

SALUS SECURITY

JUL 2025



CODE SECURITY ASSESSMENT

VANILLA FINANCE

Overview

Project Summary

- Name: Vanilla Finance - MemePerps
- Platform: The BSC Blockchain
- Language: Solidity
- Repository:
 - <https://github.com/VanillaDevTeam/MemePerps>
- Audit Range: See [Appendix - 1](#)

Project Dashboard

Application Summary

Name	Vanilla Finance - MemePerps
Version	v4
Type	Solidity
Dates	Aug 02 2025
Logs	Jul 18 2025; Jul 22 2025; Jul 23 2025; Aug 02 2025

Vulnerability Summary

Total High-Severity issues	4
Total Medium-Severity issues	4
Total Low-Severity issues	4
Total informational issues	3
Total	15

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction	4
1.1 About SALUS	4
1.2 Audit Breakdown	4
1.3 Disclaimer	4
Findings	5
2.1 Summary of Findings	5
2.2 Notable Findings	5
1. Signature not bound to parameters enables forged and replay Attacks	6
2. Unvalidated params enables bad debt and draining of protocol-held balance	8
3. The whitelist pool will be drained	10
4. Liquidity manipulation attack	11
5. Inflation attack	12
6. Pools remain active after token de-whitelisting, with no pause/close mechanism	13
7. Lack of slippage check in closePositionWithData	14
8. Centralization risk	15
9. Create pool can be front-running	16
10. Unit mismatch in getQuoterAmountIn	17
11. Inconsistent decimal handling	18
12. Use a strict less than sign in getAllPositions	19
2.3 Informational Findings	20
13. Gas optimization suggestions	20
14. Use of floating pragma	21
15. Callback adaptation error	22
Appendix	23
Appendix 1 - Files in Scope	23

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (<https://t.me/salusec>), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Signature not bound to parameters enables forged and replay Attacks	High	Business Logic	Resolved
2	Unvalidated params enables bad debt and draining of protocol-held balance	High	Data Validation	Mitigated
3	The whitelist pool will be drained	High	Business logic	Resolved
4	Liquidity manipulation attack	High	Business logic	Mitigated
5	Inflation attack	Medium	Business Logic	Resolved
6	Pools remain active after token de-whitelisting, with no pause/close mechanism	Medium	Configuration	Resolved
7	Lack of slippage check in closePositionWithData	Medium	Data Validation	Resolved
8	Centralization risk	Medium	Centralization	Resolved
9	Create pool can be front-running	Low	Front-running	Resolved
10	Unit mismatch in getQuoterAmountIn	Low	Numerics	Resolved
11	Inconsistent decimal handling	Low	Inconsistency	Resolved
12	Use a strict less than sign in getAllPositions	Low	Business Logic	Resolved
13	Gas optimization suggestions	Informational	Gas optimization	Resolved
14	Use of floating pragma	Informational	Configuration	Resolved
15	Callback adaptation error	Informational	Business Logic	Resolved

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Signature not bound to parameters enables forged and replay Attacks

Severity: High

Category: Business logic

Target:

- contracts/leverage/facets/LeverageTradingOperationsFacet.sol

Description

The functions `_decodeOpenPositionParams()`, `_decodeClosePositionParams()`, `_decodeLiquidatePositionParams()`, each split the caller-supplied `signedData` into `(bytes32 message, uint8 v, bytes32 r, bytes32 s, <Params> params)`, verifying only that `(v,r,s)` form a valid signature over the 32-byte message

contracts/leverage/facets/LeverageTradingOperationsFacet.sol:L1015-L1088

```
function _decodeOpenPositionParams(
    bytes calldata signedData
) internal view returns (OpenPositionParams memory) {
    (
        bytes32 message,
        uint8 v,
        bytes32 r,
        bytes32 s,
        OpenPositionParams memory params
    ) = abi.decode(
        signedData,
        (bytes32, uint8, bytes32, bytes32, OpenPositionParams)
    );

    // Create Ethereum signed message hash
    bytes32 ethSignedMessageHash = keccak256(
        abi.encodePacked("\x19Ethereum Signed Message:\n32", message)
    );
    address signer = ecrecover(ethSignedMessageHash, v, r, s);
    if (!hasRole(DATA_PROVIDER_ROLE, signer))
        revert Error.Unauthorized();

    return params;
}

function _decodeClosePositionParams(bytes calldata signedData) internal view
returns (ClosePositionParams memory) {...}

function _decodeLiquidatePositionParams(bytes calldata signedData) internal view
```



```
returns (LiquidatePositionParams memory) {...}
```

The contract never verifies that the message is derived from, or otherwise commits to, the decoded parameters.

