VERILOG

2023

PIREX-GMX: SECURITY REVIEW REPORT

Feb 03, 2023

Contents

1 | Introduction

Figure 1.1: Pirex Report Cover

This report presents our engineering engagement with the Redacted-Cartel team on their integration with the GMX Protocol.

Project Name	**Pirex-GMX**
Repository Link	https://github.com/redacted-cartel/pirex-gmx
First Commit Hash	First: 3aadd22;
Final Commit Hash	Final: d2b1c65;
Language	Solidity
Chain	Arbitrum

2 | About Verilog Solutions

Founded by a group of cryptography researchers and smart contract engineers in North America, Verilog Solutions elevates the security standards for Web3 ecosystems by being a full-stack Web3 security firm covering smart contract security, consensus security, and operational security for Web3 projects.
Verilog Solutions team works closely with major ecosystems and Web3 projects and applies a quality above quantity approach with a continuous security model. Verilog Solutions onboards the best and most innovative projects and provides the best-in-class advisory services on security needs, including on-chain and off-chain components.

3 | Service Scope

3.1 | Service Stages

Our auditing service includes the following two stages:

- Pre-Audit Consulting Service
- Smart Contract Auditing Service

3.1.1 | Pre-Audit Consulting Service

As a part of the audit service, the Verilog Solutions team worked closely with the Redacted-Cartel development team to discuss potential vulnerability and smart contract development best practices in a timely fashion. Verilog Solutions team is very appreciative of establishing an efficient and effective communication channel with the Redacted-Cartel team, as new findings were exchanged promptly and fixes were deployed quickly, during the preliminary report stage.

3.1.2 | Smart Contract Auditing Service

The Verilog Solutions team analyzed the entire project using a detailed-oriented approach to capture the fundamental logic and suggested improvements to the existing code. Details can be found under Findings And Improvement Suggestions.

3.2 | Methodology

- **Code Assessment**
  - We evaluate the overall quality of the code and comments as well as the architecture of the repository.
  - We help the project dev team improve the overall quality of the repository by providing suggestions on refactorization to follow the best practices of Web3 software engineering.
- **Code Logic Analysis**
  - We dive into the data structures and algorithms in the repository and provide suggestions to improve the data structures and algorithms for the lower time and space complexities.
  - We analyze the hierarchy among multiple modules and the relations among the source code files in the repository and provide suggestions to improve the code architecture with better readability, reusability, and extensibility.
- **Business Logic Analysis**
  - We study the technical whitepaper and other documents of the project and compare its specifications with the functionality implemented in the code for any potential mismatch between them.
  - We analyze the risks and potential vulnerabilities in the business logic and make suggestions to improve the robustness of the project.
- **Access Control Analysis**
  - We perform a comprehensive assessment of the special roles of the project, including their authorities and privileges.
  - We provide suggestions regarding the best practice of privilege role management according to the standard operating procedures (SOP).
- **Off-Chain Components Analysis**
  - We analyze the off-chain modules that are interacting with the on-chain functionalities and provide suggestions according to the SOP.
  - We conduct a comprehensive investigation for potential risks and hacks that may happen on the off-chain components and provide suggestions for patches.

3.3 | Audit Scope

Our auditing for Redacted-Cartel covered the Solidity smart contracts under the folder ‘contracts‘ in the repository(https://github.com/redacted-cartel/pirex-gmx) with commit hash ‘3aadd22‘.

4 | Project Summary

Pirex-GMX aims to integrate GMX into Pirex by providing users with a way to tokenize their GMX, GLP, and esGMX tokens. This protocol consists of the "Standard Mode", which allows users to tokenize their assets for receiving rewards, and the "Easy Mode", which is built on top of the "Standard Mode" and enables auto compounding of the rewards that are being received. Since GMX has three different tokens, the Pirex strategies vary as follows:

- GLP: Consists of an asset index used for swaps and leverage trading.

  GLP will be tokenized as pxGLP, effectively making GLP liquid. pxGLP can be redeemed for ETH/AVAX or other GLP assets.

- GMX: Platform's utility and governance token. GMX can be staked for ETH/AVAX rewards, esGMX, and multiplier points.

  GMX will be tokenized as pxGMX, becoming a liquid form of GMX. All GMX will be permanently staked.

