shieldify

Pear Protocol V1

SECURITY REVIEW

Date: 3 Jul 2024

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Pear Protocol V1

Pear Protocol represents an innovative solution designed to streamline and enhance the efficiency of on-chain pairs trading. It enables users to execute leveraged long and short positions within a single transaction, addressing the complexities and inefficiencies traditionally associated with pair trading in cryptocurrencies. By integrating a variety of on-chain trading engines alongside a dedicated user interface and experience, Pear Protocol simplifies the process of initiating simultaneous long and short positions in correlated assets, such as going long on BTC while shorting ETH with leverage. This liquidity-agnostic platform offers deep liquidity access, and flexibility in managing trading parameters, and mitigates the custody and trust issues found in centralized exchanges by ensuring traders retain asset custody. Beyond simplifying trading executions, Pear Protocol extends its utility with a tokenized trading system, allowing trading positions to be represented as ERC-721 tokens for increased composability within DeFi ecosystems. With features emphasizing simplicity, flexibility, scalability, optionality, and ease of use, Pear Protocol aims to revolutionize pair trading, making it more accessible and efficient while fostering broader decentralized finance trading solutions adoption.

Learn more about Pear's concept and the technicalities behind it here.

3.1 Observations

- While Pear plans to integrate with multiple trading engines, like GMX, Vertex, and SYMM, this security review is concerned mostly with the GMX part. The only other engine used during this review was Vertex.
- The main components of the system are GMXFactory and GMXAdapter, responsible for communication with GMX and position management;
- The PlatformLogic contract is a central component in the management of fees within the Pear protocol. In addition to its core fee management functions, PlatformLogic is also tasked with calculating and tracking various discounts. This includes referral discounts, which incentivize users to bring in new traders, and stakers discounts, which reward users for staking their tokens. Through these mechanisms, PlatformLogic ensures a fair and efficient fee structure that benefits both the protocol and its users.

- The Comptroller contract contains important config details like: GMX Exchange Router, PlatformLogic, admin, Referral code, etc. This central contract acts as the main configuration hub, providing essential references and parameters required by other components of the protocol.

3.2 Privileged Roles and Actors

- `GMXFactory.sol` – creates new pair positions (adapters) and implements functionalities to manage them
- `GMXAdapter.sol` – GMX pair position adapter. An interface between GMX positions and position owners.
- `NFTPosition.sol` – allows for position tokenization and making it transferrable.
- `Admin` – has powers to set all the Pear protocol fees, and referrals and upgrade the protocol
- `Position owner` – user of the protocol. Are owners of GMXAdapter owning GMX position, hence owners of the positions.
- `Stakers and Treasury` – fees recipient.
- `ComptrollerManager` – contract provides mechanisms for managing a comptroller reference within contracts. This design ensures a modular and extensible architecture, allowing new components to integrate with the existing system seamlessly.
- `PearStaker` – Manages staking of PEAR tokens, calculates rewards, and applies exit-fees for unstaking early.
- `FeeRebateManager` – Manages fee rebates and discounts for users based on trading volumes and PEAR token stakes.

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 21 days with a total of 504 hours dedicated to the audit by the Shieldify team.

Overall, the code is well-written. The audit report identified two high-severity issues related to referrer fees and reward handling, a medium-severity issue concerning the incorrect implementation of the admin setter function, and several other potential vulnerabilities related to the oracle integration, along with other findings of lower severity.

This is the second security review that Shieldify has conducted on Pear protocol, with this audit focused primarily on the newly-added functionality.

5.1 Protocol Summary

Project Name	**Pear Protocol**
Repository	pear-gmx-v2-core
Type of Project	DEX, On-Chain Pairs Trading
Audit Timeline	21 days
Review Commit Hash	f3329d0474013d60d183a5773093b94a9e55caae
Fixes Review Commit Hash	a8c0982974a8f1c77df0ca0ed4468f15b4e3443d

