Introducing Code4rena Profiles: a solo auditor's highlight reel. Learn more →

Redacted Cartel contest Findings & Analysis Report

2023-01-27

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Redacted Cartel smart contract system written in Solidity. The audit contest took place between November 21—November 28 2022.

Wardens

106 Wardens contributed reports to the Redacted Cartel contest:

1. 0x52
2. 0xAgro
3. 0xLad
4. 0xNazgul
5. 0xPanda
6. 0xSmartContract
7. 0xbepresent
8. 0xfuje
9. 8olidity
10. Awesome
11. B2
12. BnkeOx0
13. Deivitto
14. Diana
15. Englave
16. Funen
17. HE1M
18. Jeiwan
19. JohnSmith
20. Josiah
21. KingNFT

22. Koolex
23. Lambda
24. PaludoX0
25. R2
26. Rahoz
27. RaymondFam
28. ReyAdmirado
29. Rolezn
30. Ruhum
31. Sathish9098
32. Schlagatron
33. Secureverse (imkapadia, Nsecv, and leosathya)
34. Tomio
35. Waze
36. __141345__
37. adriro
38. ajtra
39. aphak5010
40. bin2chen
41. brgltd
42. btk
43. c3phas
44. carrotsmuggler
45. cccz
46. ch0bu
47. chaduke
48. codeislight

49. codexploder
50. cryptoDave
51. cryptonue
52. cryptostellar5
53. csanuragjain
54. danyams
55. datapunk
56. delfin454000
57. deliriusz
58. dharma09
59. eierina
60. erictee
61. fatherOfBlocks
62. gogo
63. gz627
64. gzeon
65. halden
66. hansfriese
67. hihen
68. hl_
69. imare
70. immeas
71. jadezti
72. joestakey
73. karanctf
74. keccak123
75. koxuan

76. kyteg
77. ladboy233
78. martin
79. nameruse
80. oyc_109
81. pashov
82. pavankv
83. peanuts
84. pedr02b2
85. pedroais
86. perseverancesuccess
87. poirots (DavideSilva, resende, naps62, and eighty)
88. rbserver
89. rotcivegaf
90. rvierdiiev
91. sakshamguruji
92. saneryee
93. seyni
94. shark
95. simon135
96. subtle77
97. unforgiven
98. wagmi
99. xiaoming90
100. yixxas
101. yongskiws

This contest was judged by Picodes.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 18 unique vulnerabilities. Of these vulnerabilities, 6 received a risk rating in the category of HIGH severity and 12 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 60 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 33 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Redacted Cartel contest repository, and is composed of 13 smart contracts written in the Solidity programming language and includes 1,981 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (6)

[H-01] The `redeem` related functions are likely to be blocked

Submitted by KingNFT, *also found by* xiaoming90, ladboy233, 0x52, rvierdiiev, *and* HE1M

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexGmx.sol#L615

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexGmx.sol#L685

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexGmx.sol#L712

Impact

The following `redeem` related functions are likely to be blocked, users will not be able to retrieve their funds.

```
    function _redeemPxGlp(
        address token,
        uint256 amount,
        uint256 minOut,
        address receiver
    );
    function redeemPxGlpETH(
        uint256 amount,
```

```
        uint256 minOut,
        address receiver
    );
    function redeemPxGlp(
        address token,
        uint256 amount,
        uint256 minOut,
        address receiver
    );
```

Proof of Concept

The `GlpManager` contract of GMX has a `cooldownDuration` limit on redeem/unstake (`\_removeLiquidity()`). While there is at least one deposit/stake (`\_addLiquidity()`) operation in the past `cooldownDuration` time, redemption would fail. Obviously this limitation is user-based, and `PirexGmx` contract is one such user.

https://github.com/gmx-io/gmx-contracts/blob/c3618b0d6fc1b88819393dc7e6c785e32e78c72b/contracts/core/GlpManager.sol#L234

```
    Current setting of `cooldownDuration` is 15 minutes, the
```

https://arbiscan.io/address/0x321f653eed006ad1c29d174e17d96351bde22649#readContract

Due to the above limit, there are 3 risks that can block redemption for Pirex users.

1. The normal case

Let's say there is 10% GMX users will use Pirex to manage their GLP.

By checking recent history of GMX router contract, we can find the average stake interval is smaller than 1 minute

https://arbiscan.io/address/0xa906f338cb21815cbc4bc87ace9e68c87ef8d8f1

Let's take

```
averageStakeIntervalOfGMX = 30 seconds
```

So if Pirex has 10% of GMX users, then

```
averageStakeIntervalOfPirex = 30 ÷ 10% = 300 seconds
```

The probability of successfully redeeming is a typical Poisson distribution: https://en.wikipedia.org/wiki/Poisson_distribution.

With

```
λ = cooldownDuration ÷ averageStakeIntervalOfPirex = 15 *
k = 0
```

So we get

```
P ≈ 1 ÷ (2.718 * 2.718 * 2.718) ≈ 0.05
```

Conclusion

```
If Pirex has 10 % of GMX users, then the redemption will
```

A full list of % of GMX users versus failure probability of redemption

```
1% : 26%
```

```
5% : 78%
10% : 95%
20% : 99.75%
30% : 99.98%
```

2. The attack case

If an attacker, such as bad competitors of similar projects, try to exploit this vulnerability.

Let's estimate the cost for attack.

An attacker can deposit a very small GLP, such as 1 wei, so we can ignore the GLP cost and only focus on GAS cost.

By checking the explorer history https://arbiscan.io We are safe to assume the cost for calling

`depositGlpETH()` or `depositGlp` is

```
txCost = 0.1 USD
```

To block redemption, attacker has to execute a deposit call every 15 minutes, so

```
dailyCost = 24 * (60 / 15) * 0.1 = 9.6 USD
yearCost = 365 * 9.6 = 3504 USD
```

Conclusion

```
If an attacker wants to block Pirex users funds, his year
```

3. GMX adjusts protocol parameters

If GMX increases `cooldownDuration` to 2 days, it will obviously cause redemption not working.

Tools Used

VS Code

Recommended Mitigation Steps

Reserve some time range for redemption only. e.g. 1 of every 7 days.

kphed (Redacted Cartel) confirmed

[H-02] Users Receive Less Rewards Due To Miscalculations

Submitted by xiaoming90*, also found by* __141345__

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexRewards.sol#L305

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexRewards.sol#L281

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexRewards.sol#L373

Background

The amount of rewards accrued by global and user states is computed by the following steps:

1. Calculate seconds elapsed since the last update (`block.timestamp - lastUpdate`)
2. Calculate the new rewards by multiplying seconds elapsed by the last supply (`(block.timestamp - lastUpdate) * lastSupply`)
3. Append the new rewards to the existing rewards (`rewards = rewards + (block.timestamp - lastUpdate) * lastSupply`)

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexRewards.sol#L305

```
/**
    @notice Update global accrual state
    @param  globalState    GlobalState  Global state of t
    @param  producerToken  ERC20        Producer token cc
*/
function _globalAccrue(GlobalState storage globalState, F
        internal
{
    uint256 totalSupply = producerToken.totalSupply();
    uint256 lastUpdate = globalState.lastUpdate;
    uint256 lastSupply = globalState.lastSupply;

    // Calculate rewards, the product of seconds elapsed
    // Only calculate and update states when needed
    if (block.timestamp != lastUpdate || totalSupply != l
        uint256 rewards = globalState.rewards +
            (block.timestamp - lastUpdate) *
            lastSupply;

            globalState.lastUpdate = block.timestamp.safe
            globalState.lastSupply = totalSupply.safeCast
            globalState.rewards = rewards;
        ..SNIP..
}
```

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/

PirexRewards.sol#L281

```
/**
    @notice Update user rewards accrual state
    @param  producerToken  ERC20    Rewards-producing tok
    @param  user           address  User address
*/
function userAccrue(ERC20 producerToken, address user) pu
    if (address(producerToken) == address(0)) revert Zerc
    if (user == address(0)) revert ZeroAddress();

    UserState storage u = producerTokens[producerToken].u
    uint256 balance = producerToken.balanceOf(user);

    // Calculate the amount of rewards accrued by the use
    uint256 rewards = u.rewards +
    u.lastBalance *
    (block.timestamp - u.lastUpdate);

    u.lastUpdate = block.timestamp.safeCastTo32();
    u.lastBalance = balance.safeCastTo224();
    u.rewards = rewards;
    ..SNIP..
}
```

When a user claims the rewards, the number of reward tokens the user is entitled to is equal to the `rewardState` scaled by the ratio of the `userRewards` to the `globalRewards`. Refer to Line 403 below.

The `rewardState` represents the total number of a specific ERC20 reward token (e.g. WETH or esGMX) held by a producer (e.g. pxGMX or pxGPL).

The `rewardState` of each reward token (e.g. WETH or esGMX) will increase whenever the rewards are harvested by the producer (e.g. `PirexRewards.harvest` is called). On the other hand, the `rewardState` will decrease if the users claim the rewards.

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexRewards.sol#L373

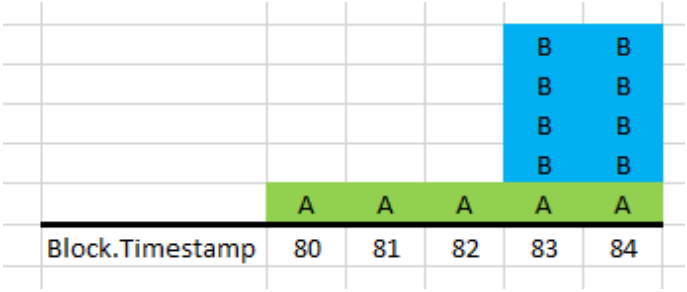
```
File: PirexRewards.sol
373:     function claim(ERC20 producerToken, address user
..SNIP..
395:             // Transfer the proportionate reward tok
396:             for (uint256 i; i < rLen; ++i) {
397:                 ERC20 rewardToken = rewardTokens[i];
398:                 address rewardRecipient = p.rewardRe
399:                 address recipient = rewardRecipient
400:                     ? rewardRecipient
401:                     : user;
402:                 uint256 rewardState = p.rewardStates
403:                 uint256 amount = (rewardState * user
..SNIP..
417:     }
```

How reward tokens are distributed

The Multiplier Point (MP) effect will be ignored for simplicity. Assume that the emission rate is constant throughout the entire period (from T80 to T84) and the emission rate is 1 esGMX per 1 GMX staked per second.

The graph below represents the amount of GMX tokens Alice and Bob staked for each second during the period.