As a result, an attacker with access to any previously published signature from a whitelisted ``DATA_PROVIDER_ROLE`` address can craft malicious signedData packets by:

1. Re-use a valid ``(message, v, r, s)`` tuple.
2. Append arbitrary ``OpenPositionParams``, ``ClosePositionParams``, or ``LiquidatePositionParams``.
3. Replaying the same signature indefinitely, since there is no nonce or deadline mechanism in place to prevent reuse..

Recommendation

Bind signature to ``params``, add replay protection(``nonce``, ``deadline``), adopt EIP-712 typed-data signatures so the whole struct is signed in an unforgeable way.

Status

The team has resolved this issue in commit [373f490](#).

2. Unvalidated params enables bad debt and draining of protocol-held balance

Severity: High

Category: Data Validation

Target:

- contracts/leverage/facets/LeverageTradingOperationsFacet.sol

Description

In the ``openPositionWithData()``, ``closePositionWithData()``, and ``liquidatePositionWithData()`` functions, the protocol accepts off-chain signed structs (``OpenPositionParams``, ``ClosePositionParams``, ``LiquidatePositionParams``). However, many of the economically critical fields in these structs are only subjected to basic sanity checks on-chain.

contracts/leverage/facets/LeverageTradingOperationsFacet.sol:L148-L179

```
struct OpenPositionParams {
    address marginToken;
    address targetToken;
    uint256 marginAmount;
    uint256 maintenanceMarginRate;
    uint256 borrowAmount;
    uint256 fee;
    uint256 swapAmount;
    uint256 leverageMultiplier;
    uint256 minTargetTokenAmount;
    uint256 slippage;
    uint256 positionId;
    bool longOrShort;
    uint256 deadline;
}

struct ClosePositionParams {
    uint256 positionId;
    uint256 slippage;
    uint256 interest;
    bytes interestHistory;
    uint256 deadline;
}

struct LiquidatePositionParams {
    uint256 positionId;
    uint256 liquidationPenalty;
    uint256 liquidatorRewardRate;
    uint256 totalInterest;
    bytes interestHistory;
    uint256 deadline;
}
```

Coupled with the previously reported issue “**Signature Not Bound to Parameters enables forged and replay Attacks**” (any valid ``(v,r,s)`` can be reused against arbitrary structs), an

attacker can Inject unlimited fees, exaggerate leverageMultiplier, and perform other unauthorized manipulations, for example:

When opening a position, the untrusted `swapAmount` is used at

When opening a position, the untrusted `swapAmount` is used in the following critical functions: `Utils.forceApprove(tokenIn, dexRouter, swapAmount)`, `dexRouter.exactInputSingle(... amountIn: swapAmount ...)`,. However, there is no invariant that ties `swapAmount` to the sum of (margin + borrow) for long positions or to the borrowed target amount for short positions. Additionally, there is no balance delta check or cap, and signatures are not bound to parameters (see separate High finding). This allows attackers to supply arbitrary values, exploiting the system.

Attack A – Under-Swap to Manufacture Bad Debt (Insolvency)

An attacker can borrow a large amount but set a minuscule `swapAmount`. Most of the borrowed tokens remain idle in the contract, never being swapped into the position. As a result, the accounting records a large `borrowedAmount` but only a tiny `actualTokenAmount`. Upon position closure, repayment is calculated based on the small traded exposure, leaving the position in NegativeEquity. The lending pool absorbs the shortfall while the borrowed tokens remain stranded in the contract. Repeating this attack can drain the pool's solvency.

Attack B – Over-Swap to Drain Protocol-Held Balances

An attacker can open a small position while supplying a disproportionately large `swapAmount`, potentially up to the contract's full balance. The `_openPosition` function will approve and execute the entire trade, resulting in a large `actualTargetAmount` being attributed to the attacker's position. Upon closing, only the small recorded borrow is repaid, and the excess proceeds are returned to the attacker as `marginReturned`, allowing them to effectively loot protocol-owned or other users' residual funds.

Recommendation

Add on chain validation for these params.

Status

The team has mitigated this issue in commit [2a91761](#).

3. The whitelist pool will be drained

Severity: High

Category: Business logic

Target:

- contracts/leverage/facets/LeverageTradingOperationsFacet.sol

Description

In the `LeverageTradingOperationsFacet` contract, the `createToken()` function will select the correct fund pool for subsequent borrowing and trading operations according to the direction selected by the user (long/short). But this only checks one token.

contracts/leverage/facets/LeverageTradingOperationsFacet.sol:L259-L300

```
function openPositionWithData(...) {  
    ...  
    uint256 poolId;  
    if (params.longOrShort) {  
        poolId = Its.stakingContract.findPoolByToken(actualMarginToken);  
    } else {  
        poolId = Its.stakingContract.findPoolByToken(params.targetToken);  
    }  
  
    if (poolId == type(uint256).max) revert("Pool not supported");  
}
```

Attach Scenario

1. The protocol adds `USDT` to white list.
2. Attacker builds a `FAKE-USDT` pool with a small supply ratio of 1 `FAKE` : 100 `USDT`.
3. Attacker opens a 10U ten-times leveraged `FAKE` long position. The protocol will swap 100 `USDT` for `FAKE` and the `USDT` supply of the pool will increase to 200 `USDT`.
4. The attacker uses `FAKE` tokens to drain the pool. The profit is 100U, the cost is 10U, resulting in a net gain of 90U

Through the above method, the attacker can drain all whitelisted tokens.