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
src/interfaces/order/OracleUtils.sol	36
src/interfaces/order/IBaseOrderUtils.sol	30
src/interfaces/order/ArbSys.sol	3
src/interfaces/order/Chain.sol	26
src/interfaces/order/IOrderVault.sol	3
src/interfaces/IReader.sol	29
src/interfaces/order/Order.sol	178
src/interfaces/IDataStore.sol	3
src/interfaces/IOrderHandler.sol	5
src/interfaces/IComptroller.sol	23

src/interfaces/IExchangeRouter.sol	4
src/interfaces/IPearGmxFactoryV2.sol	5
src/interfaces/IChainLinkAggregator.sol	3
src/interfaces/IPearGmxAdapterV2.sol	28
src/interfaces/IFeeRebateManager.sol	16
src/interfaces/IAcceptComptroller.sol	3
src/interfaces/IPearStaker.sol	4
src/interfaces/IPearBase.sol	46
src/interfaces/IPlatformLogic.sol	73
src/interfaces/IPositionNFT.sol	15
src/interfaces/IRewardsClaimer.sol	19
src/PlatformLogic.sol	569
src/helpers/PearGmxHelper.sol	6
src/helpers/ComptrollerManager.sol	32
src/helpers/GMXOrderCallbackReceiver.sol	69
src/staker/PearStaker.sol	204
src/staker/PearToken.sol	11
src/Comptroller.sol	222
src/GmxAdapterV2.sol	98
src/rewards/RewardsClaimer.sol	219
src/rewards/FeeRebateManager.sol	136
src/NFT/PositionNFT.sol	169
src/libraries/Events.sol	79
src/libraries/Errors.sol	110
src/libraries/EventUtils.sol	201
src/NFT/ERC721Upgradeable.sol	223
src/factory/GmxFactoryV2.sol	574
src/NFT/lib/UpgradableUtils.sol	475
Total	**3949**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical** and **High** issues: **2**
- **Medium** issues: **5**
- **Low** issues: **2**

ID	Title	Severity	Status
[H-01]	User May Pay Smaller Fees, Get No Refund, Or Force Treasury and Stakers to Not Receive Fees at All	High	Fixed
[H-02]	The Amount of The Pending Referrer Fee is Overwritten in `_applyPlatformFeeETHGmx()`	High	Fixed
[M-01]	`setMintPositionFee()` Function Almost Always Revert	Medium	Fixed
[M-02]	Missing Check For Active L2 Sequencer in `calculateArbAmount()`	Medium	Fixed
[M-03]	Oracle Will Return the Wrong Price for Asset if Underlying Aggregator Hits `minAnswer`	Medium	Acknowledged
[M-04]	Missing Stale Price Check in `convertFeeToEth()` and `calculateArbAmount()` Functions	Medium	Acknowledged
[M-05]	`RewardsClaimer` Uses Transfer Functions that Do Not Check The Return Value	Medium	Fixed
[L-01]	Use the `setRewardTokenDecimals()` Function Carefully	Low	Acknowledged
[L-02]	The `setPearTreasury()` Function Lacks the `notZeroAddress` Modifier	Low	Fixed

7. Findings

[H-01] If the User's Pending Rewards Exceed `maxClaimPerWallet` the Remaining Reward Will be Reset to Zero

Severity

High Risk

Description

The `RewardsClaimer.sol` contract manages the distribution and claiming of rewards. There we have `maxClaimPerWallet`, which limits how many tokens users can claim. It is important to note that the variable is not a constant and can be changed by the admin.

```
function setMaxClaimPerWallet(uint128 _maxClaimPerWallet) external
    override onlyOwner {
     maxClaimPerWallet = _maxClaimPerWallet;
}
```

The variable is used as a check when a user calls `claimRewards()`. This function allows users to claim their pending rewards from a specified platform.

First, the `claimRewards()` retrieves the user's pending rewards for the specified platform and the total claimed rewards across all platforms:

```
uint256 _pendingRewards = userPendingRewards[msg.sender][_platform];
uint256 _claimedRewards =
   userClaimedRewardsAcrossAllPlatforms[msg.sender];

uint256 _totalClaimed = _pendingRewards + _claimedRewards;
```

After this, If the total claimed rewards are greater than the maximum claim per wallet, then the pending rewards are the maximum claim per wallet minus the claimed rewards.

```
if (_totalClaimed > maxClaimPerWallet) {
    _pendingRewards = maxClaimPerWallet - _claimedRewards;
}
```

That means if a user needs to receive 500 tokens, but the maximum allowed is only 200, then the remainder will be lost because the `userPendingRewards` mapping is reset below.

```
userPendingRewards[msg.sender][_platform] = 0;
```

Scenario

Let's review the following situation:

- Alice has to receive 1000 tokens.
- However, `maxClaimPerWallet` is only 500.
- Alice gets only 500 tokens, and the others are lost.
- After 1 hour, the admin decides to increase `maxClaimPerWallet` to 1000 tokens.
- Bob has exactly the same situation as Alice. He has to get 1000 tokens, and he gets them all. In the same situation, Bob will get 500 more tokens than Alice.

