SMART CONTRACT AUDIT REPORT

for

ADD.XYZ

Prepared By: Shuxiao Wang

Hangzhou, China

Dec. 17, 2020

Document Properties

Client	ADD.xyz
Title	Smart Contract Audit Report
Target	ADD.xyz V1
Version	1.0
Author	Xudong Shao
Auditors	Xudong Shao, Chiachih Wu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	Dec. 17, 2020	Xudong Shao	Final Release
1.0-rc	Dec. 5, 2020	Xudong Shao	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 **ADD.xyz V1 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 ADD.xyz V1 Protocol

ADD.xyz V1 is a full-stack DeFi aggregator, plugging in multiple products and DeFi applications into one single platform, focusing on user experience, design, privacy and anonymity. Products offered by ADD.xyz include aggregated DeFi lending, insurance & privacy, fiat to crypto savings bridge, DeFi debit cards, and DeFi-as-a-Service (SDK) for exchanges, etc.

The basic information of ADD.xyz V1 is as follows:

Table 1.1: Basic Information of ADD.xyz V1

Item	Description
Issuer	ADD.xyz
Website	https://add.xyz/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Dec. 17, 2020

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

- https://github.com/PlutusDefi/plutus-smart-contracts-audit (4122686)

And here is the list of products and contracts being audited:

Product: **Lending System**:
Contracts included:

* ModuleRegistry.sol: Keeps track of system modules for lending.
* Deployer.sol: Deploys smart contract wallet for users.
* BaseWallet.sol: User smart wallet contract to handle user funds.
* CompoundRegistry.sol: Handle compound protocol market information.
* CompoundInvest: Handle deposit and withdraw logic for compound protocol for users.
* DYDXRegistry.sol: Handle dYdX protocol market information.
* DYDXInvest: Handle deposit and withdraw logic for dYdX protocol for users.
* FulcrumRegistry.sol: Handle fulcrum protocol market information.
* FulcrumInvest: Handle deposit and withdraw logic for fulcrum protocol for users.
* AaveRegistry.sol: Handle Aave protocol market information.
* AaveInvest: Handle deposit and withdraw logic for Aave protocol for users.
* YearnRegistry.sol: Handle yEarn protocol market information.
* YearnInvest: Handle deposit and withdraw logic for yEarn protocol for users.

Product: **Blender**:
Contracts included:

* ERC20Tornado.sol
* ETHTornado.sol
* MerkleTreeWithHistory.sol
* Migrations.sol
* Tornado.sol

1.2 About PeckShield

PeckShield Inc. [14] 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 [9]:

- 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 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

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

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) [8], 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 ADD.xyz V1 Protocol design and implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	7	■■■■■■■
Informational	3	■■■
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 7 low-severity vulnerabilities and 3 informational recommendations.

Table 2.1: Key ADD.xyz V1 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Re-Initialization Risks in BaseWallet::init()	Business Logic	Confirmed
PVE-002	Low	Re-Initialization and DoS Risks in BaseWallet::authoriseModule()	Business Logic	Fixed
PVE-003	Low	Missed Sanity Check in BaseWallet::enableStaticCall()	Coding Practices	Fixed
PVE-004	Low	Unsafe Ownership Transition in Owned	Coding Practices	Fixed
PVE-005	Info.	Unused Events and Interfaces	Coding Practices	Fixed
PVE-006	Info.	Integer Overflow in AAVEInvest::withdrawInvestment()	Coding Practices	Fixed
PVE-007	Low	Missed Owner Fee Collection in AAVEInvest::withdrawEntireInvestment()	Business Logic	Fixed
PVE-008	Low	Wrong Token Amount Burned in AAVEInvest::withdrawEntireInvestment()	Coding Practices	Fixed
PVE-009	Info.	Unsafe ERC20 transfer() Calls	Business Logic	Fixed
PVE-010	Low	Over-Privileged Operator in Tornado	Security Features	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Re-Initialization Risks in BaseWallet::init()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: BaseWallet
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In ADD.xyz V1, the BaseWallet contract delegates calls to the authorized DeFi modules such as AAVE, Compound, etc. To achieve that, the authorised[] mapping is used to book-keep the authorised modules. Besides, the public integer variable, modules, is used to keep the count of authorized modules. As shown in the code snippet below, the init() function set the input _modules.length to modules in line 44. Later on, the for-loop walks through the _modules array and invokes the init() handler of each previously unauthorized module in line 48.

```
37     function init(
38         address[] calldata _modules
39     )
40         external
41         onlyOwner
42     {
43         require(_modules.length > 0, "BW: construction requires at least 1 module");
44         modules = _modules.length;
45         for(uint256 i = 0; i < _modules.length; i++) {
46             require(authorised[_modules[i]] == false, "BW: module is already added");
47             authorised[_modules[i]] = true;
48             Module(_modules[i]).init(this);
49             emit AuthorisedModule(_modules[i], true);
50         }
51     }
```

