SMART CONTRACT AUDIT REPORT

for

ZERION

Prepared By: Shuxiao Wang

Hangzhou, China

Aug. 31, 2020

Document Properties

Client	Zerion
Title	Smart Contract Audit Report
Target	DeFi SDK (Core)
Version	1.0
Author	Huaguo Shi
Auditors	Huaguo Shi, Chiachih Wu, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	Aug. 31, 2020	Huaguo Shi	Final Release
1.0-rc2	Aug. 23, 2020	Xuxian Jiang	Release Candidate #2
1.0-rc1	Aug. 20, 2020	Jeff Liu	Release Candidate #1
0.1	Aug. 16, 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 **DeFi SDK (Core)** smart contracts' 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 DeFi SDK (Core)

Zerion DeFi SDK is designed to provide an unified, standardized interface to access or interact with a variety of Decentralized Finance or DeFi protocols (e.g., borrow crypto assets or invest to earn interests). The core contracts being audited here are part of Zerion DeFi SDK with the goal of enabling precise DeFi portfolio accounting and interaction. DeFi SDK can be regarded as the on-chain `balanceOf` for DeFi protocols, which consists of `Core`, `Adaptors`, and `InteractiveAdaptors` components.

The basic information of **Zerion DeFi SDK** is as follows:

Table 1.1: Basic Information of DeFi SDK (Core)

Item	Description
Vendor	Zerion
Website	https://zerion.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Aug. 31, 2020

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

- https://github.com/zeriontech/defi-sdk/tree/interactive/contracts/core (806d868)
- https://github.com/zeriontech/defi-sdk/tree/interactive/contracts/core (9ee397e)

1.2 About PeckShield

PeckShield Inc. [15] 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	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- 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) [9], 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 DeFi SDK (Core) contract 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	2	🟦🟦
Low	3	🟦🟦🟦
Informational	4	🟦🟦🟦🟦
Total	10	

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 vulnerability, 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 4 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incompatibility With Deflationary Tokens	Business Logics	Confirmed
PVE-002	Low	Locked Assets From Yield-Farming	Business Logics	Confirmed
PVE-003	Low	Better Handling of Ownership Transfers	Business Logics	Fixed
PVE-004	Info.	Inconsistency Between Component Struct Definition And Comments	Coding Practices	Fixed
PVE-005	Info.	Improved Type Checking of ActionType & AmountType	Coding Practices	Fixed
PVE-006	Medium	Possible Reentrancy Mitigation	Security Features	Fixed
PVE-007	Info.	Unused Import Removal	Coding Practices	Fixed
PVE-008	High	Possible Front-Running For Nonce Invalidation	Business Logics	Fixed
PVE-009	Low	Logic Error in toString(bytes32 data)	Coding Practices	Fixed
PVE-010	Info.	Undocumented Underflow Use in toString(bytes32 data)	Coding Practices	Fixed

Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incompatibility With Deflationary Tokens

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Router`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

The DeFi SDK (Core) contract acts as a trustless intermediary between users and various DeFi protocols. The intermediary allows users to access supported DeFi protocols with the unified, standardized interface. In particular, there is a `Router` contract that provides users a handy `startExecution()` function. Users can make use of this function to execute a set of `actions`, each representing either deposit or withdraw on a specified protocol.

If we zoom in this function, `startExecution()` firstly transfers all input assets from the calling user to `Router` (see `transferTokens()` in line 173 as shown in the following code snippet), and then executes these enclosed actions (via `executeActions()` in line 176).

```
161    function startExecution(
162        Action[] memory actions,
163        Input[] memory inputs,
164        Output[] memory requiredOutputs,
165        address payable account
166    )
167        internal
168        returns (Output[] memory)
169    {
170        // save initial gas to burn gas token later
171        uint256 gas = gasleft();
172        // transfer tokens to core_, handle fees (if any), and add these tokens to
               outputs
173        transferTokens(inputs, account);
174        Output[] memory modifiedOutputs = modifyOutputs(requiredOutputs, inputs);
```

```
        // call Core contract with all provided ETH, actions, expected outputs and
            account address
        Output[] memory actualOutputs = core_.executeActions{value: msg.value}(
            actions,
            modifiedOutputs,
            account
        );
        // burn gas token to save some gas
        freeGasToken(gas - gasleft());

        return actualOutputs;
    }
```