Impact

A user may lose their deserved rewards.

Location of Affected Code

File: src/rewards/RewardsClaimer.sol#L229

Recommendation

Instead of resetting the `userPendingRewards`, a better idea is to calculate how many tokens a user has claimed and keep his rewards in case you increase the maximum allowed reward per wallet in the future.

Team Response

Fixed as proposed.

[H-02] The Amount of The Pending Referrer Fee is Overwritten in `_applyPlatformFeeETHGmx()`

Severity

High Risk

Description

The `_applyPlatformFeeETHGmx()` function is an internal function that calculates and applies platform fees in ETH for a given transaction. This function is called by `GmxFactory` and also considers staked fee discounts, referral discounts, and referrer withdrawals.

If the `_referee` (the user) has been referred by someone, we enter the following code block:

```
function _applyPlatformFeeETHGmx(address adapter, address _referee,
    uint256 _grossAmount, address _factory) internal returns (uint256,
    uint256) {
// code
     if (referredUsers[_referee] != 0) {
          uint256 _refereeDiscount =
          calculateFees(_feeAmount, refereeDiscount);

          _feeAmount -= _refereeDiscount;

// calculate the referrer discount
// for testing the referralFee is set to 5 bps
          uint256 _referrerWithdrawal = calculateFees(_feeAmount,
    referrerFee);

          bool success = IPearGmxFactoryV2(_factory).
    feeTransferPlatformLogic(_feeAmount);
          if (!success) {
               revert Errors.PlatformLogic_NotEnoughBalance();
          }

          setPendingReferrerFeeAmount(
          adapter, checkReferredUser(_referee), _referrerWithdrawal
          );

          uint256 _grossAmountAfterFee = _grossAmount - _feeAmountInUsd;

          emit FeesPaid(_referee, _feeAmount, _grossAmountAfterFee);

// remove the discounted % referee withdrawal
          _feeAmount -= _referrerWithdrawal;

          return (_feeAmount, _referrerWithdrawal);
// code
     }
```

First, we calculate the discount fee:

```
uint256 _refereeDiscount = calculateFees(_feeAmount, refereeDiscount);
```

After this, we calculate the referrer fee:

```
uint256 _referrerWithdrawal = calculateFees(_feeAmount, referrerFee);
```

This fee should be given to the referrer. This is handled by the `setPendingReferrerFeeAmount()` function:

```
setPendingReferrerFeeAmount(adapter, checkReferredUser(_referee),
   _referrerWithdrawal);
```

```
function setPendingReferrerFeeAmount(address adapterAddress, address
   referrer, uint256 amount) internal {
    pendingReferrerFeeAmounts[adapterAddress][referrer] = amount;
    emit SetPendingReferrerFeeAmount(adapterAddress, referrer, amount);
}
```

The `setPendingReferrerFeeAmount()` function should add the referrer fee to the existing pending referrer fee amount. But instead, it rewrites them. This will remove all previous pending fees of the referrer.

Impact

All pending fees of a user are deleted.

Location of Affected Code

File: src/PlatformLogic.sol#L689

Recommendation

To ensure the referrer fee does not overwrite previous values, we should add `_referrerWithdrawal` /newline to the existing value of `pendingReferrerFeeAmounts[adapterAddress][referrer]` mapping.

Team Response

Fixed as proposed.

[M-01] `setMintPositionFee()` Function Almost Always Revert

Severity

Medium Risk

Description

The `setMintPositionFee()` function can set the `mintFeeInUsdc` value and can only be called by the admin. The problem is that the value of `mintFeeInUsdc` is too big:

```
mintFeeInUsdc = 1_000_000;
```

The variable has 6 decimals because the USDC token has 6 decimals. But this causes a problem in the `notExceedingOneHundredPercent()` modifier.

If the protocol wants to add an approximate value for some reason, the function will revert because of the `notExceedingOneHundredPercent` modifier:

```
modifier notExceedingOneHundredPercent(uint256 _bps) {
    if (_bps > TOTAL_BASIS_POINTS) {
        revert Errors.PlatformLogic_ExceedingBps();
    }
    _;
}
```

```
uint256 constant TOTAL_BASIS_POINTS = 10_000;
```

We can see that `TOTAL_BASIS_POINTS` is 10 000 and `_bps` will be a value with six decimals, and because of this check, `setMintPositionFee()` will almost always revert.