A = Alice and B = Bob; each block represents 1 GMX token staked.



Based on the above graph:

- Alice staked 1 GMX token from T80 to T84. Alice will earn five (5) esGMX tokens at the end of T84.
- Bob staked 4 GMX tokens from T83 to T84. Bob will earn eight (8) esGMX tokens at the end of T84.
- A total of 13 esGMX will be harvested by `PirexRewards` contract at the end of T84

The existing reward distribution design in the `PirexRewards` contract will work perfectly if the emission rate is constant, similar to the example above.

In this case, the state variable will be as follows at the end of T84, assuming both the global and all user states have been updated and rewards have been harvested.

- rewardState = 13 esGMX tokens (5 + 8)
- globalRewards = 13
- Accrued `userRewards` of Alice = 5
- Accrued `userRewards` of Bob = 8

When Alice calls the `PirexRewards.claim` function to claim her rewards at the end of T84, she will get back five (5) esGMX tokens, which is correct.

```
(rewardState * userRewards) / globalRewards
(13 * 5) / 13 = 5
```

Proof of Concept

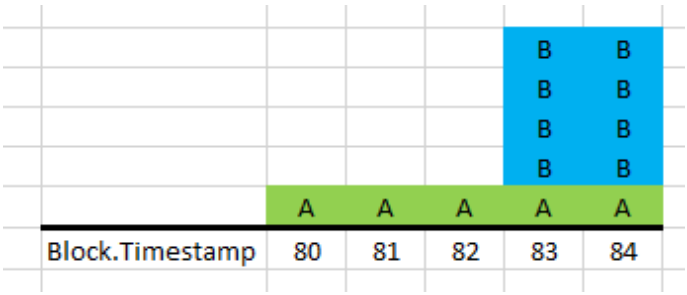
However, the fact is that the emission rate of reward tokens (e.g. esGMX or WETH) is not constant. Instead, the emission rate is dynamic and depends on various factors, such as the following:

- The number of rewards tokens allocated by GMX governance for each month. Refer to https://gov.gmx.io/t/esgmx-emissions/272. In some months, the number of esGMX emissions will be higher.

- The number of GMX/GLP tokens staked by the community. The more tokens being staked by the community users, the more diluted the rewards will be.

The graph below represents the amount of GMX tokens Alice and Bob staked for each second during the period.

A = Alice and B = Bob; each block represents 1 GMX token staked.



The Multiplier Point (MP) effect will be ignored for simplicity. Assume that the emission rate is as follows:

- From T80 to 82: 2 esGMX per 1 GMX staked per second (Higher emission rate)
- From T83 to 84: 1 esGMX per 1 GMX staked per second (Lower emission rate)

By manually computing the amount of esGMX reward tokens that Alice is entitled to at the end of T84:

```
[1 staked GMX * (T82 - T80) * 2esGMX/sec] + [1 staked GMX
[1 staked GMX * 3 secs * 2esGMX/sec] + [1 staked GMX * 2s
6 + 2 = 8
```

Alice will be entitled to 8 esGMX reward tokens at the end of T84.

By manually computing the amount of esGMX reward tokens that Bob is entitled to at the end of T84:

```
[4 staked GMX * 2secs * 1esGMX/sec] = 8
```

Bob will be entitled to 8 esGMX reward tokens at the end of T84.

However, the existing reward distribution design in the `PirexRewards` contract will cause Alice to get fewer reward tokens than she is entitled to and cause Bob to get more rewards than he is entitled to.

The state variable will be as follows at the end of T84, assuming both the global and all user states have been updated and rewards have been harvested.

- rewardState = 16 esGMX tokens (8 + 8)
- globalRewards = 13
- Accrued `userRewards` of Alice = 5
- Accrued `userRewards` of Bob = 8

When Alice calls the `PirexRewards.claim` function to claim her rewards at the end of T84, she will only get back six (6) esGMX tokens, which is less than eight (8) esGMX tokens she is entitled to or earned.

```
(rewardState * userRewards) / globalRewards
(16 * 5) / 13 = 6.15 = 6
```

When Bob calls the `PirexRewards.claim` function to claim his rewards at the end of T84, he will get back nine (9) esGMX tokens, which is more than eight (8) esGMX tokens he is entitled to or earned.

```
(rewardState * userRewards) / globalRewards
(16 * 8) / 13 = 9.85 = 9
```

Impact

As shown in the PoC, some users will lose their reward tokens due to the miscalculation within the existing reward distribution design.

Recommended Mitigation Steps

Update the existing reward distribution design to handle the dynamic emission rate. Implement the RewardPerToken for users and global, as seen in many of the well-established reward contracts below, which are not vulnerable to this issue:

- https://github.com/fei-protocol/flywheel-v2/blob/dbe3cb81a3dc2e46536bb8af9c2bdc585f63425e/src/FlywheelCore.sol#L226
- https://github.com/Synthetixio/synthetix/blob/2cb4b23fe409af526de67dfbb84aae84b2b13747/contracts/StakingRewards.sol#L61

kphed (Redacted Cartel) confirmed

[H-03] Malicious Users Can Drain The Assets Of Auto Compound Vault

Submitted by xiaoming90, *also found by* pashov, adriro, poirots, unforgiven, bin2chen, PaludoX0, 0xSmartContract, ladboy233, Ruhum, cccz, koxuan, 8olidity, *and* rvierdiiev

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/PirexERC4626.sol#L156

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L199

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L315

Proof of Concept

> Note: This issue affects both the AutoPxGmx and AutoPxGlp vaults. Since the root cause is the same, the PoC of AutoPxGlp vault is omitted for brevity.

The `PirexERC4626.convertToShares` function relies on the `mulDivDown` function in Line 164 when calculating the number of shares needed in exchange for a certain number of assets. Note that the computation is rounded down, therefore, if the result is less than 1 (e.g. 0.9), Solidity will round them down to zero. Thus, it is possible that this function will return zero.

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/PirexERC4626.sol#L156

```
File: PirexERC4626.sol
156:     function convertToShares(uint256 assets)
157:         public
158:         view
159:         virtual
160:         returns (uint256)
161:     {
162:         uint256 supply = totalSupply; // Saves an ex
163:
164:         return supply == 0 ? assets : assets.mulDivD
165:     }
```

The `AutoPxGmx.previewWithdraw` function relies on the `PirexERC4626.convertToShares` function in Line 206. Thus, this function will also “round down”.

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L199

```
File: AutoPxGmx.sol
199:     function previewWithdraw(uint256 assets)
200:         public
201:         view
202:         override
203:         returns (uint256)
204:     {
205:         // Calculate shares based on the specified a
206:         uint256 shares = convertToShares(assets);
207:
208:         // Save 1 SLOAD
209:         uint256 _totalSupply = totalSupply;
210:
211:         // Factor in additional shares to fulfill wi
212:         return
213:             (_totalSupply == 0 || _totalSupply - sha
214:                 ? shares
215:                 : (shares * FEE_DENOMINATOR) /
216:                     (FEE_DENOMINATOR - withdrawalPer
217:     }
```

The `AutoPxGmx.withdraw` function relies on the `AutoPxGmx.previewWithdraw` function. In certain conditions, the `AutoPxGmx.previewWithdraw` function in Line 323 will return zero if the withdrawal amount causes the division within the `PirexERC4626.convertToShares` function to round down to zero (usually due to a small amount of withdrawal amount).

If the `AutoPxGmx.previewWithdraw` function in Line 323 returns zero, no shares will be burned at Line 332. Subsequently, in Line 336, the contract will transfer the assets to the users. As a result, the users receive the assets without burning any of their shares, effectively allowing them to receive assets for free.

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L315

```
File: AutoPxGmx.sol
315:     function withdraw(
316:         uint256 assets,
317:         address receiver,
318:         address owner
319:     ) public override returns (uint256 shares) {
320:         // Compound rewards and ensure they are prop
321:         compound(poolFee, 1, 0, true);
322:
323:         shares = previewWithdraw(assets); // No need
324:
325:         if (msg.sender != owner) {
326:             uint256 allowed = allowance[owner][msg.s
327:
328:             if (allowed != type(uint256).max)
329:                 allowance[owner][msg.sender] = allow
330:         }
331:
332:         _burn(owner, shares);
333:
334:         emit Withdraw(msg.sender, receiver, owner, a
335:
336:         asset.safeTransfer(receiver, assets);
337:     }
```

Assume that the vault with the following state:

- Total Asset = 1000 WETH
- Total Supply = 10 shares

Assume that Alice wants to withdraw 99 WETH from the vault. Thus, she calls the `AutoPxGmx.withdraw(99 WETH)` function.

The `PirexERC4626.convertToShares` function will compute the number of shares that Alice needs to burn in exchange for 99 WETH.

```
assets.mulDivDown(supply, totalAssets())
99WETH.mulDivDown(10 shares, 1000WETH)
```

```
(99 * 10) / 1000
990 / 1000 = 0.99 = 0
```

However, since Solidity rounds `0.99` down to `0`, Alice does not need to burn a single share. She will receive 99 WETH for free.

Impact

Malicious users can withdraw the assets from the vault for free, effectively allowing them to drain the assets of the vault.

Recommended Mitigation Steps

Ensure that at least 1 share is burned when the users withdraw their assets.

This can be mitigated by updating the `previewWithdraw` function to round up instead of round down when computing the number of shares to be burned.

```
function previewWithdraw(uint256 assets)
        public
        view
        override
        returns (uint256)
{
        // Calculate shares based on the specified assets
-       uint256 shares = convertToShares(assets);
+       uint256 shares = supply == 0 ? assets : assets.mu

        // Save 1 SLOAD
        uint256 _totalSupply = totalSupply;

        // Factor in additional shares to fulfill withdra
        return
                (_totalSupply == 0 || _totalSupply - shar
                        ? shares
                        : (shares * FEE_DENOMINATOR) /
                                (FEE_DENOMINATOR - withdr
```

```
    }
```

kphed (Redacted Cartel) confirmed

[H-04] User's Accrued Rewards Will Be Lost

Submitted by xiaoming90

If the user deposits too little GMX compared to other users (or total supply of pxGMX), the user will not be able to receive rewards after calling the `PirexRewards.claim` function. Subsequently, their accrued rewards will be cleared out (set to zero), and they will lose their rewards.

The amount of reward tokens that are claimable by a user is computed in Line 403 of the `PirexRewards.claim` function.