- esGMX: Escrowed GMX is a non-transferrable GMX token staked for GMX tokens or vested to become GMX over one year.

  esGMX will be tokenized as pxGMX, becoming a liquid form of esGMX. All esGMX will be permanently staked.

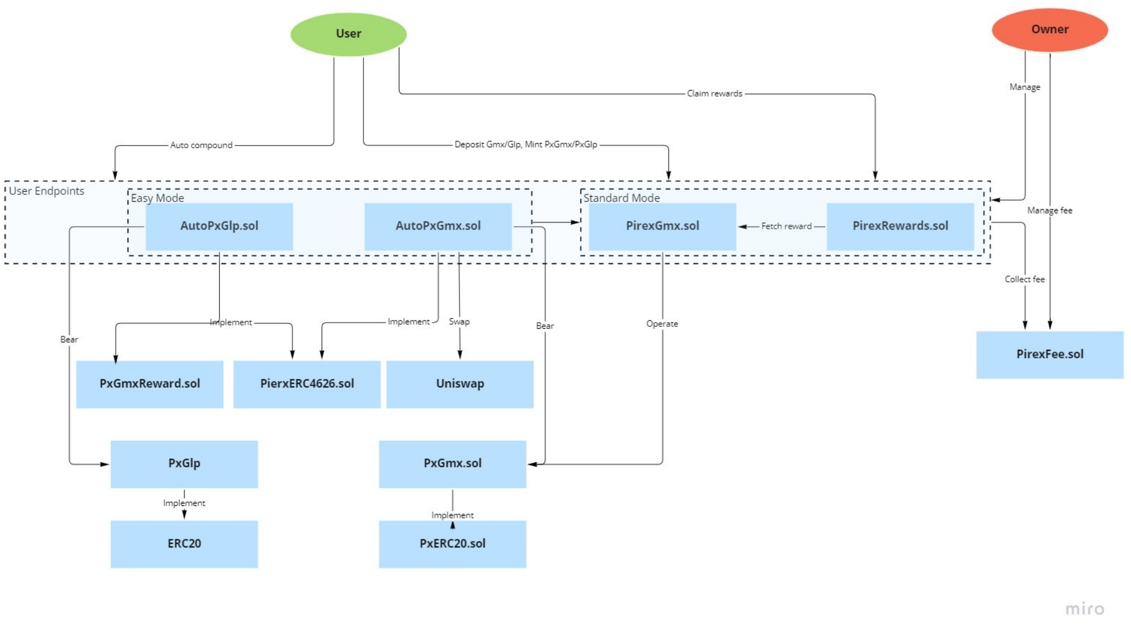

Figure 4.1: Pirex Architecture

Pirex-GMX "Standard Mode" mainly consists of four smart contracts with the following logic blocks:

4.1 | PirexGmx.sol

PirexGmx.sol contains the logic for interacting with the three GMX tokens. This contract mainly has the following functionality:

- Set fees for different operations.
- Set external addresses for the contracts that the protocol will be interacting with.
- Deposit GMX for pxGMX.

- Deposit fsGLP for pxGLP.
- Redeem pxGLP for GLP or ETH.
- Delegation vote logic.
- Emergency/migration logic.

4.2 | PirexRewards.sol

PirexRewards.sol contains the logic for handling the rewards in the "Standard Mode". This contract mainly has the following functionalities:

- Set addresses of producers (pxGLP and pxGMX).
- Set/Unset reward recipients for reward tokens.
- Add/Remove reward tokens.
- Update global and user accrual rewards.
- Harvest and claim rewards.
- Set/Unset reward recipient for privileged method.

4.3 | PxERC20.sol

PxERC20.sol contains the logic for interacting with the PirexGmx token. This contract mainly has the following functionalities:

- ERC20 functionality while updating the reward accrual state.

4.4 | PirexFees.sol

PirexFees.sol contains the logic for handling the fees obtained by the transactions on the three GMX tokens. This contract mainly has the following functionalities:

- Set the treasury and contributor manager addresses. These two can then set the treasury and contributor addresses.
- Set a treasury fee percent.
- Distribute fees between the treasury and contributors.

4.5 | PirexFeesContributors.sol

PirexFeesContributors.sol contains the logic for distributing fees between contributors. This contract mainly has the following functionalities:

- Set and update contributors addresses.
- Distribute ETH and ERC20 tokens.

