contract, the `deployVault()` function allows liquidity providers to deploy the so-called credit delegation vault. The two related bookkeeping arrays, i.e. `_vaults` and `_ownedVaults`, are accordingly updated with the new vault deployment to properly mark the vault address and set up the owner.

```
    function deployVault() public returns (address) {
        address vault = address(new iCollateralVault());

        // Mark address as vault
        _vaults[vault] = msg.sender;

        // Set vault owner
        address[] storage owned = _ownedVaults[msg.sender];
```

```
        owned.push(vault);
        _ownedVaults[msg.sender] = owned;
        emit DeployVault(vault, msg.sender);
        return vault;
    }
```

Listing 3.8: iCollateralVaultProxy.sol

The code snippet can be simplified a bit to remove the use of an internal variable and improve the readability.

Recommendation Simplify the `deployVault()` routine as follows.

```
    function deployVault() public returns (address) {
        address vault = address(new iCollateralVault());

        // Mark address as vault
        _vaults[vault] = msg.sender;

        // Set vault owner
        _ownedVaults[msg.sender].push(vault);
        emit DeployVault(vault, msg.sender);
        return vault;
    }
```

Listing 3.9: iCollateralVaultProxy.sol

3.5 Improved Sanity Checks in borrow()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `iCollateralVaultProxy`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `iCollateralVaultProxy` contract, the `borrow()` function allows users to borrow a specific amount of the reserve currency. The argument `vault` is an vault instance of `iCollateralVault`. However, the sanity checks of the vault are not thorough and an invalid vault may be given. The execution may still lead to the generation of a misleading or even erroneous event entry `Borrow`, which otherwise can be avoided in the first place.

```
    // amount needs to be normalized
    function borrow(iCollateralVault vault, address reserve, uint amount) external {
        uint _borrow = amount;
        if (vault.asset() == address(0)) {
```

```
            _borrow = getReservePriceUSD(reserve).mul(amount);
        }
        _approve(address(vault), msg.sender, _limits[address(vault)][msg.sender].sub(
            _borrow, "borrow amount exceeds allowance"));
        vault.borrow(reserve, amount, msg.sender);
        emit Borrow(address(vault), msg.sender, reserve, amount);
    }
```

Listing 3.10: iCollateralVaultProxy.sol

Recommendation Revise the above sanity checks by ensuring the given vault is legitimate.

```
    // amount needs to be normalized
    function borrow(iCollateralVault vault, address reserve, uint amount) external {
        require(isVault(address(vault)), "not a vault");
        uint _borrow = amount;
        if (vault.asset() == address(0)) {
            _borrow = getReservePriceUSD(reserve).mul(amount);
        }
        _approve(address(vault), msg.sender, _limits[address(vault)][msg.sender].sub(
            _borrow, "borrow amount exceeds allowance"));
        vault.borrow(reserve, amount, msg.sender);
        emit Borrow(address(vault), msg.sender, reserve, amount);
    }
```

Listing 3.11: iCollateralVaultProxy.sol

3.6 Avoidance of Duplicate Reserves in activate()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `iCollateralVaultProxy`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [6]

Description

In the `iCollateralVault` contract, the `deposit()` is used to top up the vault's reserve as collateral. Each invocation of `deposit()` will call the vault's `activate()` function, which `pushes` the reserve one more time to the vault storage `_activeReserves`. In other words, when `deposit()` are called multiple times, the `activate()` function would be called multiple times, resulting in multiple `_activeReserves.push()`. The additional `pushes` on the same reserve are a waste of storage use.

```
    // LP deposit, anyone can deposit/topup
    function deposit(iCollateralVault vault, address aToken, uint amount) external {
        require(isVault(address(vault)), "!vault");
        IERC20(aToken).safeTransferFrom(msg.sender, address(vault), amount);
```

```
        vault.activate(AaveToken(aToken).underlyingAssetAddress());
        emit Deposit(address(vault), msg.sender, aToken, amount);
    }
```

Listing 3.12: iCollateralVaultProxy.sol

```
    // LP deposit, anyone can deposit/topup
    function activate(address reserve) external {
        _activeReserves.push(reserve);
        Aave(getAave()).setUserUseReserveAsCollateral(reserve, true);
    }
```

Listing 3.13: iCollateralVaultProxy.sol

An alternative approach will be to recognize existing reserves already pushed into the `_activeReserves` storage. For next `deposit()` on an existing reserve, simply skip the `push` operation. Note that the subsequent call of `Aave(getAave()).setUserUseReserveAsCollateral(reserve, true)` still needs to be performed as a later `repay()` may clear the collateral flag (maintained in the Aave protocol) associated with the user's reserve.