If the balance of pxGMX of a user is too small compared to other users (or total supply of pxGMX), the code below will always return zero due to rounding issues within solidity.

```
uint256 amount = (rewardState * userRewards) / globalRewa
```

Since the user's accrued rewards is cleared at Line 391 within the `PirexRewards.claim` function (`p.userStates[user].rewards = 0;`), the user's accrued rewards will be lost.

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexRewards.sol#L373

```
File: PirexRewards.sol
368:     /**
369:         @notice Claim rewards
```

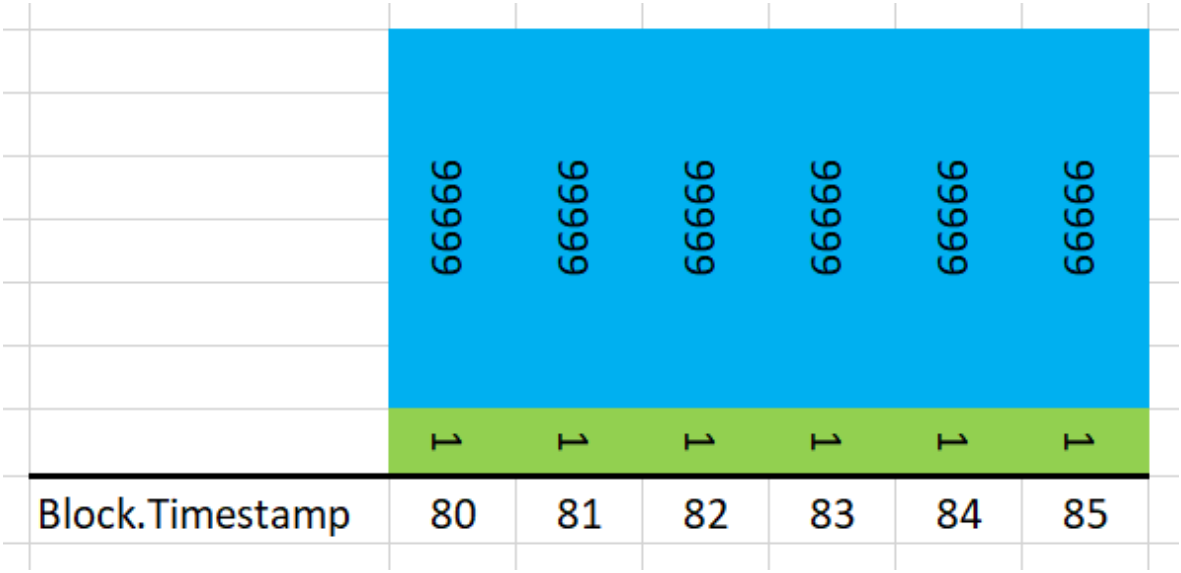
```
370:          @param  producerToken  ERC20    Producer tok
371:          @param  user           address  User
372:      */
373:      function claim(ERC20 producerToken, address user
374:          if (address(producerToken) == address(0)) re
375:          if (user == address(0)) revert ZeroAddress()
376:
377:          harvest();
378:          userAccrue(producerToken, user);
379:
380:          ProducerToken storage p = producerTokens[prc
381:          uint256 globalRewards = p.globalState.rewarc
382:          uint256 userRewards = p.userStates[user].rew
383:
384:          // Claim should be skipped and not reverted
385:          if (globalRewards != 0 && userRewards != 0)
386:              ERC20[] memory rewardTokens = p.rewardTc
387:              uint256 rLen = rewardTokens.length;
388:
389:              // Update global and user reward states
390:              p.globalState.rewards = globalRewards -
391:              p.userStates[user].rewards = 0;
392:
393:              emit Claim(producerToken, user);
394:
395:              // Transfer the proportionate reward tok
396:              for (uint256 i; i < rLen; ++i) {
397:                  ERC20 rewardToken = rewardTokens[i];
398:                  address rewardRecipient = p.rewardRe
399:                  address recipient = rewardRecipient
400:                      ? rewardRecipient
401:                      : user;
402:                  uint256 rewardState = p.rewardStates
403:                  uint256 amount = (rewardState * user
404:
405:                  if (amount != 0) {
406:                      // Update reward state (i.e. amc
407:                      p.rewardStates[rewardToken] = re
408:
409:                      producer.claimUserReward(
410:                          address(rewardToken),
411:                          amount,
412:                          recipient
```

```
413:                         );
414:                     }
415:                 }
416:             }
417:         }
```

The graph below represents the amount of GMX tokens Alice and Bob staked in `PirexGmx` for each second during the period. Note that the graph is not drawn proportionally.

Green = Number of GMX tokens staked by Alice

Blue = Number of GMX tokens staked by Bob



Based on the above graph:

- Alice staked 1 GMX token for 4 seconds (From T80 to T85)
- Bob staked 99999 GMX tokens for 4 seconds (From T80 to T85)

Assume that the emission rate is 0.1 esGMX per 1 GMX staked per second.

In this case, the state variable will be as follows at the end of T83, assuming both the global and all user states have been updated and rewards have been harvested.

- rewardState = 60,000 esGMX tokens (600,000 * 0.1)
- globalRewards = 600,000 (100,000 * 6)
- Accrued `userRewards` of Alice = 6
- Accrued `userRewards` of Bob = 599,994 (99,999 * 6)

Following is the description of `rewardState` for reference:

> The `rewardState` represents the total number of a specific ERC20 reward token (e.g. WETH or esGMX) held by a producer (e.g. pxGMX or pxGPL).

> The `rewardState` of each reward token (e.g. WETH or esGMX) will increase whenever the rewards are harvested by the producer (e.g. `PirexRewards.harvest` is called). On the other hand, the `rewardState` will decrease if the users claim the rewards.

At the end of T85, Alice should be entitled to 1.2 esGMX tokens (0.2/sec * 6).

Following is the formula used in the `PirexRewards` contract to compute the number of reward tokens a user is entitled to.

```
amount = (rewardState * userRewards) / globalRewards;
```

If Alice claims the rewards at the end of T85, she will get zero esGMX tokens instead of 1.2 esGMX tokens.

```
amount = (rewardState * userRewards) / globalRewards;
60,000 * 6 / 600,000
360,000/600,000 = 0.6 = 0
```

Since Alice's accrued rewards are cleared at Line 391 within the `PirexRewards.claim` function (`p.userStates[user].rewards = 0;`),

Alice's accrued rewards will be lost. Alice will have to start accruing the rewards from zero after calling the `PirexRewards.claim` function.

Another side effect is that since the 1.2 esGMX tokens that belong to Alice are still in the contract, they will be claimed by other users.

Impact

Users who deposit too little GMX compared to other users (or total supply of pxGMX), the user will not be able to receive rewards after calling the `PirexRewards.claim` function. Also, their accrued rewards will be cleared out (set to zero). Loss of reward tokens for the users.

Additionally, the `PirexRewards.claim` function is permissionless, and anyone can trigger the claim on behalf of any user. A malicious user could call the `PirexRewards.claim` function on behalf of a victim at the right time when the victim's accrued reward is small enough to cause a rounding error or precision loss, thus causing the victim accrued reward to be cleared out (set to zero).

Recommended Mitigation Steps

Following are some of the possible remediation actions:

1. Use `RewardPerToken` approach

Avoid calculating the rewards that the users are entitled based on the ratio of `userRewards` and `globalRewards`.

Instead, consider implementing the RewardPerToken for users and global, as seen in many of the well-established reward contracts below, which are not vulnerable to this issue:

- https://github.com/fei-protocol/flywheel-v2/blob/dbe3cb81a3dc2e46536bb8af9c2bdc585f63425e/src/FlywheelCore.sol#L226

- https://github.com/Synthetixio/synthetix/blob/2cb4b23fe409af526de67dfbb84aae84b2b13747/contracts/StakingRewards.sol#L61

2. Fallback logic if `amount == 0`

If the `amount` is zero, revert the transaction. Alternatively, if the `amount` is zero, do not clear out the user's accrued reward state variable since the user did not receive anything yet.

```
function claim(ERC20 producerToken, address user) externa
..SNIP..
                        uint256 amount = (rewardState * u

                        if (amount != 0) {
                                // Update reward state (i
                                p.rewardStates[rewardToke

                                producer.claimUserReward(
                                        address(rewardTok
                                        amount,
                                        recipient
                                );
-                       }
+                       } else {
+                               revert ZeroRewardTokens()
+                       }
..SNIP..
}
```

kphed (Redacted Cartel) confirmed

[H-05] Underlying assets stealing in `AutoPxGmx` and `AutoPxGlp` via share price manipulation

Submitted by Jeiwan, *also found by* seyni, gogo, pashov, hl_, rbserver, peanuts, __141345__, unforgiven, Lambda, joestakey, JohnSmith, R2, Koolex, xiaoming90, yongskiws, carrotsmuggler, ladboy233,

0xSmartContract, KingNFT, cccz, HE1M, rvierdiiev, koxuan, 8olidity, *and* 0xLad

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/PirexERC4626.sol#L156-L165

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/PirexERC4626.sol#L167-L176

Impact

pxGMX and pxGLP tokens can be stolen from depositors in `AutoPxGmx` and `AutoPxGlp` vaults by manipulating the price of a share.