5 | Findings and Improvement Suggestions

Severity	Total	Acknowledged	Resolved
High	1	1	1
Medium	1	1	1
Low	8	8	6
Informational	7	7	5

5.1 | High

5.1.1 | Issues with the distribution of ETH

Severity	High
Source	src/PirexFeesContributors.sol#L210;
Commit	eb297c5;
Status	Resolved in commit b1550bf;

- **Description**
  According to the current design, the `distributeETH()` function allows the distribution of ETH between contributors. However, there's no check that guarantees that the transfers succeed:

```
 // Emit the ETHDistributionFailure event so that it can be handled
if (!sent)
emit ETHDistributionFailure(contributors[i], feePercents[I]);
```

  This means that the contract could keep more ETH when transactions fail. Additionally, the contract implementation considers sending the remaining ETH to the last address:

```
// If this is the *last* contributor set the transfer amount to the balance
// to account for Solidity decimal truncation
uint256 amount = isLast
    ? address(this).balance
    : totalETH.mulDivDown(feePercents[i], FEE_PERCENT_DENOMINATOR);
```

  This leads to the last address receiving more when transactions fail.

- **Exploit Scenario**
  - Suppose there are 9 contributors added to this contract;
  - Contributor number 1 calls the `distributeETH()` function;
  - The transfer fails for contributor 2;
  - Contributor 9 will receive the fees that were meant for contributor 2.

- **Recommendations**
  In case of a transfer failure, send the respective funds to a multi-sig wallet controlled by the team.

- **Results**
  Resolved in commit b1550bf.

  The code was modified in order to consider the transfer failures. Now, the respective funds are sent to a multi-sig wallet controlled by the Redacted Cartel team. The correct distribution will be handled afterward from that wallet to the respective team addresses.

5.2 | Medium

5.2.1 | Migration does not transfer the WETH rewards to the new PirexGMX contract

Severity	**Medium**
Source	src/PirexGmx.sol#L904;
Commit	3aadd22;
Status	Resolved in commit a6d9c75;

- **Description**
  The migration of the `PirexGmx` contract doesn't consider transferring the WETH tokens to the new contract. This causes WETH rewards to remain in the old contract as long as users don't claim them.

  For `PirexGmx`, the WETH rewards are stored in the contract itself and the other rewards are staked into the GMX protocol. When issuing migration, all the staked rewards will be transferred to the new contract by the GMX protocol, but the WETH rewards will remain in the old contract since they are not to be transferred to the new migrated contract.

  Those WETH rewards will remain in the original contract, not allowing the protocol to keep the right track of all the rewards.

- **Exploit Scenario**
  - The owner calls the `initiateMigration()` function to start the migration to a new contract.
  - The owner calls the `completeMigration()` function to finish the migration.
  - All the GMX rewards have been transferred to the new contract but the WETH rewards will remain in the old contract.

- **Recommendations**
  Add functionality to transfer the WETH rewards to the migrated contract.

- **Results**
  Resolved in commit a6d9c75.

  The `migrateReward()` function was added to consider the migration of rewards.

5.3 | Low

5.3.1 | Inaccurate comment

Severity	**Low**
Source	src/vaults/AutoPxGlp.sol#L83;
Commit	a6d9c75;
Status	Resolved in commit a75304e;

- **Description**
  The comment in the line specified of the `AutoPxGlp` states that the approval is for the Uniswap V3 router when actually the platform refers to the `PirexGmx` contract. This can be misleading for users and other developers.

```
// Approve the Uniswap V3 router to manage our base reward (inbound swap token)
gmxBaseReward.safeApprove(address(_platform), type(uint256).max);
```

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Correct the comment.

- **Results**
  Resolved in commit a75304e.

  The comment was changed to:

```
//Approve the main Pirex contract to manage our base reward (inbound swap token)
```

5.3.2 | Unused variables

Severity	**Low**
Source	src/vaults/AutoPxGlp.sol#L20;
Commit	a6d9c75;
Status	Resolved in commit e9fb8ab;

- **Description**
  The constant variable `EXPANDED_DECIMALS` is defined but not used. It's good practice to only keep the necessary code.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Remove unused variables or use them when necessary.
- **Results**
  Resolved in commit e9fb8ab.

