Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

3JANE-EETH-X-C

Veridise Inc.

May 28, 2024

► **Prepared For:**

3Jane
https://www.3jane.xyz/

► **Prepared By:**

Ajinkya Rajput
Mark Anthony

► **Contact Us:** contact@veridise.com

► **Version History:**

May 29, 2024	V1
May 7, 2024	Initial Draft

Contents

1 Executive Summary

From Apr. 22, 2024 to Apr. 25, 2024, 3Jane engaged Veridise to review the security of their 3JANE-EETH-X-C project. The review covered the on-chain contracts of the 3JANE-EETH-X-C. The audit was limited to changes performed in 3 commits that adds some features to 3JANE-EETH-X-C protocol. Veridise conducted the assessment over 8 person-days, with 2 engineers reviewing code over 4 days on code changed in commits `9de142d,459b62d,6d1a622`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as an extensive manual code review.

Project summary. 3JANE-EETH-X-C is a crypto-native derivatives protocol built on restaking and cash-and-carry. The protocol implements a vault for each of the supported liquid staked token, through which the users can trade derivatives. Before the commits reviewed in this audit, the settlement of the trades happened periodically after a fixed interval. The audited commits introduce 2 functional changes to the protocol.

- Allows each vault to have a variable settlement period.
- Implement two deposit helper contracts that allow user to deposit tokens various tokens, swap them for non-staked tokens supported by the vault, stake them and deposit the liquid staked tokens to the corresponding vault.

The protocol interacts with contracts outside the protocol in deposit helpers. Few of these external contracts are upgradable.

Code assessment. The 3JANE-EETH-X-C developers provided the source code of the 3JANE-EETH-X-C contract for review. The code is forked from Ribbon-v2 protocol (Audit Report). The code is very well documented. 3JANE-EETH-X-C To facilitate the Veridise auditors' understanding of the code, access to the protocol's developer documentation website was provided, which documents the intended behavior of the protocol at a high level. Additionally, the code contained some in-line comments on structs and functions. The delivered source code also contained a test suite which the Veridise auditors noted tested many of the expected user-flows and much of the protocol's behavior. The clients also provided the auditors a write-up that summarized the changes and intended behaviour of those changes.

Summary of issues detected. The audit uncovered 6 issues, the most severe of which is a high severity issue, V-3JN-VUL-001 which is a divide before multiply issue which may set the management fee rate to zero. The Veridise auditors also identified issue 1 medium-severity issue, V-3JN-VUL-002, that points to insufficient check for slippage amount in a deposit helper. The Veridise auditors also identified 2 warnings, and 1 informational finding.

Among the 6 issues, 4 issues have been acknowledged and 5 issues have been fixed by the 3Jane. 1 issue is reported to be intended behaviour.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the 3JANE-EETH-X-C project and to avoid similar issues to those discovered in the audit in the future.

The protocol interacts with contracts outside the protocol. Specifically, the deposit helpers interact with 1inch Network Aggregation router to swap tokens. The auditors recommend the developers to minimize and tightly control these interactions.

The auditors also recommend the developers to add tests for the EthenaDepositHelper that manipulate the swap calldata.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
3JANE-EETH-X-C	`9de142d,459b62d,6d1a622`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Apr. 22 - Apr. 25, 2024	Manual & Tools	2	8 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	1	1	1
Medium-Severity Issues	1	0	0
Low-Severity Issues	1	1	1
Warning-Severity Issues	2	1	1
Informational-Severity Issues	1	1	1
TOTAL	6	4	4

Table 2.4: Category Breakdown.

Name	Number
Logic Error	3
Data Validation	1
Usability Issue	1
Maintainability	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of 3JANE-EETH-X-C's smart contracts. In our audit, we sought to answer questions such as:

- Is the introduction of variable settlement period consistent with other logic of protocol?
- Are the validations of all protocol parameters correct?
- Is the implementation of deposit helpers correct?
- Are all quantities calculated correctly?
- Can an attacker steal funds from protocol or users?
- Can an attacker make external calls on behalf of the users?
- Does the removal of the code introduce inconsistencies in the business logic?
- Are there any attack vectors introduced from interactions with other contracts?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of this audit is limited to the files

- `EthenaDepositHelper.sol`
- `EtherfiDepositHelper.sol`
- `VaultLifecycleWithSwap.sol`
- `RibbonThetaVaultStorage.sol`
- `RibbonThetaVaultWithSwap.sol`
- `RibbonVault.sol`

The contracts `EthenaDepositHelper`, and `EtherfiDepositHelper` are entirely new contracts which accept user funds, swap them to corresponding assets (accepted by the vaults) and deposit them to the RibbonVault. The contracts `VaultLifecycleWithSwap`, `RibbonThetaVaultStorage` and `RibbonThetaVaultWithSwap` are inherited by the `RibbonVault` and these contracts include some additions/removals from the Ribbon Finance implementation.