Proof of Concept

ERC4626 vaults are subject to a share price manipulation attack that allows an attacker to steal underlying tokens from other depositors (this is a known issue of Solmate's ERC4626 implementation). Consider this scenario (this is applicable to `AutoPxGmx` and `AutoPxGlp` vaults):

1. Alice is the first depositor of the `AutoPxGmx` vault;
2. Alice deposits 1 wei of pxGMX tokens;
3. in the `deposit` function (PirexERC4626.sol#L60), the amount of shares is calculated using the `previewDeposit` function:

```
function previewDeposit(uint256 assets)
    public
    view
    virtual
    returns (uint256)
{
    return convertToShares(assets);
}

function convertToShares(uint256 assets)
```

```
        public
        view
        virtual
        returns (uint256)
    {
        uint256 supply = totalSupply; // Saves an extra SLOAD if to

        return supply == 0 ? assets : assets.mulDivDown(supply, tot
    }
```

4. Since Alice is the first depositor (totalSupply is 0), she gets 1 share (1 wei);
5. Alice then *sends* 9999999999999999 pxGMX tokens (10e18 - 1) to the vault;
6. The price of 1 share is 10 pxGMX tokens now: Alice is the only depositor in the vault, she's holding 1 wei of shares, and the balance of the pool is 10 pxGMX tokens;
7. Bob deposits 19 pxGMX tokens and gets only 1 share due to the rounding in the `convertToShares` function: `19e18 * 1 / 10e18 == 1`;
8. Alice redeems her share and gets a half of the deposited assets, 14.5 pxGMX tokens (less the withdrawal fee);
9. Bob redeems his share and gets only 14.5 pxGMX (less the withdrawal fee), instead of the 19 pxGMX he deposited.

```
// test/AutoPxGmx.t.sol
function testSharePriceManipulation_AUDIT() external {
    address alice = address(0x31337);
    address bob = address(0x12345);
    vm.label(alice, "Alice");
    vm.label(bob, "Bob");

    // Resetting the withdrawal fee for cleaner amounts.
    autoPxGmx.setWithdrawalPenalty(0);

    vm.startPrank(address(pirexGmx));
    pxGmx.mint(alice, 10e18);
    pxGmx.mint(bob, 19e18);
```

```
        vm.stopPrank();

        vm.startPrank(alice);
        pxGmx.approve(address(autoPxGmx), 1);
        // Alice deposits 1 wei of pxGMX and gets 1 wei of sh
        autoPxGmx.deposit(1, alice);
        // Alice sends 10e18-1 of pxGMX and sets the price of
        pxGmx.transfer(address(autoPxGmx), 10e18-1);
        vm.stopPrank();

        vm.startPrank(bob);
        pxGmx.approve(address(autoPxGmx), 19e18);
        // Bob deposits 19e18 of pxGMX and gets 1 wei of shar
        autoPxGmx.deposit(19e18, bob);
        vm.stopPrank();

        // Alice and Bob redeem their shares.
        vm.prank(alice);
        autoPxGmx.redeem(1, alice, alice);
        vm.prank(bob);
        autoPxGmx.redeem(1, bob, bob);

        // Alice and Bob both got 14.5 pxGMX.
        // But Alice deposited 10 pxGMX and Bob deposited 19
        // With withdrawal fees enabled, Alice would've been
        // (14.065 pxGMX vs 14.935 pxGMX tokens withdrawn, re
        // but Alice would've still gotten more pxGMX that sh
        assertEq(pxGmx.balanceOf(alice), 14.5e18);
        assertEq(pxGmx.balanceOf(bob), 14.5e18);
    }
```

Recommended Mitigation Steps

Consider either of these options:

1. In the `deposit` function of `PirexERC4626`, consider requiring a reasonably high minimal amount of assets during first deposit. The amount needs to be high enough to mint many shares to reduce the rounding error and low enough to be affordable to users.

2. On the first deposit, consider minting a fixed and high amount of shares, irrespective of the deposited amount.
3. Consider seeding the pools during deployment. This needs to be done in the deployment transactions to avoid front-running attacks. The amount needs to be high enough to reduce the rounding error.
4. Consider sending first 1000 wei of shares to the zero address. This will significantly increase the cost of the attack by forcing an attacker to pay 1000 times of the share price they want to set. For a well-intended user, 1000 wei of shares is a negligible amount that won't diminish their share significantly.

Picodes (judge) increased severity to High

kphed (Redacted Cartel) confirmed

[H-06] fee loss in AutoPxGmx and AutoPxGlp and reward loss in AutoPxGlp by calling `PirexRewards.claim(pxGmx/pxGpl, AutoPx*)` directly which transfers rewards to AutoPx* pool without compound logic get executed and fee calculation logic and pxGmx wouldn't be executed for those rewards

Submitted by unforgiven*, also found by* bin2chen *and* 0x52

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGlp.sol#L197-L296

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L230-L313

Impact

Function `compound()` in `AutoPxGmx` and `AutoPxGlp` contracts is for compounding `pxGLP` (and additionally `pxGMX`) rewards. it works by calling `PirexGmx.claim(px*, this)` to collect the rewards of the vault and then swap the received amount (to calculate the reward, contract save the balance of a contract in that reward token before and after the call to the `claim()` and by subtracting them finds the received reward amount) and deposit them in `PirexGmx` again for compounding and in doing so it calculates fee based on what it received and in `AutoPxGlp` case it calculates `pxGMX` rewards too based on the extra amount contract receives during the execution of `claim()`. But attacker can call `PirexGmx.claim(px*, PirexGlp)` directly and make `PirexGmx` contract to transfer (`gmxBaseReward` and `pxGmx`) rewards to `AutoPxGlp` and in this case the logics of fee calculation and reward calculation in `compound()` function won't get executed and contract won't get it's fee from rewards and users won't get their `pxGmx` reward. So this bug would cause fee loss in `AutoPxGmx` and `AutoPxGlp` for contract and `pxGmx`'s reward loss for users in `AutoPxGlp`.

Proof of Concept

The bug in `AutoPxGmx` is similar to `AutoPxGlp`, so we only give Proof of Concept for `AutoPxGlp`.

This is `compound()` function code in `AutoPxGlp` contract:

```
    function compound(
        uint256 minUsdg,
        uint256 minGlp,
        bool optOutIncentive
    )
        public
        returns (
            uint256 gmxBaseRewardAmountIn,
            uint256 pxGmxAmountOut,
            uint256 pxGlpAmountOut,
            uint256 totalPxGlpFee,
            uint256 totalPxGmxFee,
```

```
            uint256 pxGlpIncentive,
            uint256 pxGmxIncentive
        )
    {
        if (minUsdg == 0) revert InvalidParam();
        if (minGlp == 0) revert InvalidParam();

        uint256 preClaimTotalAssets = asset.balanceOf(adc
        uint256 preClaimPxGmxAmount = pxGmx.balanceOf(adc

        PirexRewards(rewardsModule).claim(asset, address(
        PirexRewards(rewardsModule).claim(pxGmx, address(

        // Track the amount of rewards received
        gmxBaseRewardAmountIn = gmxBaseReward.balanceOf(a

        if (gmxBaseRewardAmountIn != 0) {
            // Deposit received rewards for pxGLP
            (, pxGlpAmountOut, ) = PirexGmx(platform).dep
                address(gmxBaseReward),
                gmxBaseRewardAmountIn,
                minUsdg,
                minGlp,
                address(this)
            );
        }

        // Distribute fees if the amount of vault assets
        uint256 newAssets = totalAssets() - preClaimTotal
        if (newAssets != 0) {
            totalPxGlpFee = (newAssets * platformFee) / F
            pxGlpIncentive = optOutIncentive
                ? 0
                : (totalPxGlpFee * compoundIncentive) / F

            if (pxGlpIncentive != 0)
                asset.safeTransfer(msg.sender, pxGlpIncer

            asset.safeTransfer(owner, totalPxGlpFee - pxG
        }

        // Track the amount of pxGMX received
        pxGmxAmountOut = pxGmx.balanceOf(address(this)) -
```

```
        if (pxGmxAmountOut != 0) {
            // Calculate and distribute pxGMX fees if the
            totalPxGmxFee = (pxGmxAmountOut * platformFee
            pxGmxIncentive = optOutIncentive
                ? 0
                : (totalPxGmxFee * compoundIncentive) / F

            if (pxGmxIncentive != 0)
                pxGmx.safeTransfer(msg.sender, pxGmxIncer

            pxGmx.safeTransfer(owner, totalPxGmxFee - pxG

            // Update the pxGmx reward accrual
            _harvest(pxGmxAmountOut - totalPxGmxFee);
        } else {
            // Required to keep the globalState up-to-dat
            _globalAccrue();
        }

        emit Compounded(
            msg.sender,
            minGlp,
            gmxBaseRewardAmountIn,
            pxGmxAmountOut,
            pxGlpAmountOut,
            totalPxGlpFee,
            totalPxGmxFee,
            pxGlpIncentive,
            pxGmxIncentive
        );
    }
```

As you can see contract collects rewards by calling `PirexRewards.claim()` and in the line `uint256 newAssets = totalAssets() - preClaimTotalAssets;` contract calculates the received amount of rewards (by subtracting the balance after and before reward claim) and then calculates fee based on this amount `totalPxGlpFee = (newAssets * platformFee) / FEE_DENOMINATOR;` and then sends the fee in the line

`asset.safeTransfer(owner, totalPxGlpFee - pxGlpIncentive)` for `owner`.

The logic for `pxGmx` rewards are the same. As you can see the calculation of the fee is based on the rewards received, and there is no other logic in the contract to calculate and transfer the fee of protocol. So if `AutoPxGpl` receives rewards without `compound()` getting called then for those rewards fee won't be calculated and transferred and protocol would lose it's fee.

In the line `_harvest(pxGmxAmountOut - totalPxGmxFee)` contract calls `_harvest()` function to update the `pxGmx` reward accrual and there is no call to `_harvest()` in any other place and this is the only place where `pxGmx` reward accrual gets updated. The contract uses `pxGmxAmountOut` which is the amount of `gmx` contract received during the call (code calculates it by subtracting the balance after and before reward claim: `pxGmxAmountOut = pxGmx.balanceOf(address(this)) - preClaimPxGmxAmount;`) so contract only handles accrual rewards in this function call and if some `pxGmx` rewards claimed for contract without `compund()` logic execution then those rewards won't be used in `_harvest()` and `_globalAccrue()` calculation and users won't receive those rewards.

As mentioned attacker can call `PirexRewards.claim(pxGmx, AutoPxGpl)` directly and make `PirexRewads` contract to transfer `AutoPxGpl` rewards. This is `claim()` code in `PirexRewards` :

```
function claim(ERC20 producerToken, address user) ext
    if (address(producerToken) == address(0)) revert
    if (user == address(0)) revert ZeroAddress();

    harvest();
    userAccrue(producerToken, user);

    ProducerToken storage p = producerTokens[producer
    uint256 globalRewards = p.globalState.rewards;
    uint256 userRewards = p.userStates[user].rewards;
```

```
        // Claim should be skipped and not reverted on ze
        if (globalRewards != 0 && userRewards != 0) {
            ERC20[] memory rewardTokens = p.rewardTokens;
            uint256 rLen = rewardTokens.length;

            // Update global and user reward states to re
            p.globalState.rewards = globalRewards - userR
            p.userStates[user].rewards = 0;

            emit Claim(producerToken, user);

            // Transfer the proportionate reward token am
            for (uint256 i; i < rLen; ++i) {
                ERC20 rewardToken = rewardTokens[i];
                address rewardRecipient = p.rewardRecipie
                address recipient = rewardRecipient != ad
                    ? rewardRecipient
                    : user;
                uint256 rewardState = p.rewardStates[rewa
                uint256 amount = (rewardState * userRewar

                if (amount != 0) {
                    // Update reward state (i.e. amount)
                    p.rewardStates[rewardToken] = rewards

                    producer.claimUserReward(
                        address(rewardToken),
                        amount,
                        recipient
                    );
                }
            }
        }
    }
```

As you can see it can be called by anyone for any user. So to perform this attack, attacker would perform these steps:

1. Suppose `AutoPxGpl` has pending rewards, for example 100 `pxGmx` and 100 `weth`.

2. Attacker would call `PirexRewards.claim(pxGmx, AutoPxGpl)` and `PirexRewards.claim(pxGpl, AutoPxGpl)` and `PirexRewards` contract would calculate and claim and transfer `pxGmx` rewards and `weth` rewards of `AutoPxGpl` address.
3. Then `AutoPxGpl` has no pending rewards but the balance of `pxGmx` and `weth` of contract has been increased.
4. If anyone calls `AutoPxGpl.compound()` because there is no pending rewards contract would receive no rewards and because contract only calculates fee and rewards based on received rewards during the call to `compound()` so contract wouldn't calculate any fee or reward accrual for those 1000 `pxGmx` and `weth` rewards.
5. `owner` of `AutoPxGpl` would lose his fee for those rewards and users of `AutoPxGpl` would lose their claims for those 1000 `pxGmx` rewards (because the calculation for them didn't happen).

