SMART CONTRACT AUDIT REPORT

for

Evrynet Protocol

Prepared By: Yiqun Chen

PeckShield

November 15, 2021

Document Properties

Client	Evrynet Finance
Title	Smart Contract Audit Report
Target	Evrynet Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 15, 2021	Xuxian Jiang	Final Release
1.0-rc	September 24, 2021	Xuxian Jiang	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Evrynet` protocol, 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 Evrynet Protocol

The `Evrynet` protocol is an intelligent financial service platform that provides open-source micro-banking services. In particular, `Evrynet` aims to create automated escrow services using an inter-operable smart contract platform that can be attractive to institutional investors by facilitating the creation and execution of micro-banking services to anyone at a competitive price. It is designed with necessary DEX support and the popular farming support which allows users to earn rewards by staking supported assets. Overall the protocol provides institutional investors an attractive environment for their assets and a place to potentially yield a high return.

The basic information of the `Evrynet` protocol is as follows:

Table 1.1: Basic Information of The `Evrynet` Protocol

Item	Description
Name	Evrynet Finance
Website	https://evrynet.io/
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	November 15, 2021

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

- https://github.com/Evry-Finance/evry-finance-amm-swap.git (35ce660)
- https://github.com/Evry-Finance/evry-finance-dmm-swap.git (144843b)
- https://github.com/Evry-Finance/evry-finance-farm.git (2d194cd)
- https://github.com/Evry-Finance/evry-finance-toolkit.git (4120ed4)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/Evry-Finance/evry-finance-amm-swap.git (a4a2322)
- https://github.com/Evry-Finance/evry-finance-dmm-swap.git (144843b)
- https://github.com/Evry-Finance/evry-finance-farm.git (573a14e)
- https://github.com/Evry-Finance/evry-finance-toolkit.git (166f5ba)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

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

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

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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Evrynet` protocol 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	1	■
Medium	1	■
Low	5	■■■■■
Informational	0	
Undetermined	1	■
Total	8	

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 high-severity vulnerability, 1 medium-severity vulnerability, 5 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key Evrynet Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improved Logic on Evry-Pair::swap()	Business Logic	Fixed
PVE-002	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-003	Low	Improved Sanity Checks For System Parameters	Coding Practices	Fixed
PVE-004	Low	Improved Logic of transferExceedAmount()	Business Logic	Fixed
PVE-005	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-006	Undetermined	Farm Incompatibility With Deflationary Tokens	Business Logic	Fixed
PVE-007	Low	Proper DMMPool Initialization On Liquidity	Business Logic	Confirmed
PVE-008	Medium	Proper Protocol Fee Calculation in DMMPool	Business Logics	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Logic on EvryPair::swap()

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `EvryPair`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

The `Evrynet` protocol has the built-in `DEX` functionality that is inspired from `UniswapV2`, but with the extension of flexible support for reconfigurable trading fee and protocol fee. Both fees can be dynamically configured via the `EvryFactory` contract on current pool pairs. In the analysis of the core `swap()` logic, we notice the current implementation needs to be improved.

To elaborate, we show below the `swap()` routine inside the the `evry-finance-amm-swap` repository. This function is designed to perform the actual swap between the related two tokens, i.e., `token0` and `token1`. Our analysis shows that the current logic can be improved in the following two aspects.