Listing 3.1: Router.sol

DeFi SDK (Core) defines the standard APIs to interact with various DeFi protocols. Specifically, both `deposit` and `withdraw` are being supported. Accordingly, current implementation provides their low-level asset-transferring routines. These routines work as expected with standard ERC20 tokens: namely the account's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts.

```
    /**
     * @notice Deposits tokens to the Aave protocol.
     * @param tokens Array with one element - underlying token address.
     * @param amounts Array with one element - underlying token amount to be deposited.
     * @param amountTypes Array with one element - amount type.
     * @return tokensToBeWithdrawn Array with ane element - aToken.
     * @dev Implementation of InteractiveAdapter function.
     */
    function deposit(
        address[] memory tokens,
        uint256[] memory amounts,
        AmountType[] memory amountTypes,
        bytes memory
    )
        public
        payable
        override
        returns (address[] memory tokensToBeWithdrawn)
    {
        require(tokens.length == 1, "AAIA: should be 1 token![1]");
        require(tokens.length == amounts.length, "AAIA: inconsistent arrays![1]");

        address pool = LendingPoolAddressesProvider(PROVIDER).getLendingPool();
        address core = LendingPoolAddressesProvider(PROVIDER).getLendingPoolCore();

        uint256 amount = getAbsoluteAmountDeposit(tokens[0], amounts[0], amountTypes[0])
            ;

        tokensToBeWithdrawn = new address[](1);
        tokensToBeWithdrawn[0] = LendingPoolCore(core).getReserveATokenAddress(tokens
            [0]);
        if (tokens[0] == ETH) {
            // solhint-disable-next-line no-empty-blocks
            try LendingPool(pool).deposit{value: amount}(ETH, amount, 0) {
            } catch Error(string memory reason) {
                revert(reason);
            } catch {
                revert("AAIA: deposit fail![1]");
            }
        } else {
            ERC20(tokens[0]).safeApprove(core, amount, "AAIA!");
            // solhint-disable-next-line no-empty-blocks
            try LendingPool(pool).deposit(tokens[0], amount, 0) {
            } catch Error(string memory reason) {
                revert(reason);
            } catch {
                revert("AAIA: deposit fail![2]");
            }
        }
    }
```

Listing 3.2: AaveAssetInteractiveAdapter.sol

However, in the cases of deflationary tokens, as shown in the above code snippets, the input amount may not be equal to the received amount due to the charged (and burned) transaction fee. 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 `withdraw`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts in the cases of deflationary tokens. Apparently, these balance inconsistencies are damaging to accurate portfolio management of DeFi SDK (Core) and affects protocol-wide operation and maintenance.

One mitigation is to query the asset change right before and after the asset-transferring routines. In other words, instead of automatically 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 the intended operation. Though these additional checks cost additional gas usage, we feel that 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 DeFi SDK (Core). However, as a trustless intermediary, DeFi SDK (Core) may not be in the position to effectively regulate the entire process.

Recommendation Add necessary mitigation mechanisms to keep track accurate balances if there is a need to support deflationary ERC20 tokens.

Status This issue has been confirmed. As it can be mitigated when the asset is being introduced, the team decided not to make a change for the time being, but will think of a future solution for it.

3.2 Locked Assets From Yield-Farming

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Core`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

DeFi SDK (Core) has already supported a number of known DeFi protocols, including `Maker`, `Compound`, and `Curve`. Some of these DeFi protocols have engaged so-called liquidity mining that may reward extra tokens to liquidity providers. Being the intermediary, DeFi SDK (Core) effectively acts as the "real" user when interacting with these protocols and correspondingly accumulates yields or farming rewards from these protocols.

However, these yields or farming rewards from liquidity mining may not credit to DeFi SDK (Core) itself. Instead, it should belong to actual users of DeFi SDK (Core). Therefore, besides the opportunity to receive all the income generated from interacting with underlying DeFi protocols, there is also a need for users to appropriately receive additional yields or rewards, such as `CRV`, `COMP`, etc.

In current implementation, these extra tokens of rewards are being locked in DeFi SDK (Core), and normal users cannot get their legitimate shares. With that, we feel the need to effectively redistribute these rewards tokens back to users.

Recommendation Develop an effective and fair approach to redistribute possible rewards, if any, back to true users.