Recommendation

Add a check for ensuring both `marginToken` and `targetToken` are on the whitelist.

Status

The team has resolved this issue in commit [373f490](#).

4. Liquidity manipulation attack

Severity: High

Category: Business logic

Target:

- contracts/leverage/PriceFeed.sol

Description

The contract's `findPool()` function finds the pool with the highest liquidity between tokenA and tokenB across different fee tiers in the PancakeSwap V3 factory and returns its address. This introduces a potential issue where the pool used to open a position differs from the one used to close it, potentially enabling price manipulation.

contracts/leverage/PriceFeed.sol:L82-L106

```
function findPool(...)
{
    address[] memory pools = new address[](4);
    pools[0] = IPancakeV3Factory(factory).getPool(tokenA, tokenB, 100);
    pools[1] = IPancakeV3Factory(factory).getPool(tokenA, tokenB, 500);
    pools[2] = IPancakeV3Factory(factory).getPool(tokenA, tokenB, 2500);
    pools[3] = IPancakeV3Factory(factory).getPool(tokenA, tokenB, 10000);
    //get best liquidity pool with best liquidity
    uint256 bestLiquidity = 0;
    address bestPool = address(0);
    for (uint256 i = 0; i < pools.length; i++) {
        if (pools[i] != address(0)) {
            uint256 liquidity = IPancakeV3Pool(pools[i]).liquidity();
            if (liquidity > bestLiquidity) {
                bestLiquidity = liquidity;
                bestPool = pools[i];
            }
        }
    }

    return bestPool;
}
```

Attach Scenario

1. The ``Meme-USDT`` has one uninitialized pool and three pools with existing liquidity.
2. Attacker initialize the pool at a high price and open long positions at the same time.
3. By leveraging the flash loan feature of the pools, the attacker combines the liquidity of three pools to create one pool with high liquidity.
4. The attacker closes the position using the manipulated pool for settlement, realizes the profit, and repays the flash loans from the three pools

Recommendation

Add a mechanism to ensure price consistency by using the same pool for both opening and closing positions.

Status

The team has mitigated this issue in commit [373f490](#).

5. Inflation attack

Severity: Medium

Category: Business Logic

Target:

- contracts/staking/facets/StackOperationFacet.sol

Description

The protocol rounds down for ``unstake()`` and ``_stake()`` functions. For an empty pool, the malicious user can manipulate it to cause inflation attacks. Specifically, the normal ratio of share to amount will not exceed 1:2.

The attacker also can transfer to the contract directly, creating inconsistency between the contract's internal and recorded balances.

The two ways can lead to Inflation attacks which makes this ratio very large.

contracts/staking/facets/StackOperationFacet.sol:L233-323

```
function unstake(uint256 _pid, uint256 _amount) external returns (uint256) {
    ...
    uint256 balance = _balanceAndBorrowedOfPool(_pid);
    uint256 userShares = _amount.mulDiv(user.shares, user.amount);
    uint256 normalizedRewards = calculateAmount(
        userShares,
        pool.totalShares,
        balance
    );
    ...
}
function _stake(uint256 _pid, uint256 _amount, address _staker) internal {
    ...
    if (pool.totalStaked == 0) {
        shares = normalizedAmount;
    } else {
        shares = calculateShares(
            normalizedAmount,
            totalStaked,
            pool.totalShares
        );
    }
    ...
}
```

Due to the pool lack of lending attack vector, the severity is medium.

Recommendation

It is recommended to ensure that markets are never empty by minting small share (or equivalent) balances at the time of pool creation, preventing the rounding error being used maliciously. Or use one-to-one minting when ``totalShares`` and ``totalStaked`` are at smaller values.

Status

The team has resolved this issue in commit [373f490](#).

6. Pools remain active after token de-whitelisting, with no pause/close mechanism

Severity: Medium

Category: Configuration

Target:

- contracts/staking/facets/StackOperationFacet.sol

Description

`StackOperationFacet.batchRemoveFromWhitelist()` removes a token from the `whitelistTokensArray`, but does not touch the associated pool state:

1. `s.pools[pid].isActive` is left true.
2. `s.tokenToPoolId[_token]` and `s.poolExists[_token]` remain set.
3. `s.creatorPools[...]` still references the pool.
4. No call path marks the pool disabled or prevents subsequent use.