  The unused variable was removed.

5.3.3 | `PirexFees` contract address cannot be changed

Severity	**Low**
Source	src/PirexGmx.sol;
Commit	3aadd22;
Status	Resolved in commit ec1db53;

- **Description**
  The `PirexGmx` contract max approves the `PirexFees` contract to spend its WETH, GMX, and GLP. However, the `PirexFees` contract address cannot be updated once the `PirexGmx` contract is deployed. The only way to update it is to redeploy the `PirexGmx` contract with the correct `PirexFees` contract address.

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Add a function to update the address of the `PirexFees` contract within the `PirexGmx` contract.

- **Results**
  Resolved in commit ec1db53.

  Contract logic has been updated to enable updating `pirexFees`.

5.3.4 | Magic Number 75

Severity	**Low**
Source	src/PirexFees.sol#L20, L77;
Commit	3aadd22;
Status	Resolved in commit ec1db53;

- **Description**
  The magic number 75 is directly used to perform a check on the maximum treasury percent. This is not recommended since it is not clear where the number is coming from, instead, it should be named as a variable with a descriptive name.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Replace the magic number 75 by a constant variable with an intuitive name.
- **Results**
  Resolved in commit ec1db53.

  Relevant names were updated to clearly communicate the purpose of the logic.

5.3.5 | Producer update logic

Severity	Low
Source	src/PirexRewards.sol#L93;
Commit	3aadd22;
Status	Acknowledged;

- **Description**

  The producer helps handle the rewards, and the address of this important contract can be modified by the `setProducer()` function directly. The problem with this is that doing so can interrupt internal accounting.

  The only two times when the producer should be updated directly are when:

  the reward states are cleared.

  the `PirexGmx` is migrated to a new address.

  For the above situation/scenarios, there should be an extra check before being able to update the address of this contract.

- **Exploit Scenario**

  N/A.

- **Recommendations**

  Add a check before being able to call the `setProducer()` function. Similar to how flywheel does to update flywheelRewards in https://github.com/fei-protocol/flywheel-v2/blob/dbe3cb8/src/FlywheelCore.sol.

- **Results**

  Acknowledged.

  Response from Redacted team:
  "We believe that the method being permissioned (i.e. only callable by the Pirex multisig) is adequate for ensuring that the producer will not be erroneously set (at minimum, the mistake can be rectified using the same amount of effort, i.e. multisig calls setProducer again with a valid producer). The main reason being that it is impractical to perform certain important checks on chain for validating that the new producer cooperates with the PirexRewards's reward state - e.g. ensuring that the new producer's esGMX balance (see additional thoughts below) is wholly transferred and adequately backs the pxGMX rewards that will be issued against it."

5.3.6 | Unnecessary max approval

Severity	**Low**
Source	src/PirexGmx.sol#L181, 183, 184;
Commit	3aadd22;
Status	Resolved in commit 5b37ad2;

- **Description**
  The `constructor()` of the `PirexGmx` contract max approves the `pirexFees` contract to spend its WETH, pxGmx, and pxGlp tokens. However, the max approval is unnecessary as there is no function that transfers these tokens from `PirexGmx` contract to the `PirexFees` contract.

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Remove the approve lines in the `constructor()` of the `PirexGmx` contract.

- **Results**
  Resolved in commit 5b37ad2.

  Unnecessary `safeApprove()` calls were removed.

5.3.7 | Should revert with zero amount

Severity	**Low**
Source	src/PirexGmx.sol#L778;
Commit	3aadd22;
Status	Resolved in commit 5e327e7;

- **Description**
  Function claimUserReward() should revert with a custom error when the amount is zero. This will allow users to understand why the method is not working as intended:

```
function claimUserReward(
    address token,
    uint256 amount,
    address receiver
) external onlyPirexRewards {
    if (token == address(0)) revert ZeroAddress();
    if (amount == 0) return;
    if (receiver == address(0)) revert ZeroAddress();
```

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Make the claimUserReward() function revert with the respective custom error when the amount is 0:

```
function claimUserReward(
    address token,
    uint256 amount,
    address receiver
) external onlyPirexRewards {
    if (token == address(0)) revert ZeroAddress();
    if (amount == 0) revert ZeroAmount();
    if (receiver == address(0)) revert ZeroAddress();
```

- **Results**
  Resolved in commit 5e327e7.