Methodology. Veridise auditors reviewed the reports of previous audits for 3JANE-EETH-X-C, inspected the provided tests, and read the 3JANE-EETH-X-C documentation. They then began a manual review of the code assisted by property-based testing. During the audit, the Veridise auditors regularly met with the 3JANE-EETH-X-C developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

Vulnerability Report 4

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-3JN-VUL-001	Divide before multiply in RibbonVault	High	Fixed
V-3JN-VUL-002	The slippage check applied in EthenaDepositHelp.	Medium	Confirming Fix
V-3JN-VUL-003	_swap() allows arbitray code execution between . .	Low	Fixed
V-3JN-VUL-004	Depositing with permit will fail for DAI and USDT	Warning	Fixed
V-3JN-VUL-005	currentOtokenPremium is not initialised	Warning	Intended Behavior
V-3JN-VUL-006	Revert messages for failed swaps are not inform. . .	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-3JN-VUL-001: Divide before multiply in RibbonVault

Severity	High	**Commit**	eecd846
Type	Logic Error	**Status**	Fixed
File(s)	`RibbonVault.sol`		
Location(s)	_perRoundManagementFee()		
Confirmed Fix At	8f4a409		

The protocol collects management fees per round. The per round management fee rate is calculated from annual management fee rate in the function `_perRoundManagementFee()`.

```
function _perRoundManagementFee(uint256 _managementFee)
    internal
    view
    returns (uint256)
{
    uint256 _period = period;
    uint256 feeDivider =
        _period % 30 == 0
            ? Vault.FEE_MULTIPLIER * (12 / (_period / 30))
            : WEEKS_PER_YEAR / (_period / 7);

    // We are dividing annualized management fee by num weeks in a year
    return _managementFee.mul(Vault.FEE_MULTIPLIER).div(feeDivider);
}
```

Snippet 4.1: Snippet from `_perRoundManagementFee()`

In this function per round fee rate is calculated by dividing annual fee rate by `feeDivider`. The the function calculates `feeDivider` as follows

- If the period is a multiple of 30,
  - then set `feeDivider` to

```
Vault.FEE_MULTIPLIER * (12 / (_period / 30))
```

  - else set `feeDivider` to

```
WEEKS_PER_YEAR / (_period / 7);
```

True branch of ternary operator: In the true branch above, if `_period>360` then then the expression `(_period / 30)` is greater than 12. This causes the subexpression `(12 / (_period / 30))` to evaluate to zero.

False branch of ternary operator: The denominator is calculated as `(_period/7)`. If period is less than 7, the denominator will be set to zero which will lead to spurious reverts due to executing a divide by zero operation.

One of the inline documentation comment mentions that the available values of `period` are one of 7,14,30,90,360. However, there is no validation in code that enforces this. Also, protocol may further update the policies so that longer periods are allowed, in such a case this bug will get triggered if not updated at such a time.

Impact Protocol will collect zero management fees per rounds

Recommendation Calculate the `feeDivider` as

```
uint256 feeDivider =
  _period % 30 == 0
      ? (Vault.FEE_MULTIPLIER * 12 * 30)/ _period
      : (WEEKS_PER_YEAR * 7) / _period;
```

Snippet 4.2: Recommended code change

4.1.2 V-3JN-VUL-002: The slippage check applied in EthenaDepositHelper is incorrect

Severity	Medium	**Commit**	eecd846
Type	Logic Error	**Status**	Confirming Fix
File(s)	EthenaDepositHelper		
Location(s)	_swap()		
Confirmed Fix At	N/A		

After a user has deposited his assets to the contract the assets are then swapped to USDE by calling the 1inch Router. After the swap has concluded the contract ensures that it has received sufficient USDE balance and the amount of slippage faced during the swap is within the defined slippage parameters. See snippet below for reference.

A oversight here is that `_usdeBal` and `amountAdj` are in different asset terms and should not be directly compared. The former is in terms of `USDE` whereas the latter is in terms of the asset provided by the user. Comparing them as it is will cause inconsistencies in the slippage check.