As a result, after governance removes a token, users can continue to route leverage trades through the stale pool.

Recommendation

Update pool state variables when de-whitelisting a token and add checks in stake, unstake, borrow, and repay, e.g. `require(pool.isActive && isTokenWhitelisted(pool.token))`.

Status

The team has resolved this issue in commit [373f490](#).

7. Lack of slippage check in closePositionWithData

Severity: Medium

Category: Data Validation

Target:

- contracts/staking/facets/StackOperationFacet.sol

Description

The `ClosePositionParams` struct includes a `slippage` field, but `closePositionWithData()` never uses it to constrain the on-chain swap:

contracts/staking/facets/StackOperationFacet.sol:L164-L170

```
struct ClosePositionParams {
    uint256 positionId;
    uint256 slippage;
    uint256 interest;
    bytes interestHistory;
    uint256 deadline;
}
```

contracts/staking/facets/StackOperationFacet.sol:L480-L642

```
function closePositionWithData(bytes calldata data){
    if (position.openInfo.longOrShort) {
        allMarginAmount = Its.dexRouter.exactInputSingle(
            ISwapRouter.ExactInputSingleParams({
                tokenIn: tokenIn,
                tokenOut: tokenOut,
                fee: fee,
                recipient: address(this),
                amountIn: swapAmount,
                amountOutMinimum: 0,
                sqrtPriceLimitX96: 0,
                deadline: params.deadline
            })
        );
    } else {
        allMarginAmount = Its.dexRouter.exactOutputSingle(
            ISwapRouter.ExactOutputSingleParams({
                tokenIn: tokenIn,
                tokenOut: tokenOut,
                fee: fee,
                recipient: address(this),
                amountOut: swapAmount,
                amountInMaximum: type(uint256).max,
                sqrtPriceLimitX96: 0,
                deadline: params.deadline
            })
        );
    }
}
```

Recommendation

Add slippage checks in the `closePositionWithData()` function.

Status

The team has resolved this issue in commit [373f490](#).

8. Centralization risk

Severity: Medium

Category: Centralization

Target:

- contracts/staking/facets/StackOperationFacet.sol

Description

The `StackOperationFacet` contract has privileged accounts. These privileged accounts can borrow all tokens from the contracts without any collateral by using the `borrow()` function, and change any user's share by using the `updateUserShares()` functions.

If privileged accounts' private key or admin's is compromised, an attacker can steal all the tokens in the contract.

If the privileged accounts are plain EOA accounts, this can be worrisome and pose a risk to the other users.

Recommendation

We recommend transferring privileged accounts to multi-sig accounts with timelock governors for enhanced security. This ensures that no single person has full control over the accounts and that any changes must be authorized by multiple parties.

Status

The team has resolved this issue in commit [373f490](#) and state that a multi-signature wallet will be used on the official network.

9. Create pool can be front-running

Severity: Low

Category: Front-running

Target:

- contracts/staking/facets/StackOperationFacet.sol

Description

In the StackOperationFacet contract, once the contract owner calls ``addTokenToWhitelist()`` function:

contracts/staking/facets/StackOperationFacet.sol:L102-L104

```
function addTokenToWhitelist(address _token) external onlyOwner {  
    _addTokenToWhitelistArray(_token);  
}
```

An attacker can monitor the mempool for the transaction and then call the ``stakeByToken()`` function, front-run the subsequently expected ``createPool()`` call by the contract owner.

contracts/staking/facets/StackOperationFacet.sol:L201-L225

```
function stakeByToken(  
    address _token,  
    uint256 _amount  
) external returns (uint256) {  
    require(_amount > 0, "Amount must be greater than 0");  
    if (_token == address(0)) {  
        _token = Constants.WETH_ADDRESS;  
    }  
    require(  
        IStackQueryFacet(address(this)).isTokenWhitelisted(_token),  
        "Token not whitelisted"  
    );  
    uint256 pid = IStackQueryFacet(address(this)).findPoolByToken(_token);  
  
    if (pid == type(uint256).max) {  
        pid = _createPool(_token, msg.sender);  
    }  
    _stake(pid, _amount, msg.sender);  
    return pid;  
}
```

Therefore, the attacker will become the pool creator, although there are no privileges for creator.

Recommendation

Add an ``onlyOwner`` modifier on the internal ``_createPool()``, or restrict the create pool path in ``stakeByToken()``.

Status

The team has resolved this issue in commit [373f490](#).