Listing 3.1: BaseWallet.sol

However, it comes to our attention that the owner can call `init()` more than once. As the function name suggests, `init()` should not be called more than once. In addition, if the owner does this by mistake, the `modules` variable would be overwritten, which makes the modules count inaccurate.

Recommendation Ensure the `BaseWallet` contract could only be initialized once by checking/setting the `initialized` flag as follows:

```
function init(
    address[] calldata _modules
)
    external
    onlyOwner
{
    require(_modules.length > 0, "BW: construction requires at least 1 module");
    require(!initialized);
    initialized = true;
    modules = _modules.length;
    for(uint256 i = 0; i < _modules.length; i++) {
        require(authorised[_modules[i]] == false, "BW: module is already added");
        authorised[_modules[i]] = true;
        Module(_modules[i]).init(this);
        emit AuthorisedModule(_modules[i], true);
    }
}
```

Listing 3.2: BaseWallet.sol

Status This issue has been confirmed by the team.

3.2 Re-Initialization and DoS Risks in BaseWallet::authoriseModule()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `BaseWallet`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As we introduced in Section 3.1, ADD.xyz V1 provides different investment modules which are authorized by the `BaseWallet::init()` function. Except the `init()` in `BaseWallet` contract, the `BaseWallet::authoriseModule()` function also allows a privileged caller to enable/disable a module.

```
function authoriseModule(
    address _module,
```

```
        bool _value
    )
        external
        moduleOnly
    {
        if (authorised[_module] != _value) {
            if(_value == true) {
                modules += 1;
                authorised[_module] = true;
                Module(_module).init(this);
            }
            else {
                modules -= 1;
                require(modules > 0, "BW: wallet must have at least one module");
                delete authorised[_module];
            }
            emit AuthorisedModule(_module, _value);
        }
    }
```

Listing 3.3: BaseWallet.sol

However, if the user wants to authorise a module that has already been initialized, the states of that module would be reset. Furthermore, the current implementation allows one authorised module to enable/disable another module. This allows a malicious/compromised module to disable all the modules of the wallet and launch a Denial-of-Service (DoS) attack.

Recommendation Add a flag to indicate if this module needs to be initialized; Make sure only the owner can call this function.

```
    function authoriseModule(
        address _module,
        bool _value,
        bool _init
    )
        external
        onlyOwner
    {
        if (authorised[_module] != _value) {
            if(_value == true) {
                modules += 1;
                authorised[_module] = true;
                if(_init){
                  Module(_module).init(this);
                }
            }
            else {
                modules -= 1;
                require(modules > 0, "BW: wallet must have at least one module");
                delete authorised[_module];
            }
```

```
            emit AuthorisedModule(_module, _value);
        }
    }
```

Listing 3.4: BaseWallet.sol

Status This issue has been fixed in the commit: f63d15006ea53e8586581936efbdb4ffd10e8401.