  The return statement was replaced by reverting with the `ZeroAmount()` custom error.

5.3.8 | Base class should be marked abstract

Severity	**Low**
Source	src/PxGmxReward.sol#L11;
Commit	3aadd22;
Status	Acknowledged;

- **Description**
  The contract `PxGmxReward` is the base class of the actual deployed contract `AutoPxGlp`. In this case, the contract `PxGmxReward` should be abstract to prevent accidental deployment.

  ```
  contract AutoPxGlp is PirexERC4626, PxGmxReward {}
  contract PxGmxReward is Owned {}
  ```

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Mark `PxGmxReward` contract as abstract.

- **Results**
  Acknowledged.

  The contract was not marked as an abstract contract.

5.4 | Informational

5.4.1 | Unused custom errors

Severity	**Informational**

Source	src/AutoPxGlp.sol#L11;
	src/AutoPxGmx.sol#L49;

Commit	3aadd22;
Status	Acknowledged;

- **Description**
  Custom errors `ZeroAmount()` in `AutoPxGlp` and `AlreadySet()` in `AutoPxGmx` are defined but not used in the respective contracts. It's good practice to only keep necessary code.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Remove unused custom errors or use them when required.
- **Results**
  Acknowledged.

  The `ZeroAmount()` custom error was implemented in the respective contract, but the `AlreadySet()` custom error remained unused.

5.4.2 | Variable can be constant

Severity	**Informational**
Source	src/PirexFeesContributors.sol#L15;
Commit	eb297c5;
Status	Resolved in commit b1550bf;

- **Description**
  The `FEE_PERCENT_DENOMINATOR` variable is defined as immutable. However, there's no need to define it this way since it's not set in the `constructor()`:

```
uint256 public immutable FEE_PERCENT_DENOMINATOR = 250_000;
```

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Define the variable as constant to save gas.

```
uint256 public constant FEE_PERCENT_DENOMINATOR = 250_000;
```

- **Results**
  Resolved in commit b1550bf.

  The variable is now defined as constant.

5.4.3 | Unnecesary check on `setContract()` function

Severity	Informational
Source	src/PirexGmx.sol#L319;
Commit	3aadd22;
Status	Resolved in commit 855d013;

- **Description**

The Pirex dev team followed the following code format when performing the conditional assignment of a list of variables:

```
function setParameters(Enumerator f, type targe)
    external
    onlyOwner
{
    // step 1: condition check
    if (target == address(0)) revert ZeroAddress();

    // step 2: emit event
    emit setParameters(f, targe);

    // step 3: assign the target
    // assume Enumerator has 3 targets A, B, C
    if (f == Enumerator.TargetA) {
        targetA = target;
        return;
    }

    if (f == Enumerator.TargetB // position 0) {
        targetB = target;
        return;
    }

    targetC = target;
}
```

in `pirexFees.sol` function `setFeeRecipient()`:

```
function setFeeRecipient(FeeRecipient f, address recipient)
    external
    onlyOwner
{
    if (recipient == address(0)) revert ZeroAddress();

    emit SetFeeRecipient(f, recipient);

    if (f == FeeRecipient.Treasury) {
        treasury = recipient;
        return;
    }

    contributors = recipient;
}
```

Thus in `pirexGmx.sol` function `setContract()`, the last conditional check is unnecessary according to the above example:

```
function setContract(Contracts c, address contractAddress)
    external
```

```
        onlyOwner
    {
        if (contractAddress == address(0)) revert ZeroAddress();

        emit SetContract(c, contractAddress);

        if (c == Contracts.RewardRouterV2) {
            gmxRewardRouterV2 = IRewardRouterV2(contractAddress);
            return;
        }

        .......

        if (c == Contracts.GlpManager) {
            glpManager = contractAddress;
            return;
        }

        // above if section can be optimized to a single line of
        glpManager = contractAddress;
    }
```

Following the above code examples, `pirexGmx.sol` function `setContract()` can be simplified to avoid the last if statement.

- **Exploit Scenario**
  N/A.
- **Recommendations**
  Remove the last if statement and only consider the assignment in the line specified:

```
    if (c == Contracts.GlpManager) {
        glpManager = contractAddress;
        return;
    }

    // above if section can be optimized to a single line of
    glpManager = contractAddress;
```

- **Results**
  Resolved in commit 855d013.