10. Unit mismatch in getQuoterAmountIn

Severity: Low

Category: Numerics

Target:

- contracts/leverage/facets/LeverageTradingQueryFacet.sol

Description

`getQuoterAmountIn()` queries `IQuoter.quoteExactOutputSingle` with `tokenIn = baseToken` and `tokenOut = quoteToken`. The quoter returns the raw `amountIn` denominated in `baseToken` decimals. However, the code normalizes this value using `quoteToken` decimals:

contracts/leverage/facets/LeverageTradingQueryFacet.sol:L163-L202

```
function getQuoterAmountIn(
    address baseToken,
    address quoteToken,
    uint256 amountOut
) external returns (uint256 amountIn) {
    ...
    (amountIn, , , ) = IQuoter(lts.quoter).quoteExactOutputSingle(
        IQuoter.QuoteExactOutputSingleParams({
            tokenIn: baseToken,
            tokenOut: quoteToken,
            fee: fee,
            amount: Utils.denormalizeTokenAmount(quoteToken, amountOut),
            sqrtPriceLimitX96: 0
        })
    );

    amountIn = Utils.normalizeTokenAmount(quoteToken, amountIn);

    return amountIn;
}
```

If the two tokens have different decimal counts (e.g., 18 vs 6), the normalized result is scaled incorrectly.

Recommendation

Normalize the returned `amountIn` using `baseToken` decimals.

Status

The team has resolved this issue in commit [373f490](#) and state that the function is called externally, so the original precision of the currency is retained.

11. Inconsistent decimal handling

Severity: Low

Category: Inconsistency

Target:

- contracts/leverage/facets/LeverageTradingQueryFacet.sol

Description

`getQuoterAmountOut()` is intended to return how much `baseToken` you would receive for supplying `amountIn` units of `quoteToken`. However, there is no decimal normalization or denormalization.

contracts/leverage/facets/LeverageTradingQueryFacet.sol:L113-L161

```
function getQuoterAmountOut(
    address baseToken,
    address quoteToken,
    uint256 amountIn
) external returns (uint256) {
    ...
    (uint256 amountOut, , , ) = IQuoter(lts.quoter).quoteExactInputSingle(
        IQuoter.QuoteExactInputSingleParams({
            tokenIn: quoteToken,
            tokenOut: baseToken,
            fee: fee,
            amountIn: amountIn,
            sqrtPriceLimitX96: 0
        })
    );
    Utils.clearAllowance(
        IERC20(quoteToken),
        address(lts.quoter)
    );
    return amountOut;
}
```

Unlike `getQuoterAmountIn()`, which at least attempts to denormalize/normalize units, this function passes `amountIn` straight through to the quoter.

Additionally, these functions should be read-only; approving tokens for a third-party contract just to compute a quote is unnecessary and dangerous.

Recommendation

Remove all token approvals from quote functions, and add the normalization and denormalization.

Status

The team has resolved this issue in commit [373f490](#) and state that the function is called externally, so the original precision of the currency is retained.

12. Use a strict less than sign in getAllPositions

Severity: Low

Category: Business Logic

Target:

- contracts/leverage/facets/LeverageTradingQueryFacet.sol

Description

Loop excludes `endIndex` because of `<` instead of `<=`:

contracts/leverage/facets/LeverageTradingQueryFacet.sol:L260-L291

```
function getAllPositions(uint256 startIndex, uint256 endIndex) external view
    returns (LibLeverageTradingStorage.Position[] memory positions)
{
    ...
    if (endIndex >= Its.positionIds.length()) {
        endIndex = Its.positionIds.length() - 1;
    }

    uint256 positionsCount = endIndex - startIndex + 1;
    positions = new LibLeverageTradingStorage.Position[](positionsCount);

    for (uint256 i = startIndex; i < endIndex; i++) {
        positions[i] = Its.positions[Its.positionIds.at(i)];
    }
    return positions;
}
```

Recommendation

Use `<=` instead of `<`.

Status

The team has resolved this issue in commit [373f490](#).

2.3 Informational Findings

13. Gas optimization suggestions

Severity: Informational

Category: Gas Optimization

Target:

- contracts/staking/facets/StackOperationFacet.sol
- contracts/leverage/PriceFeed.sol

Description

Memory reading saves more gas than storage reading multiple times when the state is not changed. So caching the storage variables in memory and using the memory instead of storage reading is effective. Cache array length outside of the loop can save gas.

contracts/staking/facets/StackOperationFacet.sol:L115

```
for (uint256 i = 0; i < _tokens.length; i++) {
```

contracts/staking/facets/StackOperationFacet.sol:L129

```
for (uint256 i = 0; i < _tokens.length; i++) {
```

contracts/leverage/PriceFeed.sol:L96

```
for (uint256 i = 0; i < pools.length; i++) {
```

In the `closePositionWithData()`, the code `position.closeInfo.isSet = true` is executed twice, and should be removed.

contracts/staking/facets/StackOperationFacet.sol:L543-L749

```
function closePositionWithData(bytes calldata data) external nonReentrant
returns (uint256 marginReturned, int256 realizedProfitLoss){
...
position.closeInfo.isSet = true;
...
position.closeInfo.isSet = true;
...
}
```

Recommendation

Consider using the above suggestions to save gas.