This bug is because of the fact that the only logic handling rewards is in `compound()` function which is only handling receiving rewards by calling `claim()` during the call to `compound()` but it's possible to call `claim()` directly (`PirexRewards` contract allows this) and `AutoPxGpl` won't get notified about this new rewards and the related logics won't get executed.

Tools Used

VIM

Recommended Mitigation Steps

Contract should keep track of it's previous balance when `compound()` get executed and update this balance in deposits and withdraws and claims so it can detect rewards that directly transferred to contract without call to `compound()`.

kphed (Redacted Cartel) confirmed

Medium Risk Findings (12)

[M-01] `PirexGmx.initiateMigration` can be blocked

Submitted by rvierdiiev*, also found by* imare *and* 0x52

`PirexGmx.initiateMigration` can be blocked so contract will not be able to migrate his funds to another contract using gmx.

Proof of Concept

PirexGmx was designed with the thought that the current contract can be changed with another during migration.

`PirexGmx.initiateMigration` is the first point in this long process.

https://github.com/code-423n4/2022-11-redactedcartel/blob/main/src/PirexGmx.sol#L921-L935

```
    function initiateMigration(address newContract)
        external
        whenPaused
        onlyOwner
    {
        if (newContract == address(0)) revert ZeroAddress

        // Notify the reward router that the current/old
        // full account transfer to the specified new cor
        gmxRewardRouterV2.signalTransfer(newContract);

        migratedTo = newContract;

        emit InitiateMigration(newContract);
    }
```

As you can see `gmxRewardRouterV2.signalTransfer(newContract);` is called to start migration.

This is the code of signalTransfer function
https://arbiscan.io/address/0xA906F338CB21815cBc4Bc87ace9e68c87eF8d8F1#code#F1#L282

```
function signalTransfer(address _receiver) external r
    require(IERC20(gmxVester).balanceOf(msg.sender) =
    require(IERC20(glpVester).balanceOf(msg.sender) =

    _validateReceiver(_receiver);
    pendingReceivers[msg.sender] = _receiver;
}
```

As you can see the main condition to start migration is that PirexGmx doesn't control any gmxVester and glpVester tokens.

So attacker can deposit and receive such tokens and then just transfer tokens directly to PirexGmx.

As a result migration will never be possible as there is no possibility for PirexGmx to burn those gmxVester and glpVester tokens.

Also in the same way, the migration receiver can also be blocked.

Tools Used

VS Code

Recommended Mitigation Steps

Think about how to make contract ensure that he doesn't control any gmxVester and glpVester tokens before migration.

Picodes (judge) decreased severity to Medium

Please note: the following comment occurred after judging and awarding were finalized.

kphed (Redacted Cartel) commented:

> This issue is invalid and not possible to carry out as a non-GMX insider (if the GMX team and multisig were malicious, there would be many other ways in which they can steal value, so this specific vector would be the least of our concerns) for the following reasons:
>
> 1. The vester token transfer methods are overridden which removes the possibility of an attacker acquiring vGMX or vGLP and transferring it to the PirexGmx contract via those methods.
>
> Vester.sol | Lines 246-263
>
> - vGMX
> - vGLP
>
> ```
> // empty implementation, tokens are non-transferrable
> function transfer(address /* recipient */, uint256 /* amount */
>  revert("Vester: non-transferrable");
> }
> ```
>
> ...
>
> // empty implementation, tokens are non-transferrable function transferFrom(address /* sender */, address /* recipient */, uint256 /* amount */) public virtual override returns (bool) { revert("Vester: non-transferrable"); }
>
> ```
> 2. The `depositForAccount` method can only be called by a
> ```

[M-02] Preventing any user from calling the functions `withdraw`, `redeem`, or `depositGmx` in contract `AutoPxGmx`

Submitted by HE1M

It is possible that an attacker can prevent any user from calling the functions `withdraw`, `redeem`, or `depositGmx` in contract `AutoPxGmx` by just manipulating the balance of token `gmxBaseReward`, so that during the function `compound` the swap will be reverted.

Proof of Concept

Whenever a user calls the functions `withdraw`, `redeem`, or `depositGmx` in contract `AutoPxGmx`, the function `compound` is called:

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L321

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L345

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L379

The function `compound` claims token `gmxBaseReward` from `rewardModule`: https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L262

Then, if the balance of the token `gmxBaseReward` in custodian of the contract `AutoPxGmx` is not zero, the token `gmxBaseReward` will be swapped to token `GMX` thrrough uniswap V3 by calling the function

`exactInputSingle`. Then the total amount of token `GMX` in custodian of the contract `AutoPxGmx` will be deposited in the contract `PirexGmx` to receive token `pxGMX`:

```
if (gmxBaseRewardAmountIn != 0) {
            gmxAmountOut = SWAP_ROUTER.exactInputSingle(
                IV3SwapRouter.ExactInputSingleParams({
                    tokenIn: address(gmxBaseReward),
                    tokenOut: address(gmx),
                    fee: fee,
                    recipient: address(this),
                    amountIn: gmxBaseRewardAmountIn,
                    amountOutMinimum: amountOutMinimum,
                    sqrtPriceLimitX96: sqrtPriceLimitX96
                })
            );

            // Deposit entire GMX balance for pxGMX, incr
            (, pxGmxMintAmount, ) = PirexGmx(platform).de
                gmx.balanceOf(address(this)),
                address(this)
            );
        }
```

Whenever the function `compound` is called inside the mentioned functions, the parameters are: `compound(poolFee, 1, 0, true);`

- `fee = poolFee`
- `amountOutMinimum = 1`
- `sqrtPriceLimitX96 = 0`
- `optOutIncentive = true`

The vulnerability is the parameter `amountOutMinimum` which is equal to 1. This provides an attack surface so that if the balance of token `gmxBaseReward` in `AutoPxGmx` is nonzero and small enough that does not worth 1 token `GMX`, the swap procedure will be reverted.

For example, if the balance of `gmxBaseReward` is equal to 1, then since the value of `gmxBaseReward` is lower than token `GMX`, the output amount of `GMX` after swapping `gmxBaseReward` will be zero, and as parameter `amountOutMinimum` is equal to 1, the swap will be reverted, and as a result, the compound function will be reverted.

Attack Scenario:

Suppose, recently the function compound was called, so the balance of token `gmxBaseReward` in contract `AutoPxGmx` is equal to zero. Later, Alice (honest user) would like to withdraw. So, she calls the function `withdraw(...)`.

```
    function withdraw(
            uint256 assets,
            address receiver,
            address owner
        ) public override returns (uint256 shares) {
            // Compound rewards and ensure they are properly
            compound(poolFee, 1, 0, true);

            shares = previewWithdraw(assets); // No need to c

            if (msg.sender != owner) {
                uint256 allowed = allowance[owner][msg.sender

                if (allowed != type(uint256).max)
                    allowance[owner][msg.sender] = allowed -
            }

            _burn(owner, shares);

            emit Withdraw(msg.sender, receiver, owner, assets

            asset.safeTransfer(receiver, assets);
        }
```

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L315

In a normal situation, the function `compound` will be called, and since the balance of `gmxBaseReward` is zero, no swap will be executed in uniswap V3, and the rest of the code logic will be executed.

But in this scenario, before Alice's transaction, Bob transfers 1 token `gmxBaseReward` directly to contract `AutoPxGmx` . So, when Alice's transaction is going to be executed, inside the function `compound` the swap function will be called (because the balance `gmxBaseReward` is equal to 1 now). In the function `exactInputSingle` in uniswap V3, there is a check: `require(amountOut >= params.amountOutMinimum, 'Too little received');`
https://etherscan.io/address/0xe592427a0aece92de3edee1f18e0157c05861564#code#F1#L128

This check will be reverted, because 1 token of `gmxBaseReward` is worth less than 1 token of `GMX` , so the amount out will be zero which is smaller than `amountOutMinimum` .

In summary, an attacker before user's deposit, checks the balance of token `gmxBaseReward` in `AutoPxGmx` . If this balance is equal to zero, the attacker transfers 1 token `gmxBaseReward` to contract `AutoPxGmx` , so the user's transaction will be reverted, and user should wait until this balance reaches to the amount that worth more than or equal to 1 token `GMX` .

Recommended Mitigation Steps

The parameter `amountOutMinimum` should be equal to zero when the function `compound` is called. `compound(poolFee, 0, 0, true);`

Picodes (judge) decreased severity to Medium and commented:

> The attack is unlikely: there is no real reason to do it as the attacker would have to pay gas, you need conditions on the price, and conditions on the reward amounts, as if rewards are accruing you won't be able to do it, so downgrading to Medium severity.

> However it's true that it may be good to set a minimum amount for the swaps that depends on the `amountIn`, or remove entirely the `minAmountOut` requirement.

kphed (Redacted Cartel) commented:

> Hi @Picodes, this is going to be resolved as a side effect of issue **#321** being fixed (i.e. `compound` will be updated to consider token balances w/o dependence on external factors to prevent DoS'ing of users). Wanted to flag it for your attention but ultimately up to you re: awarding. Thanks for your help ser.

[M-03] Anyone can call AutoPxGmx.compound and perform sandwich attacks with control parameters

Submitted by **cccz**, *also found by* **Englave**, **immeas**, **hansfriese**, **rbserver**, **Jeiwan**, **xiaoming90**, *and* **aphak5010**

AutoPxGmx.compound allows anyone to call to compound the reward and get the incentive.