Depending on the exchange rate between the two there are chances of a transaction failing even if the slippage is within the defined limit. And the chances of such an issue arising increase as the ratio of `price_USDE / price_asset` increases. Conversely as the ratio of `price_asset / price_USDE` increases the transaction may be accepted on the basis of currently defined slippage but the actual realized slippage may be much more.

For example, let us assume the current ratio of `price_DAI : price_USDE` is 1.07. A user swaps 1 DAI for USDE and the contract gets 0.97 USDE in return post swap. Without any slippage the user would have received 1.07 USDE.

As per the currently calculated slippage, `slippage => (1 - 0.97) = 3%` whereas the `actual_slippage = (1.07 - 0.97) = 10%` which is much larger.

```
// Target call must result in sufficient USDe
require(
    _usdeBal >=
        amountAdj.mul(MAX_SLIPPAGE.sub(slippage)).div(MAX_SLIPPAGE),
    "!_usdeBal"
);

return _usdeBal;
```

Snippet 4.3: Snippet from `_swap()`

Impact A user can face a larger amount of slippage during the swap compared to the defined value. Or a transaction can get rejected because of the slippage check even if the actual slippage is within intended bounds.

Recommendation Consider letting users pass in the amount of minimum USDE they want to be able to deposit for sUSDE with their provided assets. The contract can then ensure that the amount received after swapping the assets is greater than the minimum that the user desires.

4.1.3 V-3JN-VUL-003: _swap() allows arbitray code execution between swaps

Severity	Low	**Commit**	eecd846
Type	Data Validation	**Status**	Fixed
File(s)	`EthenaDepositHlper.sol`		
Location(s)	_swap()		
Confirmed Fix At	N/A		

The protocol supports an Ethena vault that accepts staked USDe tokens, sUSDe as assets. The developers have defined the `EthenaDepositHelper` contract, that helps the user to deposit to the vault in any of the following tokens

- USDC
- USDT
- DAI
- USDE

`EthenaDepositHelper` accepts the tokens in any of the above currencies, swaps the received tokens into USDe if the received tokens are not in USDe, stakes them to get sUSDe tokens and deposits the staked tokens to Ethena vault.

If the tokens are not of the type USDe, the protocol calls swaps them to USDe by using "1inch.io" router. This is performed in the `_swap()` internal function as shown below.

This function first gets the balance of `EthenaDepositHelper` before the swap and stores it in `_usdeBalBefore`. Then makes a low level call to the swap router and passes the `calldata` received from the caller of this function. i.e. either `deposit()` function or `depositWithPermit()` function.

This function is not validated and is passed along as it is to the router.

Impact An attacker can send crafted calldata that can call any function in the `TARGET` router. This allows arbitrary code execution between swaps.

Recommendation The calldata to be sent to the `TARGET` router should be constructed within the contract itself. It reduces the risk of allowing users to craft arbitrary calldata and also allows the contract more control over the swap procedure.

```
function _swap(
    IERC20 _asset,
    uint256 _amount,
    bytes calldata _data
) internal returns (uint256) {
    uint256 _usdeBalBefore = USDE.balanceOf(address(this));

    // Double-approve for non-compliant USDT
    if (_asset == USDT) {
        USDT.safeApprove(TARGET, 0);
        USDT.safeApprove(TARGET, _amount);
    }

    (bool success, ) = TARGET.call(_data);
    require(success, "!success");

    uint256 _usdeBal = USDE.balanceOf(address(this)).sub(_usdeBalBefore);
    uint256 amountAdj =
        _amount.mul(
            10 **
                (
                    uint256(18).sub(
                        IERC20Detailed(address(_asset)).decimals()
                    )
                )
        );

    // Target call must result in sufficient USDe
    require(
        _usdeBal >=
            amountAdj.mul(MAX_SLIPPAGE.sub(slippage)).div(MAX_SLIPPAGE),
        "!_usdeBal"
    );

    return _usdeBal;
}
```

Snippet 4.4: Snippet from `_swap()`

4.1.4 V-3JN-VUL-004: Depositing with permit will fail for DAI and USDT

Severity	Warning	**Commit**	eecd846
Type	Usability Issue	**Status**	Fixed
File(s)	`EthenaDepositHelper.sol`		
Location(s)	depositWithPermit()		
Confirmed Fix At	965697a		

The function `depositWithPermit()` is meant to allow users to deposit assets to the Vault by making use of signed messages. The permit implementation conforms to the ERC2612 standard. See snippet of the function below.