Status

The team has resolved this issue in commit [373f490](#).

14. Use of floating pragma

Severity: Informational

Category: Configuration

Target:

- All

Description

```
pragma solidity ^0.8.28;
```

The QuillToken uses a floating compiler version ^0.8.28.

Using a floating pragma ^0.8.28 statement is discouraged, as code may compile to different bytecodes with different compiler versions. Use a locked pragma statement to get a deterministic bytecode. Also use the latest Solidity version to get all the compiler features, bug fixes and optimizations.

Recommendation

It is recommended to use a locked Solidity version throughout the project. It is also recommended to use the most stable and up-to-date version.

Status

The team has resolved this issue in commit [373f490](#).

15. Callback adaptation error

Severity: Informational

Category: Business Logic

Target:

- contracts/leverage/facets/LeverageTradingOperationsFacet.sol

Description

During position opening, closing, and liquidation operations, the protocol performs token swaps with a call flow of `V3SwapRouter.exactOutputSingle -> pool.swap ->

V3SwapRouter.pancakeV3SwapCallback`, without triggering any `LeverageTradingOperationsFacet` contract's callback functions. This same situation also applies to Uniswap V3.

contracts/leverage/facets/LeverageTradingOperationsFacet.sol:L998-L1013

```
function pancakeV3SwapCallback(
    int256 amount0Delta,
    int256 amount1Delta,
    bytes calldata data
) external {
    PancakeHelper.pancakeV3SwapCallback(amount0Delta, amount1Delta, data);
}

// ISwapCallback implementation
function uniswapV3SwapCallback(
    int256 amount0Delta,
    int256 amount1Delta,
    bytes calldata data
) external {
    PancakeHelper.pancakeV3SwapCallback(amount0Delta, amount1Delta, data);
}
```

Recommendation

Remove the Callback function.

Status

The team has resolved this issue in commit [373f490](#).

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit [60818ee](#):

File	SHA-1 hash
contracts/Constants.sol	5a269719002f73d14291ef83ab700dd2c27a5ea4
contracts/Utils.sol	84e7d7d7d4eff1e729f7346d34c4d4615928d4d6
contracts/leverage/LeverageTradingDiamond.sol	699822fc1feb5fd0da6fc59f5c9e06cec1559efd
contracts/leverage/LeverageTradingInit.sol	3b9899b0374a463e3fb2a8e88b6450403e1b97c1
contracts/leverage/LibLeverageTradingStorage.sol	0ef11b42b8a2d06c6644c2890bf997e207906e2f
contracts/leverage/PancakeHelper.sol	8119ea74608e59b4edb62027e4a4e6f6f4fe1d7c
contracts/leverage/PriceFeed.sol	0a28feb9a360dc3bed054903847881199df9769
contracts/leverage/facets/LeverageTradingQueryFacet.sol	6bb8940cbb123bc3a08b920e25192ce9f14f8e5e
contracts/leverage/facets/LeverageTradingOperationsFacet.sol	ff68aeca4125c3aaf9771af83033c0dd10744851
contracts/staking/MultiTokenStakingDiamond.sol	3bc0bfe2ac8e18210c271b625685ccd0b4217916
contracts/staking/LibMultiTokenStakingStorage.sol	5e276b3caeddd39278b2d3d12da660235730a6ec
contracts/staking/MultiTokenStakingInit.sol	24fddfb4b448191e2d5f023b2f0187c4d53c9cbc
contracts/staking/facets/StackQueryFacet.sol	575fb45d14905394bb621022f68519329c9ca2dd
contracts/staking/facets/StackOperationFacet.sol	2006a04c099d81c6d93f328652f93592fdab92d4



SMART CONTRACT AUDIT REPORT

for

Vanilla Money/MarketMaker Vaults



Prepared By: Xiaomi Huang

PeckShield
April 22, 2025

Document Properties

Client	VanillaExchange
Title	Smart Contract Audit Report
Target	Vanilla Money/MarketMaker Vaults
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 22, 2025	Xuxian Jiang	Final Release
1.0-rc	April 21, 2024	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About Vanilla Money/MarketMaker Vaults	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Possibly Inconsistent UnStake Events in VanillaMarketMakerVault	11
3.2	Improved Order Creation/Settlement Logic in VanillaMoneyVault	12
3.3	Trust Issue Of Admin Keys	14
4	Conclusion	16
	References	17