However, AutoPxGmx.compound calls `SWAP_ROUTER`.exactInputSingle with some of the parameters provided by the caller, which allows the user to perform a sandwich attack for profit.

For example, a malicious user could provide the fee parameter to make the token swap occur in a small liquid pool, and could make the amountOutMinimum parameter 1 to make the token swap accept a large slippage, thus making it easier to perform a sandwich attack.

Proof of Concept

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L242-L278

Recommended Mitigation Steps

Consider using poolFee as the fee and using an onchain price oracle to calculate the amountOutMinimum.

Picodes (judge) commented:

> Flagging as best as the warden identifies that the main risk is not the possibility to increase fees but the fact that some of the pools will be highly illiquid.

drahrealm (Redacted Cartel) disagreed with severity and commented:

> Please refer to: https://github.com/code-423n4/2022-11-redactedcartel-findings/issues/185#issuecomment-1341252133

Picodes (judge) commented:

> It's very likely that this attack is profitable as most of the time only 1 or 2 pools have decent liquidity, so Medium severity is appropriate.

[M-04] AutoPxGmx.maxWithdraw and AutoPxGlp.maxWithdraw functions calculate asset amount that is too big and cannot be withdrawn

Submitted by aphak5010

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/PirexERC4626.sol#L225

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L14

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGlp.sol#L14

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L315

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L199

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L215-L216

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L332

Impact

The ERC-4626 Tokenized Vault Standard requires the `maxWithdraw` function to be implemented (https://ethereum.org/en/developers/docs/standards/tokens/erc-4626/#maxwithdraw).

This function is supposed to return “the maximum amount of underlying assets that can be withdrawn from the owner balance with a single withdraw call”.

The `PirexERC4626` contract implements this function (https://github.com/code-423n4/2022-11-

redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/PirexERC4626.sol#L225).
It is implemented correctly when `PirexERC4626` is used on its own.

However in this project, the `PirexERC4626` contract is not used on its own but inherited by `AutoPxGmx` (https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L14) and `AutoPxGlp` (https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGlp.sol#L14).

`AutoPxGmx` and `AutoPxGlp` implement a `withdrawalPenalty` i.e. a fee that is paid when a user wants to withdraw assets from the vault.

`AutoPxGmx` and `AutoPxGlp` do not override the `maxWithdraw` function.

This causes the `maxWithdraw` function to return an amount of assets that is too big to be withdrawn.

So when `maxWithdraw` is called and with the returned amount `withdraw` is called, the call to `withdraw` will revert.

This can cause issues in any upstream components that rely on `AutoPxGmx` and `AutoPxGlp` to correctly implement the ERC4626 standard.

For example an upstream wrapper might only allow withdrawals with the maximum amount and determine this maximum amount by calling the `maxWithdraw` function. As this function returns a value that is too big, no withdrawals will be possible.

Proof of Concept

1. The `maxWithdraw` function in a `AutoPxGmx` contract is called

2. Now the `withdraw` function is called with the value that was returned by the `maxWithdraw` function (https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L315)
3. The `withdraw` function in turn calls the `previewWithdraw` function (https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L199)
4. The `previewWithdraw` function will increase the amount of shares to include the `withdrawalPenalty` (https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L215-L216) which causes the amount of shares to burn to be too large and the call to `burn` will revert (https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L332)

Tools Used

VS Code

Recommended Mitigation Steps

In the `AutoPxGmx` and `AutoPxGlp` function, implement the `maxWithdraw` function that overrides the function in `PirexERC4626` and takes into account the `withdrawalPenalty`.

Potential fix:

```
function maxWithdraw(address owner) public view override
    uint256 shares = balanceOf(owner);

    // Calculate assets based on a user's % ownership of
    uint256 assets = convertToAssets(shares);
```

```
        uint256 _totalSupply = totalSupply;

        // Calculate a penalty - zero if user is the last to
        uint256 penalty = (_totalSupply == 0 || _totalSupply
            ? 0
            : assets.mulDivDown(withdrawalPenalty, FEE_DENOMI

        // Redeemable amount is the post-penalty amount
        return assets - penalty;
    }
```

drahrealm (Redacted Cartel) confirmed

[M-05] `SWAP_ROUTER` in `AutoPxGmx.sol` is hardcoded and not compatible on Avalanche

Submitted by ladboy233, *also found by* gzeon

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L18

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L96

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L268

Impact

I want to quote from the doc:

```
    - Does it use a side-chain? Yes
```

```
- If yes, is the sidechain evm-compatible? Yes, Avalanche
```

This shows that the project is intended to support Avalanche side chain.

`SWAP_ROUTER` in the AutoPxGmx.sol is hardcoded as:

```
IV3SwapRouter public constant SWAP_ROUTER =
        IV3SwapRouter(0x68b3465833fb72A70ecDF485E0e4C7bD8
```

But this address is the Uniswap V3 router address in arbitrium, but it is a EOA address in Avalanche,

https://snowtrace.io/address/0x68b3465833fb72a70ecdf485e0e4c7bd8665fc45

Then the AutoPxGmx.sol is not working in Avalanche.

```
gmxAmountOut = SWAP_ROUTER.exactInputSingle(
        IV3SwapRouter.ExactInputSingleParams({
                tokenIn: address(gmxBaseReward),
                tokenOut: address(gmx),
                fee: fee,
                recipient: address(this),
                amountIn: gmxBaseRewardAmountIn,
                amountOutMinimum: amountOutMinimum,
                sqrtPriceLimitX96: sqrtPriceLimitX96
        })
);
```

Proof of Concept

The code below reverts because the EOA address on Avalanche does not have exactInputSingle method in compound method.

```
gmxAmountOut = SWAP_ROUTER.exactInputSingle(
        IV3SwapRouter.ExactInputSingleParams({
                tokenIn: address(gmxBaseReward),
                tokenOut: address(gmx),
                fee: fee,
                recipient: address(this),
                amountIn: gmxBaseRewardAmountIn,
                amountOutMinimum: amountOutMinimum,
                sqrtPriceLimitX96: sqrtPriceLimitX96
        })
);



/**
        @notice Compound pxGMX rewards
        @param  fee                     uint24   Uniswap p
        @param  amountOutMinimum        uint256  Outbound
        @param  sqrtPriceLimitX96       uint160  Swap pric
        @param  optOutIncentive         bool     Whether t
        @return gmxBaseRewardAmountIn   uint256  GMX base
        @return gmxAmountOut            uint256  GMX outbc
        @return pxGmxMintAmount         uint256  pxGMX mir
        @return totalFee                uint256  Total pla
        @return incentive               uint256  Compound
 */
function compound(
        uint24 fee,
        uint256 amountOutMinimum,
        uint160 sqrtPriceLimitX96,
        bool optOutIncentive
)
```

Recommended Mitigation Steps

We recommend the project not hardcode the `SWAP_ROUTER` in AutoPxGmx.sol, can pass this parameter in the constructor.

kphed (Redacted Cartel) commented:

> The core set of contracts currently functions for both Arbitrum and Avalanche, but the AutoPxGmx contract does not (the auto-compounding contracts are part of the non-core "Easy Mode" offering). We're aware of this and are holding off on completing those changes launching on Avalanche until after our Arbitrum launch goes smoothly. Thank you for participating in our C4 contest!

[M-06] Assets may be lost when calling unprotected `AutoPxGlp::compound` function

Submitted by deliriusz, *also found by* keccak123, wagmi, pashov, 0xbepresent, rbserver, unforgiven, simon135, xiaoming90, gzeon, R2, ladboy233, 0xLad, 0x52, pedroais, Ruhum, cccz, hihen, rvierdiiev, perseverancesuccess, *and* Englave

https://github.com/code-423n4/2022-11-redactedcartel/blob/main/src/vaults/AutoPxGlp.sol#L210

https://github.com/code-423n4/2022-11-redactedcartel/blob/main/src/PirexGmx.sol#L497-L516

Impact

Compounded assets may be lost because `AutoPxGlp::compound` can be called by anyone and minimum amount of Glp and USDG are under caller's control. The only check concerning minValues is that they are not zero (1 will work, however from the perspective of real tokens e.g. 1e6, or 1e18 it's virtually zero). Additionally, internal smart contract functions use it as well with minimal possible value (e.g. `beforeDeposit` function).

Proof of Concept

`compound` function calls PirexGmx::depositGlp, that uses external GMX reward router to mint and stake GLP.

https://snowtrace.io/address/0x82147c5a7e850ea4e28155df107f2590fd4ba327#code

```
141:     function mintAndStakeGlpETH(uint256 _minUsdg, ui
    ...
148: uint256 glpAmount = IGlpManager(glpManager).addLiqui
```

Next `GlpManager::addLiquidityForAccount` is called https://github.com/gmx-io/gmx-contracts/blob/master/contracts/core/GlpManager.sol#L103

```
    function addLiquidityForAccount(address _fundingAccou
        _validateHandler();
        return _addLiquidity(_fundingAccount, _account, _
    }
```

which in turn uses vault to swap token for specific amount of USDG before adding liquidity: https://github.com/gmx-io/gmx-contracts/blob/master/contracts/core/GlpManager.sol#L217

The amount of USGD to mint is calcualted by GMX own price feed: https://github.com/gmx-io/gmx-contracts/blob/master/contracts/core/Vault.sol#L765-L767

In times of market turbulence, or price oracle manipulation, all compound value may be lost

Tools Used

VS Code, arbiscan.io

Recommended Mitigation Steps

Don't depend on user passing minimum amounts of usdg and glp tokens. Use GMX oracle to get current price, and additionally check it against some

other price feed (e.g. ChainLink).

kphed (Redacted Cartel) commented:

> We're using the following combination of mechanics in order to make front-running `compound` calls economically unattractive (or, at the very least, minimally impactful) for would-be attackers:
>
> - Compound incentives
> - Execution as a side effect of vault functions
>
> Both will result in a higher frequency of the vault compounding its rewards and less resources available for potential attackers.

[M-07] Deposit Feature Of The Vault Will Break If Update To A New Platform

Submitted by xiaoming90, *also found by* joestakey, hansfriese, unforgiven, bin2chen, ladboy233, 0x52, aphak5010, hihen, 8olidity, cccz, *and* rvierdiiev

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L73

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L152

Proof of Concept

During initialization, the `AutoPxGMX` vault will grant max allowance to the platform (PirexGMX) to spend its GMX tokens in Line 97 of the constructor method below. This is required because the vault needs to deposit GMX tokens to the platform (PirexGMX) contract. During deposit, the platform (PirexGMX) contract will pull the GMX tokens within the vault and send them

to GMX protocol for staking. Otherwise, the deposit feature within the vault will not work.