There is a tacit assumption that `IERC20Permit(asset).permit()` will succeed for all the tokens which the protocols intends to accept for the deposits. Based on the developer comments the supported tokens are (`USDE`, `USDC`, `USDT`, `DAI`). But the `USDT` implementation does not include a permit function. And the `DAI` permit implementation is non-standard so it does not follow the ERC2612 standard.

As a result `depositWithPermit()` will fail for both DAI and USDC.

```
function depositWithPermit(
    IERC20 _asset,
    uint256 _amount,
    bytes calldata _data,
    uint256 _deadline,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
) external {
    // Sign for transfer approval
    IERC20Permit(address(_asset)).permit(
        msg.sender,
        address(this),
        _amount,
        _deadline,
        _v,
        _r,
        _s
    );

    _deposit(_asset, _amount, _data);
}
```

Snippet 4.5: Snippet of Function `depositWithPermit()`

Impact Transactions calling `depositWithPermit()` for assets DAI and USDT will revert.

Recommendation Write a safePermit() function which gracefully handles the exception for the DAI permit implementation. Also, consider documenting the assets for which depositing with permit will not work.

4.1.5 V-3JN-VUL-005: currentOtokenPremium is not initialised

Severity	Warning	**Commit**	eecd846
Type	Logic Error	**Status**	Intended Behavior
File(s)	`RibbonVault.sol`		
Location(s)	See Issue Description		
Confirmed Fix At	N/A		

The protocol implements options using `oTokens`. It mints and sells them for a premium. The premium is stored in the state variable `currentOtokenPremium`. This variable is used in `_createOffer` function in `RibbonThetaVaultWithSwap`. This function calls `createOffer()` in `VaultLifecycleWithSwap` which reverts if `currentOtokenPremium` is zero.

```
function _createOffer() private {
    address currentOtoken = optionState.currentOption;
    uint256 currOtokenPremium = currentOtokenPremium;

    optionAuctionID = VaultLifecycleWithSwap.createOffer(
        currentOtoken,
        currOtokenPremium,
        SWAP_CONTRACT,
        vaultParams
    );
}
```

Snippet 4.6: Snippet from `_createOffer()`

This variable is not initialised in initialisers therefore, this variable has a default value of zero when contract is deployed. This variable is set in function `setMinPrice()` in contract `RibbonThetaVaultWithSwap`

```
function setMinPrice(uint256 minPrice) external onlyKeeper {
    require(minPrice > 0, "!minPrice");
    currentOtokenPremium = minPrice;
}
```

Snippet 4.7: Snippet from `_createOffer()`

Impact If a deployment script misses to call `setMinPrice()`, the protocol will revert till `currentOtokenPremium` is set.

Recommendation Initialise the value of `currentOtokenPremium` in initialiser.

Developer Response The developers responded

> will leave as is

4.1.6 V-3JN-VUL-006: Revert messages for failed swaps are not informative

Severity	Info	**Commit**	eecd846
Type	Maintainability	**Status**	Fixed
File(s)	`EthenaDepositHelper.sol`		
Location(s)	_swap()		
Confirmed Fix At	ceb1a57		

In the `EthenaDepositHelper` contract the `_swap()` function calls the 1inch Network Aggregation Router to swap the `_amount` of provided asset to USDE. The routers address is stored in constant `TARGET`. See below snippet for the call to `TARGET`.

If for any reason the external call to `TARGET` fails, the only revert error shown is `!success` which is neither informative nor helpful in debugging transaction failure.

```
function _swap(
    IERC20 _asset,
    uint256 _amount,
    bytes calldata _data
) internal returns (uint256) {
uint256 _usdeBalBefore = USDE.balanceOf(address(this));
if (_asset == USDT) {
    USDT.safeApprove(TARGET, 0);
    USDT.safeApprove(TARGET, _amount);
}
(bool success, ) = TARGET.call(_data);
require(success, "!success");
```

Snippet 4.8: Snippet from `_swap()`

Impact Debugging failed swaps will be troublesome in absence of proper revert information as there are multiple ways to swap using the 1inch Router and the calldata passed for the call is arbitrary.

Recommendation Have the revert error message from the 1inch router be bubbled up for more accurate information as to why the transaction failed. See below snippet for example implementation.

```
(bool success, bytes memory result) = TARGET.call(_data);
if (!success) { // If call reverts
  // If there is return data, the call reverted without a reason or a custom error.
  if (result.length == 0) revert();
  assembly {
    // We use Yul's revert() to bubble up errors from the target contract.
    revert(add(32, result), mload(result))
  }
}
```