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the Vanilla Money/MarketMaker Vaults contracts, 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 Vanilla Money/MarketMaker Vaults

This audit covers four specific Vanilla vaults contracts, i.e., VanillaMoneyVault, VanillaMoneyVaultV2, VanillaMarketMakerVault, and VanillaMarketMakerVaultV2. The first two vaults are mainly used for users to deposit and withdraw funds, as well as provide two order interfaces for users with BOT_ROLE to operate. The last two act as a fund storage and token collateral. After the user places an order, a portion of the user's deposit will be transferred to VanillaMarketMakeVault(V2). The user's collateral can serve as a betting against the platform to earn interest. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Audited Contracts

Item	Description
Target	Vanilla Money/MarketMaker Vaults
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	April 22, 2025

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

- <https://github.com/VanillaDevTeam/PSC-Contract.git> (3ddb000)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- <https://github.com/VanillaDevTeam/PSC-Contract.git> (750cda2)

1.2 About PeckShield

PeckShield Inc. [9] 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	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], 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 security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, 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 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 implementation of four `vanilla` vaults. 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 logic, 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	2	
Low	1	
Informational	0	
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in [Section 3](#).

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerabilities and 2 low-severity vulnerability.

Table 2.1: Key Vanilla Money/MarketMaker Vaults Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possibly Inconsistent UnStake Events in VanillaMarketMakerVault	Coding Practices	Resolved
PVE-002	Medium	Improved Order Creation/Settlement Logic in VanillaMoneyVault	Business Logic	Resolved
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

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 Possibly Inconsistent UnStake Events in VanillaMarketMakerVault

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: VanillaMarketMakerVault
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In Ethereum, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `VanillaMarketMakerVault` contract as an example. This contract is designed to be a `VanillaMarketMakerVault` that allows users to stake/unstake their funds. While examining the events that reflect the `unstake` operation, we notice the emitted important `UnStake` event may not be consistent. In particular, The `UnStake` event has four parameters and the last one indicates the respective `pledgedFunds` amount of the actual amount being transferred out. With that, the following `UnStake` event (line 200) in `partialUnstake()` is incorrect (while the same vent in `unstake()` is correct).

```
177     function partialUnstake(  
178         uint256 amount  
179     ) external nonReentrant whenNotPaused {  
180         uint256 balances = userInfo[_msgSender()].amounts;  
181         if (amount == 0 || amount > balances) {  
182             revert VanillaMarketMakerVault__InvalidAmount();  
183         }
```

```

184     uint256 shares = (amount * userInfo[_msgSender()].shares) / balances;
185     uint256 amountToTransfer = calculateAmounts(shares);
186     if (slot1.cumulativeShares < shares)
187         revert VanillaMarketMakerVault__cumulativeSharesInsufficient();
188     if (assetsManagement() < amountToTransfer)
189         revert VanillaMarketMakerVault__InsufficientVaultBalance();
190     slot1.pledgedFunds -= amount;
191     slot1.cumulativeShares -= shares;
192     userInfo[_msgSender()].shares -= shares;
193     userInfo[_msgSender()].amounts -= amount;

195     if (userInfo[_msgSender()].amounts == 0) {
196         userNumber -= 1;
197     }

199     IERC20(slot1.assetId).safeTransfer(_msgSender(), amountToTransfer);
200     emit UnStake(
201         _msgSender(),
202         amountToTransfer,
203         shares,
204         userInfo[_msgSender()].amounts
205     );
206 }

```

Listing 3.1: VanillaMarketMakerVault::partialUnstake()

Recommendation Properly emit the `UnStake` event when an user intends to unstake the staked funds.

Status This issue has been fixed in the following commit: 750cda2.

3.2 Improved Order Creation/Settlement Logic in VanillaMoneyVault

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: VanillaMoneyVault
- Category: Business Logic [6]
- CWE subcategory: CWE-770 [3]

Description

The `VanillaMoneyVault` contract allows the privileged bot accounts to place/settle user orders. In the process of examining the order creation and settlement logic, we notice current implementation may be improved.

```

106     function createOrder(
107         CreateOrderParams calldata params
108     ) external override onlyRole(BOT_ROLE) {
109         if (balances[params.account] < params.amount)
110             revert VanillaMoneyVault__PledgeFundInsufficient();
111         if (orderInfo[params.orderId].isExistence)
112             revert VanillaMoneyVault__AlreadyExistOrder(params.orderId);
113         orderInfo[params.orderId] = OrderInfo({
114             owner: params.account,
115             isSettlement: false,
116             isExistence: true,
117             amount: params.amount
118         });
119         balances[params.account] -= params.amount;
120         if (slot0.platformFeeAccount != address(0)) {
121             if (params.fee > 0) {
122                 balances[params.account] -= params.fee;
123                 IERC20(slot0.assetId).safeTransfer(
124                     slot0.platformFeeAccount,
125                     params.fee
126                 );
127                 emit PlatformCollectFee(slot0.platformFeeAccount, params.fee);
128             }
129         }
130
131         emit CreateOrder(params.account, params.orderId, params);
132     }