3.3 Missed Sanity Check in BaseWallet::enableStaticCall()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `BaseWallet`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [2]

Description

As mentioned in Section 3.1, the `BaseWallet` contract delegates calls to authorised DeFi modules. The delegation is actually done by routing the function calls based on the `enabled[]` mapping to the destination module addresses. Specifically, as shown in the code snippet below, if the `msg.sig` points to a non-zero module address in the routing table (i.e., `enabled[]` mapping), `staticcall` is executed (line 163) if that module address is authorised (line 159).

```
152 function() external payable {
153     if(msg.data.length > 0) {
154         address module = enabled[msg.sig];
155         if(module == address(0)) {
156             emit Received(msg.value, msg.sender, msg.data);
157         }
158         else {
159             require(authorised[module], "BW: must be an authorised module for static
                    call");
160             // solium-disable-next-line security/no-inline-assembly
161             assembly {
162                 calldatacopy(0, 0, calldatasize())
163                 let result := staticcall(gas, module, 0, calldatasize(), 0, 0)
164                 returndatacopy(0, 0, returndatasize())
165                 switch result
166                 case 0 {revert(0, returndatasize())}
167                 default {return (0, returndatasize())}
168             }
169         }
170     }
171 }
```

Listing 3.5: BaseWallet.sol

To configure the routing table, the `enableStaticCall()` function allows an authorized module to set the path to a specific `_method` as the address `_module`.

```
function enableStaticCall(
    address _module,
    bytes4 _method
)
    external
    moduleOnly
{
    require(authorised[_module], "BW: must be an authorised module for static call");
    enabled[_method] = _module;
    emit EnabledStaticCall(_module, _method);
}
```

Listing 3.6: BaseWallet.sol

However, the `enableStaticCall()` function fails to check the duplicate `_method` such that another authorized `msg.sender` could always overwrite the routing table. This allows a malicious/compromised authorized `msg.sender` to overwrite the routing table to route other methods' calls to its hook function.

Recommendation Prevent enabled method from being overwritten.

```
function enableStaticCall(
    address _module,
    bytes4 _method
)
    external
    moduleOnly
{
    require(authorised[_module], "BW: must be an authorised module for static call");
    require(enabled[_method] == address(0), "the method has been enabled");
    enabled[_method] = _module;
    emit EnabledStaticCall(_module, _method);
}
```

Listing 3.7: BaseWallet.sol

Status This issue has been fixed in the commit: f63d15006ea53e8586581936efbdb4ffd10e8401.

3.4 Unsafe Ownership Transition

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Owned`, `AAVEInvest`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In ADD.xyz V1, the `Owned` contract is widely used for ownership management in many contracts such as `BaseWallet`, `AAVERegistry`, etc. When the contract owner needs to transfer the ownership to another address, she could invoke the `changeOwner()` function with a `_newOwner` address.

```
function changeOwner(
    address _newOwner
)
    external
    onlyOwner
{
    require(_newOwner != address(0), "Owned: Address must not be null");
    owner = _newOwner;
    emit OwnerChanged(_newOwner);
}
```

Listing 3.8: Owned.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. Same logic applies to the `changeFeeOwner()` function in the `AAVEInvest` contract.

```
function changeFeeOwner(
    address _newFeeOwner
)
    external
{
    require(_newFeeOwner != address(0), "Owned: Address must not be null");
    require(msg.sender == address(FEE_OWNER), "Owned: Caller must be Fee Owner");
    emit OwnerChanged(FEE_OWNER, _newFeeOwner);
    FEE_OWNER = _newFeeOwner;
}
```

Listing 3.9: AAVEInvest.sol

Recommendation Implement a two-step ownership transfer mechanism that allows the new owner to claim the ownership by signing a transaction.

```
function changeOwner(
    address _newOwner
)
    external
    onlyOwner
{
    require(_newOwner != address(0), "Owned: Address must not be null");
    require(candidateOwner != _newOwner, "Owned: Same candidate owner");
    candidateOwner = _newOwner;
}

function claimOwner()
```

```
    external
{
    require(candidateOwner == msg.sender, "Owned: Claim ownership failed");
    owner = candidateOwner;
    emit OwnerChanged(candidateOwner);
}
```

Listing 3.10: Owned.sol

Status This issue has been fixed in the commit: f63d15006ea53e8586581936efbdb4ffd10e8401.

3.5 Unused Events and Interfaces

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BaseWallet`, `AAVEInvest`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [2]

Description

In the `BaseWallet` contract, there is an unused event, `OwnerChanged`, which could be safely removed.

```
    event OwnerChanged(address owner);
```

Listing 3.11: BaseWallet.sol

Besides the unused event, there are some unused interfaces in the `AAVEInvest` contract.