Firstly, the fee collection is only performed in one side, but not both. In particular, the current logic examines the input amount `amount0In`. If it is positive, the helper routine `sendFeeToPlatform()` is called to collect the `token0`-side protocol fee. Otherwise, the `token1`-side protocol fee will be collected. However, in a flashswap scenario, it is possible both `amount0In` and `amount1In` are positive, which means the protocol fee collection needs to be performed in both sides.

```
    // this low-level function should be called from a contract which performs important
        safety checks
    function swap(
            uint[2] memory amountOut,
            address to,
            address feeToPlatform,
            uint feePlatformBasis,
            uint feeLiquidityBasis,
            bytes calldata data
```

```
        )
            external
            lock
            override
        {

        FeeConfiguration memory feeConfiguration = FeeConfiguration({
            feeToPlatform: feeToPlatform,
            feePlatformBasis: feePlatformBasis,
            feeLiquidityBasis: feeLiquidityBasis,
            amount0Out: amountOut[0],
            amount1Out: amountOut[1]
        });

        require(feeConfiguration.amount0Out > 0  feeConfiguration.amount1Out > 0, 'Evry: INSUFFICIENT_OUTPUT_AMOUNT');

        (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
        require(feeConfiguration.amount0Out < _reserve0 && feeConfiguration.amount1Out < _reserve1, 'Evry: INSUFFICIENT_LIQUIDITY');

        uint balance0;
        uint balance1;
        { // scope for _token{0,1}, avoids stack too deep errors
            address _token0 = token0;
            address _token1 = token1;
            require(to != _token0 && to != _token1, 'Evry: INVALID_TO');
            if (feeConfiguration.amount0Out > 0) _safeTransfer(_token0, to, feeConfiguration.amount0Out); // optimistically transfer tokens
            if (feeConfiguration.amount1Out > 0) _safeTransfer(_token1, to, feeConfiguration.amount1Out); // optimistically transfer tokens
            if (data.length > 0) IEvryCallee(to).evryCall(msg.sender, feeConfiguration.amount0Out, feeConfiguration.amount1Out, data);
            balance0 = IERC20(_token0).balanceOf(address(this));
            balance1 = IERC20(_token1).balanceOf(address(this));
        }
        uint amount0In = balance0 > _reserve0 - feeConfiguration.amount0Out ? balance0 - (_reserve0 - feeConfiguration.amount0Out) : 0;
        uint amount1In = balance1 > _reserve1 - feeConfiguration.amount1Out ? balance1 - (_reserve1 - feeConfiguration.amount1Out) : 0;
        require(amount0In > 0  amount1In > 0, 'Evry: INSUFFICIENT_INPUT_AMOUNT');

        { // scope for reserve{0,1}Adjusted, avoids stack too deep errors
            uint totalFee = feeConfiguration.feePlatformBasis.add(feeConfiguration.feeLiquidityBasis);
            uint balance0Adjusted = balance0.mul(10000).sub(amount0In.mul(totalFee));
            uint balance1Adjusted = balance1.mul(10000).sub(amount1In.mul(totalFee));
            require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1).mul(10000**2), 'Evry: K');
        }

        _update(balance0, balance1);
```

```
        {
        // emit Swap(msg.sender, amount0In, amount1In, amountOut, to);
}
        if (amount0In > 0) {
            sendFeeToPlatform(token0, amount0In, feeConfiguration.feePlatformBasis,
                feeConfiguration.feeToPlatform);
        } else {
            sendFeeToPlatform(token1, amount1In, feeConfiguration.feePlatformBasis,
                feeConfiguration.feeToPlatform);
        }
        _sync();

    }
```

Listing 3.1: EvryPair::swap()

Secondly, the fee, including both trade fee and protocol fee, is collected according to the given input to the `swap()` function. However, the input cannot be trusted! In other words, the caller may intentionally craft the input to avoid paying any trade fee, which could seriously dis-incentivize the liquidity providers.

Recommendation Revise the above `swap()` routine to reliably collect trade fee and protocol fee.

Status This issue has been fixed in the following commit: 26f055c.