```

Listing 3.2: VanillaMoneyVault::createOrder()

To elaborate, we show above the implementation of the related `createOrder()` routine. When creating an order, there is a need to ensure the user funds are sufficient to cover the order amount as well as possible fee. However, current implementation only validates the coverage of order amount, not the fee. Also, the given input parameters are defined in `CreateOrderParams`, which contains a number of unused member fields and unused ones can be simplified removed.

```

134     function settleOrder(
135         bytes32 orderId,
136         uint256 revenue,
137         uint256 fee
138     ) public override onlyRole(BOT_ROLE) {
139         if (orderInfo[orderId].isSettlement)
140             revert VanillaMoneyVault__AlreadySettleOrder(orderId);
141         orderInfo[orderId].isSettlement = true;
142         address account = orderInfo[orderId].owner;
143         // transfer
144         IERC20(slot0.assetId).safeTransfer(
145             slot0.marketMakerVault,
146             orderInfo[orderId].amount
147         );
148
149         IVanillaMarketMakerVault(slot0.marketMakerVault).settlement(

```

```

150         account,
151         revenue + fee
152     );
153     balances[account] += revenue;
154     if (fee > 0) {
155         IERC20(slot0.assetId).safeTransfer(slot0.profitSharingAccount, fee);
156         emit ProfitSharingCollectFee(slot0.profitSharingAccount, fee);
157     }
158
159     emit SettleOrder(account, orderId, revenue);
160 }

```

Listing 3.3: VanillaMoneyVault::settleOrder()

Similarly, the `settleOrder()` routine in the same contract can also be improved by validating the given order is a valid one, i.e., `require (orderInfo[params.orderId].isExistence);`.

Recommendation Revisit the above-mentioned routines to ensure the user orders are properly created and settled.

Status This issue has been fixed in the following commit: 750cda2.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the audited Vanilla vaults, there is a privileged account (with the `ADMIN_ROLE/DEFAULT_ADMIN_ROLE` role) that plays a critical role in governing and regulating the vault-wide operations (e.g., assign BOT roles, pause/unpause the vault, and settle orders). In the following, we show the representative functions potentially affected by the privilege of the privileged account.

```

106     function createOrder(
107         CreateOrderParams calldata params
108     ) external override onlyRole(BOT_ROLE) {
109         ...
110     }
111
112     function settleOrder(
113         bytes32 orderId,
114         uint256 revenue,
115         uint256 fee

```

```
116     ) public override onlyRole(BOT_ROLE) {  
117         ...  
118     }  
119     ...
```

Listing 3.4: Example Privileged Operations in `VanillaMoneyVault`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive vault parameters, which directly undermines the assumption of the vault design.

In the meantime, the vault contract makes use of the proxy contract to allow for future upgrades. The upgrade is a privileged operation, which also falls in this trust issue on the admin key.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

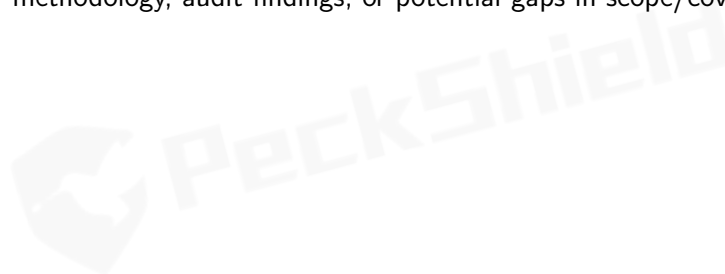
Status The issue has been confirmed and will be mitigated with the use of a multi-sig to manage the privileged account.



4 | Conclusion

In this audit, we have analyzed the design and implementation of four specific `Vanilla` vaults contracts, i.e., `VanillaMoneyVault`, `VanillaMoneyVaultV2`, `VanillaMarketMakerVault`, and `VanillaMarketMakerVaultV2`. The first two vaults are mainly used for users to deposit and withdraw funds, as well as provide two order interfaces for users with `BOT_ROLE` to operate. The last two act as a fund storage and token collateral. After the user places an order, a portion of the user's deposit will be transferred to `VanillaMarketMakeVault(V2)`. The user's collateral can serve as a betting against the platform to earn interest. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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.



References

- [1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [2] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [3] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. <https://cwe.mitre.org/data/definitions/770.html>.
- [4] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [5] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [6] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [7] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [9] PeckShield. PeckShield Inc. <https://www.peckshield.com>.