Case I `getAssetsPrices()`, `getSourceOfAsset()`, `getFallbackOracle()` in line $12-14$.

```
10 interface IPriceOracleGetter {
11     function getAssetPrice(address _asset) external view returns (uint256);
12     function getAssetsPrices(address[] calldata _assets) external view returns(uint256[]
           memory);
13     function getSourceOfAsset(address _asset) external view returns(address);
14     function getFallbackOracle() external view returns(address);
15 }
```

Listing 3.12: AAVEInvest.sol

Case II `getReserves()` in line 24.

```
23 interface LendingPool{
24     function getReserves() external view returns (address[] memory);
25 }
```

Listing 3.13: AAVEInvest.sol

Recommendation Remove the unused events and interfaces.

Status This issue has been fixed in the commit: cbd67fbc8215741e7a7106052927f5f85eab5049.

3.6 Integer Overflow in AAVEInvest::withdrawInvestment()

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AAVEInvest`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [2]

Description

In ADD.xyz V1, the `AAVEInvest` contract allows wallet owners to interact with AAVE. As a common use case, the wallet owner is allowed to invest tokens with the `addInvestment()` function. On the other hand, the `withdrawInvestment()` function allows the wallet owner to withdraw a specific `_amount` of `_token`. As shown in the code snippet below, `FEE_PERCENT` of `_amount` is taken and transferred to `FEE_OWNER` in line 164.

```
146     function withdrawInvestment(
147         BaseWallet _wallet,
148         address _token,
149         uint256 _amount
150     )
151         external
152         onlyWalletOwner(_wallet)
153     {
154         //Getting the underlying protocol token
155         address Token = aaveRegistry.getToken(_token);
156
157         //Calculating owner fee wrt fee percent value
158         uint256 ownerWithdrawalFeeAmount = (_amount * FEE_PERCENT) /
159             ((100 * FEE_PERCENT_PRECESION));
160         burn(_wallet, Token, _amount);
161         emit InvestmentRemoved(address(_wallet), _token, _amount);
162
163         //sending fee amount to plutus onwer
164         _wallet.invoke(_token, 0, abi.encodeWithSignature("transfer(address,uint256)",
                FEE_OWNER, ownerWithdrawalFeeAmount));
```

Listing 3.14: AAVEInvest.sol

However, `SafeMath` is not used for `_amount * FEE_PERCENT`, leading to `ownerWithdrawalFeeAmount = 0` if the multiplication overflows. Fortunately, the `burn()` call in line 160 would prevent this loophole from being exploited. But we still suggest using `SafeMath` for multiplication here.

Recommendation Use SafeMath for the multiplication.

```
function withdrawInvestment(
    BaseWallet _wallet,
    address _token,
    uint256 _amount
)
    external
    onlyWalletOwner(_wallet)
{
    //Getting the underlying protocol token
    address Token = aaveRegistry.getToken(_token);

    //Calculating owner fee wrt fee percent value
    uint256 ownerWithdrawalFeeAmount = _amount.mul(FEE_PERCENT).div
        (100.mul(FEE_PERCENT_PRECESION));
    burn(_wallet, Token, _amount);
```

Listing 3.15: AAVEInvest.sol

Status This issue has been fixed in the commit: f63d15006ea53e8586581936efbdb4ffd10e8401.

3.7 Missed Owner Fee Collection in AAVEInvest::withdrawEntireInvestment()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: AAVEInvest
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As described in Section 3.6, the withdrawInvestment() function in the AAVEInvest contract could be used by the wallet owner to withdraw a specific _amount of _token with fee paid to the FEE_OWNER. There's another function, withdrawEntireInvestment(), in the AAVEInvest contract which allows wallet owner to withdraw all of the _token withheld by _wallet. However, the current implementation fails to pay fee to the FEE_OWNER, which makes it incompatible to withdrawInvestment().

```
function withdrawEntireInvestment(
    BaseWallet _wallet,
    address _token
)
    external
    onlyWalletOwner(_wallet)
{
```

```
        address Token = aaveRegistry.getToken(_token);
        uint256 balance = currentBalance(_wallet, _token, Token);
        burn(_wallet, Token, balance);
        emit InvestmentRemoved(address(_wallet), _token, balance);
    }
```

Listing 3.16: AAVEInvest.sol

Therefore, wallet owners could always use the `withdrawEntireInvestment()` function to avoid paying fee while withdrawing invested positions.