Status This issue has been confirmed. The team decide that the functionality will be implemented in protocol adapters, which are upgradable for this purpose.

3.3 Better Handling of Ownership Transfers

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Ownable`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

The `_transferOwnership()` function in `Ownable` contract allows the current admin of the contract to transfer her privilege to another address. However, in the `_transferOwnership()` function, the `newOwner` is directly stored into the storage, `_owner`, after only validating that the `newOwner` is a non-zero address (line 70).

```
69      function _transferOwnership(address newOwner) internal {
70          require(newOwner != address(0), "Owner should not be 0 address");
71          emit OwnershipTransferred(_owner, newOwner);
72          _owner = newOwner;
73      }
```

Listing 3.3: Ownable.sol

This is reasonable under the assumption that the `newOwner` parameter is always correctly provided. However, in the unlikely situation, when an incorrect `newOwner` is provided, the contract owner may be forever lost, which might be devastating for DeFi SDK (Core) operation and maintenance.

As a common best practice, instead of achieving the owner update within a single transaction, it is suggested to split the operation into two steps. The first step initiates the owner update intent and the second step accepts and materializes the update. Both steps should be executed in two separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract ownership to an uncontrolled address. In other words, this two-step procedure ensures that an owner public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the owner transfer process.

Recommendation As suggested, the ownership transition can be better managed with a two-step approach, such as, using these two functions: `_transferOwnership()` and `_updateOwnership()`. Specifically, the `_transferOwnership()` function keeps the new address in the storage, `_newOwner`, instead of modifying the `_owner` directly. The `updateOwnership()` function checks whether `_newOwner` is `msg.sender` to ensure that `_newOwner` signs the transaction and verifies herself as the new owner. Only after the successful verification, `_newOwner` would effectively become the `_owner`.

```
69      function _transferOwnership(address newOwner) internal {
70          require(newOwner != address(0), "Owner should not be 0 address");
71          require(newOwner != _owner, "The current and new owner cannot be the same");
72          require(newOwner != _newOwner, "Cannot set the candidate owner to the same
                address");
73          _newOwner = newOwner;
74      }

76      function _updateOwnership() public {
77          require(_newOwner != address(0), "Candidate owner had not been set");
78          require(msg.sender == _newOwner, "msg.sender and _newOwner must be the same");
79          _owner = _newOwner;
80          emit OwnershipTransferred(_owner, _newOwner);
```

```
    }
```

Listing 3.4: Ownable.sol (revised)

Status This issue has been fixed by this particular commit: 806d868d55e4f39c35eb161d027799511bf121df.

3.4 Inconsistency Between Component Struct Definition And Comments

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `TokenAdapter`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [2]

Description

In the `Structs.sol` file, a structure named `Component` is defined with two members, `token` and `rate` (as shown in the code snippet below).

```
    // The struct consists of token address,
    // and price per full share (1e18).
    struct Component {
        address token;
        uint256 rate;
    }
```

Listing 3.5: Structs . sol

Meanwhile, we notice in the `TokenAdapter.sol` file, the `Component` is commented (inside `getComponents()` – lines 36 – 39) to have a definition with three member fields: `token`, `tokenType`, and `rate`. The field `tokenType` is not present yet.

```
34      /**
35       * @dev MUST return array of Component structs with underlying tokens rates for the
             given token.
36       * struct Component {
37       *     address token;    // Address of token contract
38       *     bytes32 tokenType; // Token type ("ERC20" by default)
39       *     uint256 rate;     // Price per share (1e18)
40       * }
41       */
42      function getComponents(address token) external view virtual returns (Component[]
            memory);
```

Listing 3.6: TokenAdapter.sol

Recommendation Make the `Component` definition consistent by revising the above-mentioned comment section.

Status This issue has been fixed by this particular commit: 806d868d55e4f39c35eb161d027799511bf121df.