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L73

```
File: AutoPxGmx.sol
73:     constructor(
74:         address _gmxBaseReward,
75:         address _gmx,
76:         address _asset,
77:         string memory _name,
78:         string memory _symbol,
79:         address _platform,
80:         address _rewardsModule
81:     ) Owned(msg.sender) PirexERC4626(ERC20(_asset), _
82:         if (_gmxBaseReward == address(0)) revert Zerc
83:         if (_gmx == address(0)) revert ZeroAddress();
84:         if (_asset == address(0)) revert ZeroAddress(
85:         if (bytes(_name).length == 0) revert InvalidA
86:         if (bytes(_symbol).length == 0) revert Invali
87:         if (_platform == address(0)) revert ZeroAddre
88:         if (_rewardsModule == address(0)) revert Zerc
89:
90:         gmxBaseReward = ERC20(_gmxBaseReward);
91:         gmx = ERC20(_gmx);
92:         platform = _platform;
93:         rewardsModule = _rewardsModule;
94:
95:         // Approve the Uniswap V3 router to manage ou
96:         gmxBaseReward.safeApprove(address(SWAP_ROUTEF
97:         gmx.safeApprove(_platform, type(uint256).max)
98:     }
```

However, when the owner calls the `AutoPxGmx.setPlatform` function to update the `platform` to a new address, it does not grant any allowance to the new platform address. As a result, the new platform (PirexGMX) will not

be able to pull the GMX tokens from the vault. Thus, the deposit feature of the vault will break, and no one will be able to deposit.

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGmx.sol#L152

```
File: AutoPxGmx.sol
152:     function setPlatform(address _platform) external
153:         if (_platform == address(0)) revert ZeroAddr
154:
155:         platform = _platform;
156:
157:         emit PlatformUpdated(_platform);
158:     }
```

Impact

The deposit feature of the vault will break, and no one will be able to deposit.

Recommended Mitigation Steps

Ensure that allowance is given to the new platform address so that it can pull the GMX tokens from the vault.

```
function setPlatform(address _platform) external onlyOwne
    if (_platform == address(0)) revert ZeroAddress();
+   if (_platform == platform) revert SamePlatformAddress

+   gmx.safeApprove(platform, 0); // set the old platform
+   gmx.safeApprove(_platform, type(uint256).max); // app

    platform = _platform;

    emit PlatformUpdated(_platform);
}
```

kphed (Redacted Cartel) confirmed

Picodes (judge) decreased severity to Medium and commented:

> Changing to Medium risk as the DOS would just be temporary as the platform could be reset to its previous value, and there is no clear risk of loss of funds.

[M-08] Tokens with fees will break the `depositGlp()` logic

Submitted by R2*, also found by* kyteg

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/vaults/AutoPxGlp.sol#L367

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexGmx.sol#L583

Impact

In `PirexGmx` and `AutoPxGlp` you have function `depositGlp()`, which accepts any ERC20 token from whitelist.

Now there are 9 tokens (see here: https://arbiscan.io/address/0x489ee077994b6658eafa855c308275ead8097c4a#readContract):

`WBTC, WETH, USDC, LINK, UNI, USDT, MIM, FRAX, DAI` And the list may extend

So any user can deposit any of those tokens and receive pxGlp token:

```
function testUSDTDepositGlp() external {
    // 0 USDT TOKENS
    address myAddress = address(this);
    assertEq(usdt.balanceOf(myAddress), 0);

    // The one with many USDT tokens
    vm.prank(0xB6CfcF89a7B22988bfC96632aC2A9D6daB60d6
    uint256 amount = 100000;
    usdt.transfer(myAddress, amount);

    // amount USDT TOKENS
    assertEq(usdt.balanceOf(myAddress), amount);

    // DEPOSIT USDT TOKENS
    usdt.approve(address(pirexGmx), amount);
    pirexGmx.depositGlp(address(usdt), amount, 1, 1,

    // SUCCESSSFULLY DEPOSITED
    assertEq(usdt.balanceOf(address(this)), 0);
    assertEq(pxGlp.balanceOf(address(this)), 11889002
}
```

But if any of these tokens will start charge fee on transfers, the logic will be broken and calls to `depositGlp()` with suck token will fail.

Because here you use the amount of tokens sent from user wallet: https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexGmx.sol#L512

```
        t.safeTransferFrom(msg.sender, address(this),

        // Mint and stake GLP using ERC20 tokens
        deposited = gmxRewardRouterV2.mintAndStakeGlp
            token,
            tokenAmount,
            minUsdg,
            minGlp
```

```
            );
```

And then `gmxRewardRouterV2` tries to transfer tokens to his balance from your balance:

```
    IERC20(_token).safeTransferFrom(_fundingAccount, address(
```

(See https://arbiscan.io/address/0x321f653eed006ad1c29d174e17d96351bde22649#code - GlpManager and https://arbiscan.io/address/0xA906F338CB21815cBc4Bc87ace9e68c87eF8d8F1#code - RewardRouterV2)

But you received less than `tokenAmount` tokens because of fee. The transaction will fail.

Proof of Concept

Let's imagine USDT in arbitrub started to charge fees 1% per transfer.

Alice wants to deposit 100 USDT through `PirexGmx.depositGlp()`

Then you do `t.safeTransferFrom(Alice, address(this), 100);`

You will receive only 99 USDT

But in the next line you will try to send 100 USDT:

```
    deposited = gmxRewardRouterV2.mintAndStakeGlp(
                    token,
                    tokenAmount,
                    minUsdg,
                    minGlp
                );
```

So transaction fails and Alice can't get pxGlp tokens.

Tools Used

VS Code

Recommended Mitigation Steps

USDT already has fees in other blockchains.

Many of these tokens use proxy pattern (and USDT too). It's quite probably that in one day one of the tokens will start charge fees. Or you would like to add one more token to whitelist and the token will be with fees.

Thats why I consider finding as Medium severity.

To avoid problems, use common pattern, when you check your balance before operation and balance after, like that:

```
uint256 balanceBefore = t.balanceOf(address(t
t.safeTransferFrom(msg.sender, address(this),
uint256 balanceAfter = t.balanceOf(address(th

uint256 tokenAmount = balanceAfter - balanceB

// Mint and stake GLP using ERC20 tokens
deposited = gmxRewardRouterV2.mintAndStakeGlp
    token,
    tokenAmount,
    minUsdg,
    minGlp
);
```

[drahrealm (Redacted Cartel) confirmed]

[Picodes (judge) commented]:

> As USDT is already an accepted underlying token, Medium severity is appropriate.

[M-09] broken logic in `configureGmxState()` of PirexGmx contract because it doesn't properly call `safeApprove()` for stakedGmx address

Submitted by unforgiven, *also found by* eierina, Jeiwan, *and* imare

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexGmx.sol#L269-L293

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexGmx.sol#L346-L355

Impact

Function `configureGmxState()` of `PirexGmx` is for configuring GMX contract state but logic is using `safeApprove()` improperly, it won't reset approval amount for old `stakedGmx` address This would cause 4 problem:

1. Different behavior in `setContract()` and `configureGmxState()` for handling `stakeGmx` address changes. in `setContract()` the logic reset approval for old address to zero but in `configureGmxState()` the logic don't reset old address GMX spending approval.
2. The call to this function would revert if `stakeGmx` address didn't changed but other addresses has been changed so `owner` can't use this to configure contract.
3. Contract won't reset approval for old `stakedGmx` address which is a threat because contract in that address can steal all the GMX balance any time in the future if that old address had been compromised.

4. Contract won't reset approval for old `stakedGmx` address, if `owner` use `configureGmxState()` to change the `stakeGmx` value then it won't be possible to set `stakedGmx` value to previous ones by using either `configureGmxState()` or `setContract()` and contract would be in broken state.

Proof of Concept

This is `configureGmxState()` code in `PirexGmx` :

```
/**
    @notice Configure GMX contract state
 */
function configureGmxState() external onlyOwner whenI
    // Variables which can be assigned by reading pre
    rewardTrackerGmx = RewardTracker(gmxRewardRouterV
    rewardTrackerGlp = RewardTracker(gmxRewardRouterV
    feeStakedGlp = RewardTracker(gmxRewardRouterV2.st
    stakedGmx = RewardTracker(gmxRewardRouterV2.stake
    glpManager = gmxRewardRouterV2.glpManager();
    gmxVault = IVault(IGlpManager(glpManager).vault()

    emit ConfigureGmxState(
        msg.sender,
        rewardTrackerGmx,
        rewardTrackerGlp,
        feeStakedGlp,
        stakedGmx,
        glpManager,
        gmxVault
    );

    // Approve GMX to enable staking
    gmx.safeApprove(address(stakedGmx), type(uint256)
}
```

As you can see it just sets the approval for new `stakeGmx` address and doesn't do anything about spending approval of old `stakeGmx` address.

This is part of the `setContract()` code that handels `stakeGmx` :

```
if (c == Contracts.StakedGmx) {
    // Set the current stakedGmx (pending change)
    gmx.safeApprove(address(stakedGmx), 0);

    stakedGmx = RewardTracker(contractAddress);

    // Approve the new stakedGmx contract address
    gmx.safeApprove(contractAddress, type(uint256
    return;
}
```

As you can see it resets the spending approval of old `stakedGmx` address to zero and then give unlimited spending approval for new address.

So the impact #1 is obvious that the code for same logic in two different functions don't behave similarly.

Function `configureGmxState()` is used for configuring GMX contract state but if `owner` uses this function one time then it won't be possible to call this function the second time if `stakedGmx` wasn't changed. For example in this scenario:

1. Owner calls `configureGmxState()` to set values for GMX contract addresses.
2. Then address of one of contract changes in GMX (`stakedGmx` stayed the same) and owner wants to call `configureGmxState()` to reset the values of variables to correct ones.
3. Owner call to `configureGmxState()` would fail because `stakedGmx` address didn't change and in the line `gmx.safeApprove(address(stakedGmx), type(uint256).max);` contract tries to set non zero approval for `stakedGmx` but it already has non zero spending allowance. (`safeApprove()` would revert if the

current spending allowance is not zero and the new allowance is not zero either).

So the impact #2 would happen and calls to this function in some scenarios would fail and it won't be functional.