  The conditional was removed and the related contract address for the last enum type was set.

5.4.4 | Use free struct

Severity	Informational
Source	src/Common.sol;
Commit	eb297c5;
Status	Resolved in commit dde2df7;

- **Description**
  Struct can be free in the Solidity file. No need to declare it in the library to have the struct shared by multiple contracts.

  The Common.sol file is a library containing two structs used for handling rewards. There's no need to declare the struct in the library, as declaring free structs (structs outside of contract) is able to have struct shared by multiple contracts already.

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Change the code in the file Common.sol to the following:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.17;

struct GlobalState {
    uint32 lastUpdate;
    uint224 lastSupply;
    uint256 rewards;
}

struct UserState {
    uint32 lastUpdate;
    uint224 lastBalance;
    uint256 rewards;
}
```

- **Results**
  Resolved in commit dde2df7.

  The structs in `Common.sol` were declared directly without being wrapped in a Library.

5.4.5 | Naming issues (non-descriptive names)

Severity	Informational
Source	src/PirexGmx.sol#L333, L377, L570, L640, L667;
Commit	3aadd22;
Status	Resolved in commit de9309d;

- **Description**
  The variables in the following lines referred to as assets are actually representing amounts. Using the word assets might confuse other developers because normally that word represents an array of ERC20. For this reason, we recommend using the word amount instead.

```
/**
@notice Deposit GMX for pxGMX
@param  assets    uint256  GMX amount  <== plz change
@param  receiver  address  pxGMX receiver
@return           address  GMX deposited
@return           uint256  pxGMX minted for the receiver
@return           uint256  pxGMX distributed as fees
*/
function depositGmx(uint256 assets, address receiver){}
function depositFsGlp(uint256 assets, address receiver){}
function _redeemPxGlp(address token,
  uint256 assets,
  uint256 minOut,
  address receiver
){}
function redeemPxGlpETH(
  uint256 assets,
  uint256 minOut,
  address receiver
){}
function redeemPxGlp(
  address token,
  uint256 assets,
  uint256 minOut,
  address receiver
){}
```

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Consider changing the word asset to amount in the functions specified.

- **Results**
  Resolved in commit de9309d.

  The variable naming for assets was swapped out to the amount in deposits and redemption methods in `PirexGmx`.

5.4.6 | Missing variables and/or indexed variables in events)

Severity	**Informational**
Source	src/PirexGmx.sol#L91, L92, L136, L137, L138, L139;
Commit	3aadd22;
Status	Acknowledged;

- **Description**

  Events are vital aids in monitoring contracts and detecting suspicious behavior. Adding the related variables associated with the events becomes critical when searching for specific information. Additionally, adding the indexed keyword allows developers/users to speed up the search and filter for logs.

  Update events do not include old values and operator addresses (index).

  ```
  // PirexGmx.sol
  event SetFee(Fees indexed f, uint256 fee);
  event SetContract(Contracts indexed c, address contractAddress);
  event InitiateMigration(address newContract);
  event CompleteMigration(address oldContract);
  event SetDelegationSpace(string delegationSpace, bool shouldClear);
  event SetVoteDelegate(address voteDelegate);
  ```

- **Exploit Scenario**

  N/A.

- **Recommendations**

  Add the missing variables and ìndexed keyword to the respective events.

- **Results**

  Acknowledged.

  Response from the Redacted team: "The 6 events referenced in the report (as of the current state) are non-frequent events with no requirement for specific filtering, and thus deemed as having sufficient parameters declared without indexing feature."

5.4.7 | Inconsistent comments

Severity	Informational
Source	src/PirexGmx.sol#L329, L373;
Commit	3aadd22;
Status	Resolved in commit 6d8e9b0;

- **Description**
  There's an inconsistency in the comments of the `depositGMX()` and `depositFsGlp()` functions. Both functions return three uint256, so the NatSpec comment associated with the GMX/fsGLP deposited should be uint256 instead of address:

```
/**
    @notice Deposit GMX for pxGMX
    @param  assets    uint256  GMX amount
    @param  receiver  address  pxGMX receiver
    @return           address  GMX deposited
    @return           uint256  pxGMX minted for the receiver
    @return           uint256  pxGMX distributed as fees
*/
function depositGmx(uint256 assets, address receiver)
    external
    whenNotPaused
    nonReentrant
    returns (
        uint256,
        uint256,
        uint256
    )