3.5 Improved Type Checking of ActionType & AmountType

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Core`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [2]

Description

In the `Core` contract, the internal function, `executeAction()`, is used to execute an `action`. Before executing the `action`, the `action.actionType` is checked against `ActionType.None` and `ActionType.Deposit` in line 115 and line 118, respectively. However, the `action.actionType` has three possible values: `None`, `Deposit`, `Withdraw`. Therefore, if an abnormal `actionType` value is used, such as 3, or a bigger value, the current code would result to a `Withdraw` operation. Same issue applies to `AmountType` and related checks.

```
107     function executeAction(
108         Action calldata action
109     )
110         internal
111         returns (address[] memory)
112     {
113         address adapter = adapterRegistry_.getProtocolAdapterAddress(action.
                protocolAdapterName);
114         require(adapter != address(0), "C: bad name!");
115         require(action.actionType != ActionType.None, "C: bad action type!");
116         require(action.amounts.length == action.amountTypes.length, "C: inconsistent
                arrays!");
117         bytes4 selector;
118         if (action.actionType == ActionType.Deposit) {
119             selector = InteractiveAdapter(adapter).deposit.selector;
120         } else {
121             selector = InteractiveAdapter(adapter).withdraw.selector;
122         }
```

Listing 3.7: Core.sol

Recommendation Directly check if the parameter value matches `Deposit` or `Withdraw`.

```
    function executeAction(
        Action calldata action
    )
        internal
        returns (address[] memory)
    {
        address adapter = adapterRegistry_.getProtocolAdapterAddress(action.
            protocolAdapterName);
        require(adapter != address(0), "C: bad name!");
        require(action.actionType == ActionType.Deposit || action.actionType ==
            ActionType.Withdraw, "C: bad action type!");
        require(action.amounts.length == action.amountTypes.length, "C: inconsistent
            arrays!");
        bytes4 selector;
        if (action.actionType == ActionType.Deposit) {
            selector = InteractiveAdapter(adapter).deposit.selector;
        } else {
            selector = InteractiveAdapter(adapter).withdraw.selector;
        }
```

Listing 3.8: Core.sol (revised)

Status This issue has been fixed by this particular commit: 806d868d55e4f39c35eb161d027799511bf121df.

3.6 Possible Reentrancy Mitigation

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Router`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `Router` contract, the `startExecution()` function allows users to perform a set of actions, i.e., `deposit` and `withdraw`, via the `InteractiveAdapter` adapters for each individual DeFi protocol. We notice that the `startExecution()` entrance is not protected with `nonReentrant` and may be exposed with the undesirable reentrancy risk.

```
    function startExecution(
        Action[] memory actions,
        Input[] memory inputs,
        Output[] memory requiredOutputs,
        address payable account
    )
        internal
```

```
168         returns (Output[] memory)
169     {
170         ...
171     }
```

Listing 3.9: Router.sol

Using the `Aave` adapter as an example (shown in the code snippets below), we notice the external call in line 162, i.e., `AToken(tokens[0]).redeem(amount)`. If we assume certain ERC777-based tokens are being supported, it is possible for a malicious actor to potentially hijack the execution flow and allow for reentrancy via the same function. Meanwhile, we emphasize that current reentrancy does not cause any damage yet. Considering its notorious history, we highly suggest to mitigate or completely prevent the reentrancy risk in the first place.

```
134     /**
135      * @notice Withdraws tokens from the Aave protocol.
136      * @param tokens Array with one element - aToken address.
137      * @param amounts Array with one element - aToken amount to be withdrawn.
138      * @param amountTypes Array with one element - amount type.
139      * @return tokensToBeWithdrawn Array with one element - underlying token.
140      * @dev Implementation of InteractiveAdapter function.
141      */
142     function withdraw(
143         address[] memory tokens,
144         uint256[] memory amounts,
145         AmountType[] memory amountTypes,
146         bytes memory
147     )
148         public
149         payable
150         override
151         returns (address[] memory tokensToBeWithdrawn)
152     {
153         require(tokens.length == 1, "AAIA: should be 1 token![2]");
154         require(tokens.length == amounts.length, "AAIA: inconsistent arrays![2]");

156         uint256 amount = getAbsoluteAmountWithdraw(tokens[0], amounts[0], amountTypes
                [0]);

158         tokensToBeWithdrawn = new address[](1);
159         tokensToBeWithdrawn[0] = AToken(tokens[0]).underlyingAssetAddress();

161         // solhint-disable-next-line no-empty-blocks
162         try AToken(tokens[0]).redeem(amount) {
163         } catch Error(string memory reason) {
164             revert(reason);
165         } catch {
166             revert("AAIA: withdraw fail!");
167         }
168     }
```

Listing 3.10: AaveAssetInteractiveAdapter.sol

Recommendation Add the reentrancy-mitigating mechanism, such as integrating the ReentrancyGuard support from OpenZeppelin.