Recommendation Avoid duplicate reserves in `activate()`.

```
    // LP deposit, anyone can deposit/topup
    function activate(address reserve) external onlyOwner {
        if(_activeReserves[reserve] == address(0)){
            _activeReserves.push(reserve);
        }
        Aave(getAave()).setUserUseReserveAsCollateral(reserve, true);
    }
```

Listing 3.14: iCollateralVaultProxy.sol

3.7 Incompatibility With Deflationary Tokens

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `iCollateralVaultProxy`, `iCollateralVault`
- Category: Time and State [8]
- CWE subcategory: CWE-362 [4]

Description

CreditDelegationVault acts as a trustless intermediary between credit providers and borrowing users. The credit providers `deposit` certain amount of `aToken` assets into the CreditDelegationVault as collateral and allow for spenders to `borrow` (per the credit delegation). The spender can later `repay` the borrowed amount (plus necessary interest).

For the above two borrower's operations, i.e., `borrow` and `repay`, CreditDelegationVault provides low-level routines to transfer assets into or out of the vault (see the code snippet below). 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.

```
    // amount needs to be normalized
    function borrow(address reserve, uint amount, address to) external nonReentrant
        onlyOwner {
        require(asset == reserve   asset == address(0), "reserve not available");
        // LTV logic handled by underlying
        Aave(getAave()).borrow(reserve, amount, model, 7);
        IERC20(reserve).safeTransfer(to, amount);
    }

    function repay(address reserve, uint amount) external nonReentrant onlyOwner {
        // Required for certain stable coins (USDT for example)
        IERC20(reserve).approve(address(getAaveCore()), 0);
        IERC20(reserve).approve(address(getAaveCore()), amount);
        Aave(getAave()).repay(reserve, amount, address(uint160(address(this))));
    }
```

Listing 3.15: iCollateralVaultProxy.sol

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every `transfer` or `transferFrom`. As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit` and `repay`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of CreditDelegationVault and affects protocol-wide operation and maintenance.

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `transfer` or `transferFrom` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer`/`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 into Aave for borrowing. However, as a plug-in component, CreditDelegationVault may not have the control of the process. Instead, it can monitor the introduction of such tokens and prevent vaults from using such tokens.

Recommendation Apply necessary mitigation mechanisms to regulate non-compliant or unnecessarily-extended ERC20 tokens.

3.8 No Return of Possible User Overpayment

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: iCollateralVaultProxy, iCollateralVault
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [6]

Description

In the iCollateralVault contract, the repay() function allows the borrower to repay previous borrows (internally billed in the name of the vault). However, in the likely case that the borrower may overpay the borrow amount, the entire amount is transferred to the vault and the overpaid portion does not automatically refunded back to the borrower.

In the following, we show below related code snippet in repay(). Notice that vault.repay(reserve , amount) (line 406) is invoked after the transferring of repayment from the borrower to the vault. Inside the vault.repay(), it directly calls into the Aave protocol, i.e., Aave(getAave()).repay(reserve , amount, address(uint160(address(this)))) (line 286).

```
403     function repay(iCollateralVault vault, address reserve, uint amount) public {
404         require(isVault(address(vault)), "not a vault");
405         IERC20(reserve).safeTransferFrom(msg.sender, address(vault), amount);
406         vault.repay(reserve, amount);
407         emit Repay(address(vault), msg.sender, reserve, amount);
408     }
```

Listing 3.16: iCollateralVaultProxy.sol

```
282     function repay(address reserve, uint amount) external nonReentrant onlyOwner {
283         // Required for certain stable coins (USDT for example)
284         IERC20(reserve).approve(address(getAaveCore()), 0);
285         IERC20(reserve).approve(address(getAaveCore()), amount);
286         Aave(getAave()).repay(reserve, amount, address(uint160(address(this))));
287     }
```

Listing 3.17: iCollateralVault.sol

The internal logic of Aave.repay() shows that it first determines the actual paybackAmount and then transfers that amount to the Aave core. In other words, the Aave protocol will not transfer the payment more than necessary from the "borrower" (the vault in our case). With the introduction of CreditDelegationVault, the overpaid amount simply stays in the vault, not back to the actual borrower. We point out that ERC20-compliant tokens staying in the vault can be retrieved by the vault owner only (via the `withdraw()` routine).

```
    function repay(address _reserve, uint256 _amount, address payable _onBehalfOf)
        external
        payable
        nonReentrant
        onlyActiveReserve(_reserve)
        onlyAmountGreaterThanZero(_amount)
    {
        // Usage of a memory struct of vars to avoid "Stack too deep" errors due to
            local variables
        RepayLocalVars memory vars;

        (
            vars.principalBorrowBalance,
            vars.compoundedBorrowBalance,
            vars.borrowBalanceIncrease
        ) = core.getUserBorrowBalances(_reserve, _onBehalfOf);

        vars.originationFee = core.getUserOriginationFee(_reserve, _onBehalfOf);
        vars.isETH = EthAddressLib.ethAddress() == _reserve;

        require(vars.compoundedBorrowBalance > 0, "The user does not have any borrow
            pending");

        require(
            _amount != UINT_MAX_VALUE  msg.sender == _onBehalfOf,
            "To repay on behalf of an user an explicit amount to repay is needed."
        );

        //default to max amount
        vars.paybackAmount = vars.compoundedBorrowBalance.add(vars.originationFee);

        if (_amount != UINT_MAX_VALUE && _amount < vars.paybackAmount) {
            vars.paybackAmount = _amount;
        }

        ...
    }
```