3.2 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [7]
- CWE subcategory: CWE-663 [2]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [12] exploit, and the recent `Uniswap/Lendf.Me` hack [11].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `ReleaseController` as an example, the `emergencyUnstake()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 64) starts before effecting the update on internal states (lines 66 – 67), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
49 function releaseToken() external onlyBeneficiary {
50   require(block.number >= nextReleaseBlock, "releaseToken: unable to claim token due to it is in a lock period");
51
52   if (releaseAmount > released) {
53     uint256 periodTimes = _getPeriodTimes();
54     require(periodTimes > 0, "releaseToken: unable to claim token due to it is not reach its distribution timeframe");
55
56     uint256 tokenAmount = releaseAmountPerPeriod.mul(periodTimes);
57     uint256 walletBalance = releaseAmount.sub(released);
58     uint256 releaseTokenAmount = tokenAmount;
59
60     if (walletBalance <= tokenAmount) {
61       releaseTokenAmount = walletBalance;
62     }
63
64     token.safeTransfer(beneficiary, releaseTokenAmount);
65
66     released = released.add(releaseTokenAmount);
67     nextReleaseBlock = nextReleaseBlock.add(blockPerPeriod.mul(periodTimes));
68
69     emit ReleaseToken(beneficiary, releaseTokenAmount, nextReleaseBlock);
70   }
71 }
```

Listing 3.2: `ReleaseController::releaseToken()`

Note that another routine also shares the same issue, i.e., `_distributeReward()` from the `PerformanceDistribution` contract.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status This issue has been fixed in the following commit: 166f5ba.

3.3 Improved Sanity Checks For System/Function Parameters

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: EvryFactory
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Evrynet protocol is no exception. Specifically, if we examine the EvryFactory and DMMFactory contracts, they have defined a number of protocol-wide risk parameters, e.g., feePlatformBasis and feeLiquidityBasis. In the following, we show an example routine that allows for their changes.

```
    function setFeeToPlatform(address _feeToPlatform) external onlyOwner override {
        feeToPlatform = _feeToPlatform;
    }

    function setPlatformFee(uint256 feeBasis) external onlyOwner override {
        feePlatformBasis = feeBasis;
    }

    function setLiquidityFee(uint256 feeBasis) external onlyOwner override {
        feeLiquidityBasis = feeBasis;
    }
```

Listing 3.3: An examplesetter in EvryFactory

Our result shows the update logic on the above parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large feeLiquidityBasis parameter will revert every single swap operation via EvryRouter.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

Status This issue has been fixed in the following commit: 26f055c.

3.4 Improved Logic of transferExceedAmount()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

The Evrynet protocol supports a number of smart contracts to manage the built-in Evry Toolkit. In particular, the evry-finance-toolkit contains the following three contracts to facilitate the distribution of specific tokens: ReleaseController, PerformanceDistribution, and TimeBasedDistribution. While these contracts work expectedly in their own logic, a specific function transferExceedAmount() can be improved. In the following, we use the ReleaseController contract as an example.

To illustrate, we show below the related transferExceedAmount() function. As the name indicates, it is designed to claim back exceed amount in the controller contract. While it works properly, it is possible that the intended beneficiary never claims the distributed/rewarded token. As a result, it will be helpful to specify a maximum time window that is expected for the beneficiary to claim. If the maximum time window has passed and the beneficiary still has not claimed yet, the owner can then claim back all remaining balance in the contract.

```
function transferExceedAmount(address _to) external onlyOwner {
  require(_to != address(0), "ReleaseController: cannot transfer exceed amount to zero
      address");
  uint256 totalBalance = token.balanceOf(address(this)).add(released);
  require(totalBalance > releaseAmount, "ReleaseController: balance is not exceed");
  token.safeTransfer(_to, totalBalance.sub(releaseAmount));
}
```

Listing 3.4: ReleaseController::transferExceedAmount()

The same issue is applicable to other two distribution contracts PerformanceDistribution and TimeBasedDistribution.

Recommendation Improve the above transferExceedAmount() function to eventually claim funds that are never claimed by the intended beneficiary. In addition, the same function can also be improved by recovering other unrelated tokens that may be accidentally sent to the contract.

Status This issue has been fixed in the following commit: 166f5ba.

3.5 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Farms
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier, the Evrynet protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via addPool() and the weights of supported pools can be adjusted via setPoolAllocation(). When analyzing the pool weight update routine setPoolAllocation (), we notice the need of timely updating all active pools to refresh the reward distribution (via massUpdatePools()) before the new pool weight becomes effective.