Status This issue has been fixed by this particular commit: 806d868d55e4f39c35eb161d027799511bf121df.

3.7 Unused Import Removal

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: TokenAdapterManager
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [2]

Description

In the TokenAdapterManager contract, there is an import TokenAdapter from ../adapters/TokenAdapter .sol. It turns out that this import is not being used and can thus be safely removed.

```
pragma solidity 0.6.11;
pragma experimental ABIEncoderV2;

import { Ownable } from "./Ownable.sol";
import { TokenAdapter } from "../adapters/TokenAdapter.sol";
```

Listing 3.11: TokenAdapterManager.sol

Recommendation Remove the unused import TokenAdapter in TokenAdapterManager.

Status This issue has been fixed by this particular commit: 806d868d55e4f39c35eb161d027799511bf121df.

3.8 Possible Front-Running For Nonce Invalidation

- ID: PVE-008
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: Router
- Category: Business Logics [8]
- CWE subcategory: CWE-754 [4]

Description

The Router contract provides a startExecution() function variant that allows users to authorize actions using an off-chain signature. The intention is to support meta-transactions such that an user can simply sign an intended transaction offline and then send the signed transaction to a relayer. The replayer will take care of submitting the transaction for mining by paying required transaction fee. The signature correctness will be verified on chain via a helper routine, i.e., `getAccountFromSignature()`.

In the following, we elaborate the related execution logic. Particularly, the `startExecution()` function verifies the signature using the `getAccountFromSignature()` helper (line 140) before executing the enclosed actions.

```
128     function startExecution(
129         TransactionData memory data,
130         bytes memory signature
131     )
132         public
133         payable
134         returns (Output[] memory)
135     {
136         return startExecution(
137             data.actions,
138             data.inputs,
139             data.requiredOutputs,
140             getAccountFromSignature(data, signature)
141         );
142     }
```

Listing 3.12: Router.sol

Within the `getAccountFromSignature()` helper, we observe that it is declared as `public` and it takes the normal procedure, i.e., `ecrecover()`, to retrieve the signer information. After that, it then checks the nonce freshness, i.e., whether `nonce_[signer]` is valid (lines 141 – 143). If yes, it advances the nonce by 1, i.e., `nonce_[signer]++`. Since this function is defined as `public`, any one could call this function to verify the signature but with the side-effect of advancing the nonce (if successfully verified)!

Here comes the problem: when an user invokes `startExecution()` to perform specified actions by signing the transaction offline, but before the transaction is mined, it is possible for a malicious actor to observe it (by closely monitoring the transaction pool) and then possibly front-runs it by crafting a new transaction and offering a higher gas fee for block inclusion. The new transaction may perform a fresh `getAccountFromSignature()` call. If the front-running is successful, the crafted transaction essentially advances the `nonce` by 1, effectively invalidating the user transaction that is being front-run.

```
107     function getAccountFromSignature(
108         TransactionData memory data,
109         bytes memory signature
110     )
111         public
112         returns (address payable)
113     {
        require(signature.length == 65, "SV: bad sig length!");
        bytes32 r;
        bytes32 s;
        uint8 v;

        // solium-disable-next-line no-inline-assembly
        // solhint-disable-next-line no-inline-assembly
        assembly {
            r := mload(add(signature, 0x20))
            s := mload(add(signature, 0x40))
            v := byte(0, mload(add(signature, 0x60)))
        }

        address signer = ecrecover(
            keccak256(
                abi.encodePacked(
                    bytes1(0x19),
                    bytes1(0x01),
                    domainSeparator_,
                    hash(data)
                )
            ),
            v,
            r,
            s
        );

        require(nonce_[signer] == data.nonce, "SV: bad nonce!");

        nonce_[signer]++;

        return payable(signer);
    }
```

Listing 3.13: SignatureVerifier.sol

Recommendation Make `getAccountFromSignature()` internal.

Status This issue has been fixed by this particular commit: 806d868d55e4f39c35eb161d027799511bf121df.