Recommendation Charge the owner fee when users withdraw entire investment.

```
    function withdrawEntireInvestment(
        BaseWallet _wallet,
        address _token
    )
        external
        onlyWalletOwner(_wallet)
    {
        address Token = aaveRegistry.getToken(_token);
        uint256 balance = currentBalance(_wallet, _token, Token);
        uint256 ownerWithdrawalFeeAmount = balance.mul(FEE_PERCENT).div
            (100.mul(FEE_PERCENT_PRECESION));
        burn(_wallet, Token, balance);
        emit InvestmentRemoved(address(_wallet), _token, balance);

        //sending fee amount to plutus onwer
        _wallet.invoke(_token, 0, abi.encodeWithSignature("transfer(address,uint256)",
            FEE_OWNER, ownerWithdrawalFeeAmount));
        emit FeeDeducted(ownerWithdrawalFeeAmount, address(_wallet), FEE_OWNER);
    }
```

Listing 3.17: AAVEInvest.sol

Status This issue has been fixed in the commit: cbd67fbc8215741e7a7106052927f5f85eab5049.

3.8 Wrong Token Amount Burned in AAVEInvest::withdrawEntireInvestment()

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `IHegicOptions`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [2]

Description

As mentioned in Section 3.7, the `withdrawEntireInvestment()` function allows wallet owner to with all `_token` withheld by `_wallet`. However, the current implementation is not compatible to that design. Specifically, as shown in the code snippet, line 135 gets `balance` with the `currentBalance()` function. Later on, the `balance` is used as the amount to redeem the underlying assets.

```
127     function withdrawEntireInvestment(
128         BaseWallet _wallet,
129         address _token
130     )
131         external
132         onlyWalletOwner(_wallet)
133     {
134         address Token = aaveRegistry.getToken(_token);
135         uint256 balance = currentBalance(_wallet, _token, Token);
136         burn(_wallet, Token, balance);
137         emit InvestmentRemoved(address(_wallet), _token, balance);
138     }
```

Listing 3.18: AAVEInvest.sol

Inside `currentBalance()` function, the `amount` retrieved with `balanceOf()` in line 202 is converted to the equivalent price in line 206, which is not necessary for burning/redeeming the entire balance.

```
193     function currentBalance(
194         BaseWallet _wallet,
195         address token,
196         address _Token
197     )
198         internal
199         view
200         returns (uint256 _balance)
201     {
202         uint amount = Token(_Token).balanceOf(address(_wallet));
203         LendingPoolAddressesProvider provider = LendingPoolAddressesProvider(
                lending_pool_provider);
204         IPriceOracleGetter priceOracle = IPriceOracleGetter(provider.getPriceOracle());
205         uint256 price = priceOracle.getAssetPrice(token);
206         _balance = amount.mul(price).div(10 ** 18);
207     }
```

Listing 3.19: AAVEInvest.sol

As a result, `withdrawEntireInvestment()` invokes `burn()` with a wrong `_token` amount, leading to a business logic error.

Recommendation Burn the exact amount of tokens in `withdrawEntireInvestment()`.

```
127     function withdrawEntireInvestment(
128         BaseWallet _wallet,
129         address _token
```

```
    )
        external
        onlyWalletOwner(_wallet)
    {
        address Token = aaveRegistry.getToken(_token);
        uint amount = Token(_Token).balanceOf(address(_wallet));
        burn(_wallet, Token, amount);
        emit InvestmentRemoved(address(_wallet), _token, balance);
    }
```

Listing 3.20: AAVEInvest.sol

Status This issue has been fixed in the commit: cbd67fbc8215741e7a7106052927f5f85eab5049.