```
/// @notice Update the given pool's EVRYToken allocation point contract. Can only be called by the owner.
/// @param _pid The index of the pool. See `poolInfo`.
/// @param _allocPoint new AP of the pool
function setPoolAllocation(uint256 _pid, uint256 _allocPoint) external onlyOwner {
  updatePool(_pid);

  // Remove current AP value of pool _pid from total AP, then add new one.
  totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);

  // Replace old AP value with new one.
  poolInfo[_pid].allocPoint = _allocPoint;

  emit SetPoolAllocation(_pid, _allocPoint);
}
```

Listing 3.5: Farms:: setPoolAllocation ()

If the call to massUpdatePools() is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the onlyOwner modifier), which greatly alleviates the concern. Note other routines addPool() and setEvryPerBlock() share the same issue.

Recommendation Timely invoke massUpdatePools() when any pool's weight has been updated.

Status This issue has been fixed in the following commit: 573a14e.

3.6 Farms Incompatibility With Deflationary Tokens

- ID: PVE-006
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `Farms`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

In the `Evrynet` protocol, the `Farms` contract is designed to take users' assets and deliver rewards depending on their share. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract using the `safeTransfer()/safeTransferFrom()` routines to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
function deposit(
  address _for,
  uint256 pid,
  uint256 amount
) external nonReentrant {
  PoolInfo memory pool = updatePool(pid);
  UserInfo storage user = userInfo[pid][_for];

  // Validation
  if (user.fundedBy != address(0)) require(user.fundedBy == msg.sender, "Farms::
      deposit:: bad sof");

  // Effects
  _harvest(_for, pid);

  user.amount = user.amount.add(amount);
  user.rewardDebt = user.rewardDebt.add(amount.mul(pool.accEVRYPerShare) /
      ACC_EVRY_PRECISION);
  if (user.fundedBy == address(0)) user.fundedBy = msg.sender;

  // Interactions
  stakeTokens[pid].safeTransferFrom(msg.sender, address(this), amount);
  if (isEvryPool(pool.lpToken)) evrySupply = evrySupply.add(amount);

  emit Deposit(msg.sender, pid, amount, _for);
}

/// @notice Withdraw LP tokens.
```

```
/// @param _for Receiver of yield
/// @param pid The index of the pool. See ‘poolInfo‘.
/// @param amount of lp tokens to withdraw.
function withdraw(
  address _for,
  uint256 pid,
  uint256 amount
) external nonReentrant {
  PoolInfo memory pool = updatePool(pid);
  UserInfo storage user = userInfo[pid][_for];

  require(user.fundedBy == msg.sender, "Farms::withdraw:: only funder");
  require(user.amount >= amount, "Farms::withdraw:: amount exceeds");

  // Effects
  _harvest(_for, pid);

  user.rewardDebt = user.rewardDebt.sub(amount.mul(pool.accEVRYPerShare) /
      ACC_EVRY_PRECISION);
  user.amount = user.amount.sub(amount);
  if (user.amount == 0) user.fundedBy = address(0);

  // Interactions
  stakeTokens[pid].safeTransfer(msg.sender, amount);
  if (isEvryPool(pool.lpToken)) evrySupply = evrySupply.sub(amount);

  emit Withdraw(msg.sender, pid, amount, _for);
}
```

Listing 3.6: `Farms::deposit()/withdraw()`

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

Specially, if we take a look at the `updatePool()` routine, this routine calculates `pool.accEVRYPerShare` via dividing `evryReward` by `stakeTokenSupply`, where the `stakeTokenSupply` is derived from `stakeTokens[pid].balanceOf(address(this))` (line 232). Because the balance inconsistencies of the pool, the `stakeTokenSupply` could be 1 `Wei` and thus may yield a huge `pool.accEVRYPerShare` as the final result, which dramatically inflates the pool's reward.

```
225 function updatePool(uint256 pid) public returns (PoolInfo memory pool) {
226   pool = poolInfo[pid];
227   if (block.number > pool.lastRewardBlock) {
228     uint256 stakeTokenSupply;
229     if (isEvryPool(pool.lpToken)) {
```

```
        stakeTokenSupply = evrySupply;
      } else {
        stakeTokenSupply = stakeTokens[pid].balanceOf(address(this));
      }
      if (stakeTokenSupply > 0 && totalAllocPoint > 0) {
        uint256 blocks = block.number.sub(pool.lastRewardBlock);
        uint256 evryReward = (blocks.mul(evryPerBlock).mul(pool.allocPoint)).div(
            totalAllocPoint);

        evryDistributor.release(evryReward);

        pool.accEVRYPerShare = pool.accEVRYPerShare.add((evryReward.mul(
            ACC_EVRY_PRECISION)).div(stakeTokenSupply));
      }
      pool.lastRewardBlock = block.number;
      poolInfo[pid] = pool;
      emit UpdatePool(pid, pool.lastRewardBlock, stakeTokenSupply, pool.accEVRYPerShare)
          ;
    }
  }
```