Location of Affected Code

File: src/PlatformLogic.sol#L365

```
/// @inheritdoc IPlatformLogic
function setMintPositionFee(uint256 _fee)
  external
  override
  onlyAdmin
  notExceedingOneHundredPercent(_fee)
{
  emit MintFeeChanged(mintFeeInUsdc, _fee);
  mintFeeInUsdc = _fee;
}
```

Recommendation

If you want the admin to have control over `mintFeeInUsdc` you need to remove the modifier from the function.

Team Response

Fixed as proposed.

[M-02] Missing Check For Active L2 Sequencer in `calculateArbAmount()`

Severity

Medium Risk

Description

The `calculateArbAmount()` function converts an amount of ETH (in Wei) to its equivalent amount in ARB tokens, using the latest exchange rates for ETH/USD and ARB/USD provided by Chainlink price feeds:

```
function calculateArbAmount(uint256 _ethAmountInWei) internal view
    returns (uint256) {
// Fetch the ETH to USD price with 8 decimal places
    (, int256 ethUsdPrice,,,) = IChainLinkAggregator(
        comptroller.getEthUsdAggregator()
    ).latestRoundData();

// Fetch the ARB to USD price with 8 decimal places
    (, int256 arbUsdPrice,,,) =
        IChainLinkAggregator(arbUsdAggregator).latestRoundData();

// Convert the ETH amount to USD
// _ethAmountInWei is in wei, ethUsdPrice has 8 decimal places.
// We multiply by 10^10 to adjust the final result to have 18 decimal
// places for ARB.
    uint256 usdAmount =
        (_ethAmountInWei * uint256(ethUsdPrice)) / rewardTokenDecimals;

// Convert the USD amount to ARB
// usdAmount is in USD with 18 decimals, arbUsdPrice has 8 decimal
// places.
// The result is in ARB's smallest unit (like wei for ETH).
    uint256 arbAmount =
        (usdAmount * rewardTokenDecimals) / uint256(arbUsdPrice);

    return arbAmount;
}
```

This function does not check if the Sequencer is active.

Optimistic rollup protocols move all execution of the layer 1 (L1) Ethereum chain, complete execution on a layer 2 (L2) chain, and return the results of the L2 execution back to the L1. These protocols have a sequencer that executes and rolls up the L2 transactions by batching multiple transactions into a single transaction.

If a sequencer becomes unavailable, it is impossible to access read/write APIs that consumers are using and applications on the L2 network will be down for most users without interacting directly through the L1 optimistic rollup contracts. The L2 has not stopped, but it would be unfair to continue providing service on your applications when only a few users can use them.

Impact

If the Arbitrum Sequencer goes down, oracle data will not be kept up to date, and the price could become stale.

Location of Affected Code

File: src/rewards/RewardsClaimer.sol#L414

Recommendation

Consider implementing logic to revert the execution when the price cannot be fetched. Check this example here.

Team Response

Fixed by adding a fallback function with 24-hour base price caching. This decision is made with the awareness of the risk of using an outdated price.

[M-03] Oracle Will Return the Wrong Price for Asset if Underlying Aggregator Hits `minAnswer`

Severity

Medium Risk

Description

`convertFeeToEth()` in `PlatformLogic.sol` and `calculateArbAmount()` in `RewardsClaimer.sol` will return the wrong price for the asset if the underlying aggregator hits `minAnswer`.

Chainlink's aggregators have a built-in circuit breaker if the price of an asset goes outside of a predetermined price band. The result is that if an asset experiences a huge drop in value (i.e. LUNA crash), the price of the oracle will continue to return the `minPrice` instead of the actual price of the asset. This would allow users to continue borrowing with the asset but at the wrong price. This is exactly what happened to Venus on BSC when LUNA imploded.

When `latestRoundData()` is called, it requests data from the aggregator. The aggregator has a `minPrice` and a `maxPrice`. If the price falls below the `minPrice` instead of reverting, it will just return the min price.

Impact

In the event that an asset crashes, the price can be manipulated.

Location of Affected Code

File: src/PlatformLogic.sol#L486

File: src/rewards/RewardsClaimer.sol#L414

Recommendation

The `convertFeeToEth()` and `calculateArbAmount()` functions should check the returned answer against the `minPrice/maxPrice` and revert if the answer is outside of the bounds.

Team Response

Acknowledged.