3.9 Unsafe ERC20 transfer() Calls

- ID: PVE-009
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BaseWallet`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `BaseWallet` contract, the `withdrawToOwner()` function allows privileged users (i.e., passing `moduleOnly()` check) to withdraw `_amount` of `_token` to the `owner`. The token transfers are conducted via ERC20-compatible transfer() calls. However, while calling `IERC20(_token).transfer()`, the `BaseWallet` contract fails to check the return value as shown in line 64 below.

```
57    function withdrawToOwner(
58        address _token,
59        uint256 _amount
60        )
61        external
62        moduleOnly
63        {
64        IERC20(_token).transfer(owner, _amount);
65        emit WithdrawToOwner(owner, _token, _amount);
66    }
```

Listing 3.21: BaseWallet.sol

Similar to the above case, in `BaseModule::recoverToken()`, the return value of `IERC20().transfer()` is also ignored.

```
    function recoverToken(address _token)
        external
    {
        uint total = IERC20(_token).balanceOf(address(this));
        IERC20(_token).transfer(address(registry), total);
    }
```

Listing 3.22: BaseModule.sol

When the _token contract fails to revert for whatever reason, the caller of transfer() functions cannot ensure if the tokens are transferred successfully. In addition, certain ERC20 token contracts do not have a return value in its transfer() 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 fixed in the commit: cbd67fbc8215741e7a7106052927f5f85eab5049.

3.10 Over-Privileged Operator in Tornado

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Tornado
- Category: Security Features [5]
- CWE subcategory: CWE-269 [3]

Description

The ADD.xyz V1 system integrates the Tornado contract to protect private transactions from prying eyes. Specifically, whenever a withdraw() operation is issued, the caller needs to provide the _proof which could verify the _recipient, the _relayer, etc. While verifying the _proof, the verifyProof() function of the verifier contract is invoked. If the verifyProof() function returns true, the funds are allowed to transferred to the recipient. This makes the verifier contract very important.

```
function withdraw(bytes calldata _proof, bytes32 _root, bytes32 _nullifierHash, address
    payable _recipient, address payable _relayer, uint256 _fee, uint256 _refund)
    external payable nonReentrant {
    require(_fee <= denomination, "Fee exceeds transfer value");
    require(!nullifierHashes[_nullifierHash], "The note has been already spent");
    require(isKnownRoot(_root), "Cannot find your merkle root"); // Make sure to use a
        recent one
```

```
    require(verifier.verifyProof(_proof, [uint256(_root), uint256(_nullifierHash),
        uint256(_recipient), uint256(_relayer), _fee, _refund]), "Invalid withdraw proof
        ");

    nullifierHashes[_nullifierHash] = true;
    _processWithdraw(_recipient, _relayer, _fee, _refund);
    emit Withdrawal(_recipient, _nullifierHash, _relayer, _fee);
}
```

Listing 3.23: Tornado.sol

While reviewing the implementation, we notice that the `verifier` could be updated by the privileged `updateVerifier()` function.

```
function updateVerifier(address _newVerifier) external onlyOperator {
    verifier = IVerifier(_newVerifier);
}
```

Listing 3.24: Tornado.sol

Therefore, a malicious or compromised `operator` could set the `verifier` to a crafted contract which returns `true` in `verifyProof()` only when the `recipient` points to a malicious address. This enables the bad actor to withdraw all the assets from the contract.

Recommendation Set the `operator` to a multisig or timelock contract to prevent single point of failure.

Status This issue has been fixed in the commit: a11349afc3585377dd02910f0a2ff8d34b926385.

4 | Conclusion

In this audit, we thoroughly analyzed the ADD.xyz V1 design and implementation. The system is a full-stack DeFi aggregator, plugging in multiple products and DeFi applications into one single platform, focusing on User experience, design, privacy and anonymity. During the audit, we notice that the current code base is well structured and neatly organized, and those identified issues are promptly confirmed and fixed.

Furthermore, 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 [10, 11, 12, 13, 15].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [16] 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-269: Improper Privilege Management. https://cwe.mitre.org/data/definitions/269.html.

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

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

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

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

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

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

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

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

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

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

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

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

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