Listing 3.7: `Farms::updatePool()`

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `Evrynet` for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

Status This issue has been fixed in the following commit: 573a14e.

3.7 Proper DMMPool Initialization On Liquidity

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `DMMPool`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

The `Evrynet` protocol also has the built-in `Dynamic Automated Market Making (DMM)`-based `DEX` functionality. The `DMM` is a generalized model for optimizing token pools with the goal of greatly improving capital efficiency and mitigating impermanent loss. This model uses three simple mechanisms, namely price curve amplification, custom base fees and dynamic fees based on volume to achieve this optimization. In the analysis of the core `DMM` logic, we notice the current pool initialization logic can be improved.

Specifically, we show below the related `mint()` routine. The issue occurs when the pool liquidity is added at the first time. It comes to our attention that the liquidity is calculated as `liquidity = MathExt.sqrt(amount0.mul(amount1))` (line 97) without considering the price curve amplification, which is not consistent with the proposed core `DMM` logic. In other words, the first-time liquidity may be calculated as `liquidity = MathExt.sqrt(amount0.mul(amount1)).mul(_ampBps).div(BPS)`.

```
81      function mint(address to) external override nonReentrant returns (uint256 liquidity) {
82          (bool isAmpPool, ReserveData memory data) = getReservesData();
83          ReserveData memory _data;
84          _data.reserve0 = token0.balanceOf(address(this));
85          _data.reserve1 = token1.balanceOf(address(this));
86          uint256 amount0 = _data.reserve0.sub(data.reserve0);
87          uint256 amount1 = _data.reserve1.sub(data.reserve1);
88
89          bool feeOn = _mintFee(isAmpPool, data);
90          uint256 _totalSupply = totalSupply(); // gas savings, must be defined here since totalSupply can update in _mintFee
91          if (_totalSupply == 0) {
92              if (isAmpPool) {
93                  uint32 _ampBps = ampBps;
94                  _data.vReserve0 = _data.reserve0.mul(_ampBps) / BPS;
95                  _data.vReserve1 = _data.reserve1.mul(_ampBps) / BPS;
96              }
97              liquidity = MathExt.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
98              _mint(address(-1), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens
99          } else {
100             liquidity = Math.min(
101                 amount0.mul(_totalSupply) / data.reserve0,
```

```
                amount1.mul(_totalSupply) / data.reserve1
            );
            if (isAmpPool) {
                uint256 b = liquidity.add(_totalSupply);
                _data.vReserve0 = Math.max(data.vReserve0.mul(b) / _totalSupply, _data.
                    reserve0);
                _data.vReserve1 = Math.max(data.vReserve1.mul(b) / _totalSupply, _data.
                    reserve1);
            }
        }
        require(liquidity > 0, "DMM: INSUFFICIENT_LIQUIDITY_MINTED");
        _mint(to, liquidity);

        _update(isAmpPool, _data);
        if (feeOn) kLast = getK(isAmpPool, _data);
        emit Mint(msg.sender, amount0, amount1);
    }
```

Listing 3.8: DMMPool::mint()

Recommendation Calculate the initial liquidity amount based on the virtual reserves, instead of real reserves.

Status This issue has been confirmed.