[M-04] Missing Stale Price Check in `convertFeeToEth()` and `calculateArbAmount()` Functions

Severity

Medium Risk

Description

The functions `convertFeeToEth()` in `PlatformLogic.sol` and `calculateArbAmount()` in `RewardsClaimer.sol` do not check if the price returned by the oracle is stale. We will give an example with `convertFeeToEth()`, but the same applies to the other function.

This is `convertFeeToEth()`:

```
function convertFeeToEth(uint256 feeAmount) public view returns (uint256)
    {
    IChainLinkAggregator ethUsdAggregator =
        IChainLinkAggregator(comptroller.getEthUsdAggregator());
    (, int256 ethUsdPrice,,,) = ethUsdAggregator.latestRoundData();
// contain 8 decimals
    uint256 feeInEth = (feeAmount * 1e20) / uint256(ethUsdPrice);
    return feeInEth;
}
```

This function converts a fee amount denominated in USD to its equivalent value in Ether, based on the latest ETH/USD exchange rate provided by a Chainlink Aggregator.

According to Chainlink's documentation, this function does not error if no answer has been reached but returns 0 or outdated round data. So we need to check that `ethUsdPrice` is not 0.

The other problem is that there is no check for `Heartbeat`. This function fetches the current price of ETH in USD using Chainlink's ETH/USD feed, but there is no check for Heartbeat. Here as a reference, you can see that there should be a check every hour here.

Impact

If these checks are missing the price that is retrieved from the oracle may be outdated.

Location of Affected Code

File: src/PlatformLogic.sol#L486

File: src/rewards/RewardsClaimer.sol#L414

Recommendation

You need to check the return value of `ethUsdPrice` and `arbUsdPrice` and the maximum delay accepted between answers from Chainlink.

Team Response

Acknowledged.

[M-05] `RewardsClaimer` Uses Transfer Functions that Do Not Check The Return Value

Severity

Medium Risk

Description

The `RewardsClaimer.sol` contract manages the distribution and claiming of rewards. In several places in the contract, the `transfer()` function is used when reward tokens are to be transferred, which does not check the returned value.

```
IERC20(rewardToken).transfer(msg.sender, _pendingRewards);
IERC20(rewardToken).transferFrom(
msg.sender, address(this), _totalAmount
IERC20(rewardToken).transfer(_to, _amount);
IERC20(_token).transfer(_to, _amount);
```

The `ERC20.transfer()` and `ERC20.transferFrom()` functions return a bool value indicating success. This parameter needs to be checked for success.

Impact

If the return value is not checked, the transfer may not be successful. However, the functions in `RewardsClaimer.sol` will still be considered a successful transfer.

Location of Affected Code

File: src/rewards/RewardsClaimer.sol

Recommendation

It is recommended to use OpenZeppelin's SafeERC20 versions with the `safeTransfer()` and `safeTransferFrom()` functions that handle the return value check.

Team Response

Fixed as proposed.

[L-01] Use the `setRewardTokenDecimals()` Function Carefully

Severity

Low Risk

Description

In `RewardsClaimer.sol`, we have the `setRewardTokenDecimals()` function:

```
function setRewardTokenDecimals(uint128 _rewardTokenDecimals) external
    override onlyOwner {
     rewardTokenDecimals = _rewardTokenDecimals;
}
```

This function allows the contract owner to update the decimals of the reward token. You have to be very careful when changing the reward token decimal. They are used in price calculation and can lead to a critical vulnerability.

Impact

Decimals are used in price calculation, and you need to track changes very carefully.

Location of Affected Code

File: src/rewards/RewardsClaimer.sol#L356

Recommendation

Before changing the decimals of a reward token, check carefully for any side effects.

Team Response

Acknowledged.

[L-02] The `setPearTreasury()` Function Lacks the `notZeroAddress` Modifier

Severity

Low Risk

Description

The protocol has a `notZeroAddress()` modifier, which has it on all setters functions on which the address changes.

However, this modifier is missing on the `setPearTreasury()` function in `PlatformLogic.sol`.

```
function setPearTreasury(address payable _newTreasury) external override
    onlyAdmin {
     emit PearTreasuryChanged(PearTreasury, _newTreasury);
     PearTreasury = _newTreasury;
}
```

Impact

The impact is low because even if, by mistake, the admin sets `_newTreasury` to a zero address, he can then change it.

Location of Affected Code

File: src/PlatformLogic.sol#L387

Recommendation

Add the `notZeroAddress` modifier to `setPearTreasury()`.

Team Response

Fixed as proposed.

shieldify

Thank you!