Listing 3.18: LendingPool.sol (Aave)

Recommendation Calculate the required payment amount and return any overpaid amount, if any, back to the borrower.

3.9 Unsupported ETH Borrow And Repay in Vaults

- ID: PVE-008
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `iCollateralVault`
- Category: Business Logics [10]
- CWE subcategory: CWE-754 [5]

Description

The Aave protocol supports a number of token assets, including `ETH`. Internally, the `ETH` asset is represented with a mock reserve address, i.e., `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`. With the introduction of CreditDelegationVault, the borrower may naturally borrow or repay the `ETH` asset by following the same fashion as the Aave protocol, including directly transferring `ETH` to the vault or using the same mock reserve address. Unfortunately, the CreditDelegationVault contract in the current prototype does not work the same way regarding the borrow or repay of `ETH`.

```
392     // amount needs to be normalized
393     function borrow(iCollateralVault vault, address reserve, uint amount) external {
394         uint _borrow = amount;
395         if (vault.asset() == address(0)) {
396             _borrow = getReservePriceUSD(reserve).mul(amount);
397         }
398         _approve(address(vault), msg.sender, _limits[address(vault)][msg.sender].sub(
                _borrow, "borrow amount exceeds allowance"));
399         vault.borrow(reserve, amount, msg.sender);
400         emit Borrow(address(vault), msg.sender, reserve, amount);
401     }

403     function repay(iCollateralVault vault, address reserve, uint amount) public {
404         require(isVault(address(vault)), "not a vault");
405         IERC20(reserve).safeTransferFrom(msg.sender, address(vault), amount);
406         vault.repay(reserve, amount);
407         emit Repay(address(vault), msg.sender, reserve, amount);
408     }
```

Listing 3.19: iCollateralVaultProxy.sol

In particular, with the so-called credit delegation, if a borrower intends to borrow `ETH` by using the above mock reserve address, the operation, i.e., `vault.borrow(reserve, amount, msg.sender)` (line 399), relays to the Aave protocol with the same set of arguments.

```
    // amount needs to be normalized
    function borrow(address reserve, uint amount, address to) external nonReentrant
        onlyOwner {
        require(asset == reserve   asset == address(0), "reserve not available");
        // LTV logic handled by underlying
```

```
278            Aave(getAave()).borrow(reserve, amount, model, 7);
279            IERC20(reserve).safeTransfer(to, amount);
280        }

282        function repay(address reserve, uint amount) external nonReentrant onlyOwner {
283            // Required for certain stable coins (USDT for example)
284            IERC20(reserve).approve(address(getAaveCore()), 0);
285            IERC20(reserve).approve(address(getAaveCore()), amount);
286            Aave(getAave()).repay(reserve, amount, address(uint160(address(this))));
287        }
```

Listing 3.20: iCollateralVaultProxy.sol

If successful, the `borrow()` call to the Aave protocol (line 278) will result in transferring the borrowed `ETH`s from the Aave `core` to the vault. However, the following call to transfer the borrowed `ETH`s from the vault to the borrower, i.e., `IERC20(reserve).safeTransfer(to, amount)` (line 279), becomes a `nop` because of the use of the above mock reserve address in `IERC20(reserve)`. In other words, the borrowed `ETH`s are stuck in the vault. It turns out even the vault owner is not able to retrieve them out, leading to borrowed `ETH` assets being locked forever. The `repay()` execution path shares the very same issue, resulting in repaid `ETH` assets being locked in the vault as well.

Recommendation Add the `ETH` support in the vault by extending the logic of both `borrow()` and `repay()`.

3.10 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, we always suggest using fixed compiler version whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.10;` instead of `pragma solidity ^0.6.10;`.

Moreover, we strongly suggest not to use experimental Solidity features (e.g., `pragma experimental ABIEncoderV2`) or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

Last but not least, 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 the mainnet.

4 Conclusion

In this audit, we thoroughly analyzed the CreditDelegationVault implementation. The proposed system for credit delegation presents a unique innovation and we are really impressed by the design and implementation. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, 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 [13, 14, 15, 16, 18].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [19] 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[5] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.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: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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

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

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

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

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

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

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

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