3.8 Proper Protocol Fee Calculation in DMMPool

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `DMMPool`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier in Section 3.7, the `Evrynet` protocol has the built-in `Dynamic Automated Market Making (DMM)`-based `DEX` functionality. And the provided `DEX` is customized with a reconfigurable trade fee and protocol fee. While examining the protocol fee extraction, we notice the current implementation is inconsistent with the intended protocol fee percentage `governmentFeeBps`.

```
    /// @dev if fee is on, mint liquidity equivalent to configured fee of the growth in
        sqrt(k)
    function _mintFee(bool isAmpPool, ReserveData memory data) internal returns (bool
        feeOn) {
        (address feeTo, uint16 governmentFeeBps,) = factory.getFeeConfiguration();
        feeOn = (feeTo != address(0) && governmentFeeBps != 0);
        uint256 _kLast = kLast; // gas savings
```

```
        if (feeOn) {
            if (_kLast != 0) {
                uint256 rootK = MathExt.sqrt(getK(isAmpPool, data));
                uint256 rootKLast = MathExt.sqrt(_kLast);
                if (rootK > rootKLast) {
                    uint256 numerator = totalSupply().mul(rootK.sub(rootKLast)).mul(
                        governmentFeeBps
                    );
                    uint256 denominator = rootK.add(rootKLast).mul(5000);
                    uint256 liquidity = numerator / denominator;
                    if (liquidity > 0) _mint(feeTo, liquidity);
                }
            }
        } else if (_kLast != 0) {
            kLast = 0;
        }
    }
```

Listing 3.9: `DMMPool::_mintFee()`

To elaborate, we show above the `_mintFee()` routine inside the the `evry-finance-dmm-swap` repository. Note the trade fee collection at the time of the trade would impose an additional gas cost on every trade. To avoid this, accumulated fees are collected only when liquidity is deposited or withdrawn. The contract computes the accumulated fees, and mints new liquidity tokens to the fee beneficiary, immediately before any tokens are minted or burned (via the above `_mintFee()` routine). It comes to our attention the accumulated fees should be computed as $\frac{\sqrt{k_2}-\sqrt{k_1}}{(\frac{1}{fee}-1)\cdot\sqrt{k_2}+\sqrt{k_1}}\cdot totalSupply()$, i.e., $\frac{(\sqrt{k_2}-\sqrt{k_1})\cdot governmentFeeBps}{(BPS-governmentFeeBps)\cdot\sqrt{k_2}+\sqrt{k_1}\cdot governmentFeeBps}\cdot totalSupply()$.

Recommendation Revise the above `_mintFee()` function to properly collect the protocol fee. An example revision is shown as follows:

```
    /// @dev if fee is on, mint liquidity equivalent to configured fee of the growth in
        sqrt(k)
    function _mintFee(bool isAmpPool, ReserveData memory data) internal returns (bool
        feeOn) {
        (address feeTo, uint16 governmentFeeBps,) = factory.getFeeConfiguration();
        feeOn = (feeTo != address(0) && governmentFeeBps != 0);
        uint256 _kLast = kLast; // gas savings
        if (feeOn) {
            if (_kLast != 0) {
                uint256 rootK = MathExt.sqrt(getK(isAmpPool, data));
                uint256 rootKLast = MathExt.sqrt(_kLast);
                if (rootK > rootKLast) {
                    uint256 numerator = totalSupply().mul(rootK.sub(rootKLast)).mul(
                        governmentFeeBps
                    );
                    uint256 denominator = rootK.mul(BPS.sub(governmentFeeBps)).add(
                        rootKLast.mul(governmentFeeBps));
                    uint256 liquidity = numerator / denominator;
```

```
                    if (liquidity > 0) _mint(feeTo, liquidity);
                }
            }
        } else if (_kLast != 0) {
            kLast = 0;
        }
    }
```

Listing 3.10: Revised `DMMPool::_mintFee()`

Status This issue has been confirmed.

4 | Conclusion

In this audit, we have analyzed the `Evrynet` protocol design and implementation. The `Evrynet` protocol provides an intelligent financial service platform that provides open-source micro-banking services. The protocol is designed with necessary DEX support and the popular farming support which allows users to earn rewards by staking supported assets. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[3] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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 CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.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. PeckShield Inc. https://www.peckshield.com.

[11] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[12] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.