/**
    @notice Deposit fsGLP for pxGLP
    @param  assets    uint256  fsGLP amount
    @param  receiver  address  pxGLP receiver
    @return           address  fsGLP deposited
    @return           uint256  pxGLP minted for the receiver
    @return           uint256  pxGLP distributed as fees
*/
function depositFsGlp(uint256 assets, address receiver)
    external
    whenNotPaused
    nonReentrant
    returns (
        uint256,
        uint256,
        uint256
    )
```

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Change inconsistent comments

- **Results**
  Resolved in commit 6d8e9b0.

  The specified inconsistent comments in `PirexGmx.sol` were updated.

6 | Use Case Scenarios

Pirex-GMX is a platform that enables users to access liquid pxGMX and pxGLP, auto-compounding, and the ability to tokenize future vote events. This integration with the decentralized exchange GMX allows users to trade the two main tokens they receive for providing liquidity and interacting with the protocol (GMX and GLP) for the equivalent liquid assets provided by Pirex. If users decide to go with the "Standard Mode," they choose to manually claim and compound rewards. On the other hand, if they opt for the "Easy Mode," all the rewards will be continuously auto-compounded.

7 | Access Control Analysis

The protocol workflow consists of depositing one of the two main tokens obtained in the GMX platform (GMX and GLP) to get liquid assets provided by Pirex (pxGMX and pxGLP). The deployer of the contracts that handle this functionality is a multi-sig wallet controlled by the Pirex team, its main access control and other privileged roles for each of the contracts are explained below:

7.1 | PirexGmx.sol

- Set the fee amount for the different types of transactions.
- Set the addresses for the contracts that the protocol will be interacting with.
- Handle delegation logic.
- Migrate contract in case of emergency.

7.2 | PirexRewards.sol

- Set addresses of producer tokens (GMX and GLP).
- Add/Remove reward tokens.
- Finalize the retirement of tokens.
- Add/Remove privileged reward recipients.

7.3 | PirexFees.sol

- Set the treasury and contributors managers.

8 | Notes and Additional Information

8.1 | ERC4626 Share Price Inflation

Source: src/vaults/PirexERC4626.sol;
Commit: a6d9c75;

Empty ERC4626 vaults can be easily manipulated to inflate the price of a share and cause depositors to lose their deposits due to rounding issues.
For more details, check https://github.com/transmissions11/solmate/issues/178.
For this reason, it's recommended to consider a minimal amount of share tokens to be minted for the first minter and send a part of the initial mints as a permanent reserve so that the price per share can be more resistant to manipulation.

8.2 | Issue With The compound() Function

Source: src/vaults/AutoPxGmx.sol#L240;
Commit: a6d9c75;

Anyone can call the `compound()` function to swap the gmxBaseReward token to the GMX token. Additionally, the `amountOutMinimum` and the `sqrtPriceLimitX96` are set by the users themselves.
The caller might perform a sandwich attack and cause the protocol to suffer from unexpected price slippage. The current `compound()` function provides the caller with some incentives to mitigate the risk.

9 | Appendix

9.1 | Appendix I: Severity Categories

Severity	Description
High	Issues that are highly exploitable security vulnerabilities. It may cause direct loss of funds / permanent freezing of funds. All high severity issues should be resolved.
Medium	Issues that are only exploitable under some conditions or with some privileged access to the system. Users' yields/rewards/information is at risk. All medium severity issues should be resolved unless there is a clear reason not to.
Low	Issues that are low risk. Not fixing those issues will not result in the failure of the system. A fix on low severity issues is recommended but subject to the clients' decisions.
Information	Issues that pose no risk to the system and are related to the security best practices. Not fixing those issues will not result in the failure of the system. A fix on informational issues or adoption of those security best practices-related suggestions is recommended but subject to clients' decision.

9.2 | Appendix II: Status Categories

Severity	Description
Unresolved	The issue is not acknowledged and not resolved.
Partially Resolved	The issue has been partially resolved
Acknowledged	The Finding / Suggestion is acknowledged but not fixed / not implemented.
Resolved	The issue has been sufficiently resolved

10 | Disclaimer