3.9 Logic Error in toString(bytes32 data)

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Helpers`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [2]

Description

In the `Helper` library, an internal function, `toString(bytes32 data)`, is used to convert `bytes32` to `string` and trim the `zeroes` in it. However, there is a logic issue in the function. Specifically, in lines $35-39$, the `length` parameter keeps track of the total number of `non-zero` characters (`zero` characters are ignored). But in lines $43-47$, the copying operation does NOT check for `non-zero` characters. As a result, suppose there is a `zero` character in the middle of a given `byte32` data and there is only one `zero` character, the `length` would equal to $total\_number\_of\_characters-1$. But after the copying operation, the last character would be dropped.

```
28    /**
29     * @dev Internal function to convert bytes32 to string and trim zeroes.
30     */
31    function toString(bytes32 data) internal pure returns (string memory) {
32        uint256 length = 0;
33        bytes memory result;

35        for (uint256 i = 0; i < 32; i++) {
36            if (data[i] != bytes1(0)) {
37                length++;
38            }
39        }

41        result = new bytes(length);

43        for (uint256 i = 0; i < length; i++) {
44            result[i] = data[i];
45        }

47        return string(result);
48    }
```

Listing 3.14: helper.sol

Recommendation Modify the code, in lines $43-47$, to copy only `non-zero` characters.

```
28    /**
29     * @dev Internal function to convert bytes32 to string and trim zeroes.
30     */
31    function toString(bytes32 data) internal pure returns (string memory) {
32        uint256 length = 0;
33        bytes memory result;

35        for (uint256 i = 0; i < 32; i++) {
36            if (data[i] != bytes1(0)) {
37                length++;
38            }
39        }

41        result = new bytes(length);
```

```
        uint256 j = 0;
        for (uint256 i = 0; i < 32; i++) {
            if (data[i] != bytes1(0)) {
                result[j++] = data[i];
            }
        }

        return string(result);
    }
```

Listing 3.15: helper.sol (revised)

Status This issue has been fixed by this particular commit: 4bd07126193a264a94d9e616eb55a4d026c47322.

3.10 Undocumented Underflow Use in toString(uint256 data)

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Helpers`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [2]

Description

In the `Helper` library, there exists another `toString()` variant. Instead of converting `bytes32` to `string` (Section 3.9), this variant i.e., `toString(uint256 data)`, converts `uint256` to `string`. As shown in the code snippets below, we observe an internal `for` loop (lines 65 – 68) that is used to copy characters to `result`. This `for` loop is interesting and deserves further elaboration.

At the first glance, the `for` loop iterates with an internal `unsigned` variable `i`. By `i--`, the internal variable `i` would be always less than `length`, leading to a concern that it may cause an infinite loop. A further examination indicates that the variable `i` is defined as `unsigned` with the type of `uint256`. As a result, continuous `i--` operations would eventually cause an underflow and make `i` a huge value, thus breaking out of the internal `for` loop. Although the logic itself is indeed correct, it takes an unconventional way to purposely using an underflow. For improved code understanding and maintenance, we feel that additional comments here would be helpful.

```
50      /**
51       * @dev Internal function to convert uint256 to string.
52       */
53      function toString(uint256 data) internal pure returns (string memory) {
54          uint256 length = 0;

56          uint256 dataCopy = data;
```

```
        while (dataCopy != 0) {
            length++;
            dataCopy /= 10;

        }

        bytes memory result = new bytes(length);
        dataCopy = data;
        for (uint256 i = length - 1; i < length; i--) {
            result[i] = bytes1(uint8(48 + dataCopy % 10));
            dataCopy /= 10;
        }

        return string(result);
    }
```

Listing 3.16: Helpers.sol

Another alternative is to take a conventional approach by defining `i` as a `signed` integer. We can still simply terminate the `for` loop by ensuring `i >= 0`.

Recommendation Document the purposeful underflow or take a more conventional approach as suggested above.

Status This issue has been fixed by this particular commit: 806d868d55e4f39c35eb161d027799511bf121df.

3.11 Other Suggestions

As a common suggestion, 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.

4 Conclusion

In this audit, we thoroughly analyzed the DeFi SDK (Core) documentation and implementation. The audited system presents a unique, much-needed innovation by defining a unified, standardized interface to access a variety of Decentralized Finance (or DeFi) protocols. We are really impressed by the clean 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 [11, 12, 13, 14, 16].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [17] 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-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

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

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

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

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

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

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

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

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

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

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

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

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

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