Because function `configureGmxState()` doesn't reset the old `stakeGmx` addresses GMX token spending approval to 0x0 so it would be possible to lose all GMX balance of `PirexGmx` contract if the old `stakeGmx` addresses are compromised. For example in this scenario:

1. GMX protocol get hacked (either by a bug or leaking some private keys) and `stakeGmx` contract control would be in hacker's hand.
2. GMX deploy new contracts and `stakeGmx` address changes in `GMX.()`
3. owner of `PirexGmx` calls `configureGmxState()` to reset the values of GMX contracts addresses in `PirexGmx` .
4. Function `configureGmxState()` logic would change the GMX contract addresses but it won't set GMX token spending approval for old `stakeGmx` address to zero.
5. Hacker who control the old `stakeGmx` address would use his access to that address contract to withdraw GMX balance of `PirexGmx` .

B `PirexGmx` won't set approval for old `stakeGmx` contract so it would be possible for that old `stakeGmx` address to transfer GMX balance of `PirexGmx` anytime in future. The bug in old `stakeGmx` contract or leakage of private keys of `stakeGmx` address (private key who can become the owner or admin of that contract) can happen after migrating GMX and Pirex contracts too. This is impact #3 scenario.

In scenario #4, contract would be stuck in unrecoverable state. The problem is that if `configureGmxState()` gets used more than once and `stakeGmx` variable's value has been changes more than once then it won't be possible to set `stakeGmx` value to old values either with `configureGmxState()` or `setContract()` and the contract won't be useful. The scenario is this:

(`safeApprove()` won't allow to set non-zero approval for address that has already non-zero approval amount. See the OZ implementation)

1. GMX protocol changes its `stakeGmx` contract address from old address to new (for any reason, they got hacked or they are migrating or ...)
2. `owner` of `PirexGmx` calls `configureGmxState()` to update GMX protocol's contract address in `PirexGmx` contract and the logic would update the values of variables. (The GMX spending approval for old and new `stakeGmx` address would be max value).
3. GMX protocol changes `stakeGmx` contract address from new value to old value (for any reason, they decided to roll back to old address).
4. Owner tries to call `configureGmxState()` to reupdate GMX protocol's address in `PirexGmx` but the call would revert because the code tries to call `safeApprove()` for address that has already non-zero allowance.
5. Owner can't call `setContract()` to update value of `stakeGmx` variable because this function tries to call `safeApprove()` to set non-zero approval value for address that already has non-zero allowance.

So in this state `owner` can't recover `PirexGmx` contract and because contract has wrong value for `stakeGmx` it won't be functional and it would stay in broken state.

Tools Used

VIM

Recommended Mitigation Steps

Like `setContract()`, function `configureGmxState()` should set approval for old `PirexGmx` to zero first.

drahrealm (Redacted Cartel) confirmed, but disagreed with severity and commented:

> The method was meant to be called only once (ie. right before unpausing the contract to make it live). To make it clearer, we will change the method name to `initializeGmxState` instead.

Picodes (judge) commented:

> Maybe you could add a flag to prevent the method from being called multiple times as well.

[M-10] `_calculateRewards()` in PirexGmx don't handle reward calculation properly, and it would revert when `totalSupply()` is zero which will cause `claimRewards()` to revert if one of 4 rewardTracker's totalSupply was zero

Submitted by **unforgiven**, *also found by* **8olidity**

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexGmx.sol#L733-L816

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexGmx.sol#L228-L267

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexRewards.sol#L332-L348

Impact

Function `claimRewards()` in `PirexGmx` claims WETH and esGMX rewards and multiplier points (MP) from GMX protocol. it uses `_calculateRewards()` to calculate the unclaimed reward token amounts produced for each token type. But because of the lack of checks, function

`_calculateRewards()` would revert when `RewardTracker.distributor.totalSupply()` is zero so if any of the 4 `RewardTracker` s has zero `totalSupply()` then function `claimRewards()` would revert too and function `harvest()` in `PirexRewards` contract would revert too because it calls `PirexGmx.claimRewards()` .

`harvest()` is used in `claim()` function so `claim()` would not work too. This bug would harvest rewards for `PirexRewards` contract and claim rewards for users from `PirexRewards` when supply of one of `RewardTracker` s contracts in GMX protocol is zero.

Function `claimRewards()` is written based on GMX code, but the logic is not correctly copied because GMX protocol contract checks for `totalSupply()` and it prevents this bug from happening.

Proof of Concept

This is function `_calculateRewards()` 's code in `PirexGmx` :

```
function _calculateRewards(bool isBaseReward, bool use
    internal
    view
    returns (uint256)
{
    RewardTracker r;

    if (isBaseReward) {
        r = useGmx ? rewardTrackerGmx : rewardTracker
    } else {
        r = useGmx ? stakedGmx : feeStakedGlp;
    }

    address distributor = r.distributor();
    uint256 pendingRewards = IRewardDistributor(distr
        .pendingRewards();
    uint256 distributorBalance = (isBaseReward ? gmxB
        .balanceOf(distributor);
    uint256 blockReward = pendingRewards > distributo
```

```
            ? distributorBalance
            : pendingRewards;
        uint256 precision = r.PRECISION();
        uint256 cumulativeRewardPerToken = r.cumulativeRe
            ((blockReward * precision) / r.totalSupply())

        if (cumulativeRewardPerToken == 0) return 0;

        return
            r.claimableReward(address(this)) +
            ((r.stakedAmounts(address(this)) *
                (cumulativeRewardPerToken -
                    r.previousCumulatedRewardPerToken(adc
                precision);
    }
```

As you can see in the line `uint256 cumulativeRewardPerToken = r.cumulativeRewardPerToken() + ((blockReward * precision) / r.totalSupply())` if `totalSupply()` was zero then code would revert because of division by zero error. So if `RewardTracker.distributor.totalSupply()` was zero then function `_calculateRewards` would revert and won't work and other function using `_calculateRewards()` would be break too.

This is part of function `claimRewards()`'s code in `PirexGmx` contract:

```
    function claimRewards()
        external
        onlyPirexRewards
        returns (
            ERC20[] memory producerTokens,
            ERC20[] memory rewardTokens,
            uint256[] memory rewardAmounts
        )
    {
        // Assign return values used by the PirexRewards
        producerTokens = new ERC20[](4);
        rewardTokens = new ERC20[](4);
        rewardAmounts = new uint256[](4);
```

```
        producerTokens[0] = pxGmx;
        producerTokens[1] = pxGlp;
        producerTokens[2] = pxGmx;
        producerTokens[3] = pxGlp;
        rewardTokens[0] = gmxBaseReward;
        rewardTokens[1] = gmxBaseReward;
        rewardTokens[2] = ERC20(pxGmx); // esGMX rewards
        rewardTokens[3] = ERC20(pxGmx);

        // Get pre-reward claim reward token balances to
        uint256 baseRewardBeforeClaim = gmxBaseReward.bal
        uint256 esGmxBeforeClaim = stakedGmx.depositBalar
            address(this),
            address(esGmx)
        );

        // Calculate the unclaimed reward token amounts p
        uint256 gmxBaseRewards = _calculateRewards(true,
        uint256 glpBaseRewards = _calculateRewards(true,
        uint256 gmxEsGmxRewards = _calculateRewards(false
        uint256 glpEsGmxRewards = _calculateRewards(false
```

As you can see it calls `_calculateRewards()` to calculate the unclaimed reward token amounts produced for each token type in GMX protocol. so function `claimRewards()` would revert too when `totalSupply()` of one of these 4 `RewardTracker` 's distributers was zero.

This is part of functions `harvest()` and `claim()` code in `PirexReward` contract:

```
    function harvest()
        public
        returns (
            ERC20[] memory _producerTokens,
            ERC20[] memory rewardTokens,
            uint256[] memory rewardAmounts
        )
    {
        (_producerTokens, rewardTokens, rewardAmounts) =
```

```
                .claimRewards();
            uint256 pLen = _producerTokens.length;
    .......
    ......
    ......


        function claim(ERC20 producerToken, address user) ext
            if (address(producerToken) == address(0)) revert
            if (user == address(0)) revert ZeroAddress();

            harvest();
            userAccrue(producerToken, user);
    ....
    ....
    ....
```

As you can see `harvest()` calls `claimRewards()` and `claim()` calls `harvest()` so these two functions would revert and won't work when `totalSupply()` of one of these 4 `RewardTracker` 's distributers in GMX protocol was zero. In that situation the protocol can't harvest and claim rewards from GMX because of this bug and users won't be able to claim their rewards from the protocol. The condition for this bug could happen from time to time as GMX decided to prevent it by checking the value of `totalSupply()`.

This is function `_updateRewards()` code in `RewardTracker` in GMX protocol (https://github.com/gmx-io/gmx-contracts/blob/65e62b62aadea5baca48b8157acb9351249dbaf1/contracts/staking/RewardTracker.sol#L272-L286):

```
        function _updateRewards(address _account) private {
            uint256 blockReward = IRewardDistributor(distribu

            uint256 supply = totalSupply;
            uint256 _cumulativeRewardPerToken = cumulativeRew
            if (supply > 0 && blockReward > 0) {
                _cumulativeRewardPerToken = _cumulativeRewarc
```

```
                cumulativeRewardPerToken = _cumulativeRewardP
            }

            // cumulativeRewardPerToken can only increase
    ....
    ....
```

As you can see it checks that `supply > 0` before using it as denominator in division. So GMX protocol handles the case when `totalSupply()` is zero and contract logics won't break when this case happens but function `_calculateRewards()`, which tries to calculate GMX protocol rewards beforehand, don't handle this case(the case where `totalSupply()` is zero) so this logics would break when this case happens and it would cause function `harvest()` and `claim()` to be unfunctional.

Tools Used

VIM

Recommended Mitigation Steps

Check that `totalSupply()` is not zero before using it.

drahrealm (Redacted Cartel) confirmed

[M-11] PirexGmx#migrateReward() may cause users to lose Reward.

Submitted by bin2chen

`PirexGmx#migrateReward()` may cause users to lose Reward before PirexRewards.sol set new PirexGmx.

Proof of Concept

The current migration process is: `call # completemigration ()-> # migrateward ()`

After this method, the producer of PirexRewards.sol contract is still the old PirexGmx.

At this time, if `AutoPxGmx#compound ()` is called by bot:

`AutoPxGmx#compound() -> PirexRewards#.claim() -> old_PirexGmx#claimRewards()`

`Old_PirexGmx#claimRewards ()` will return zero rewards and the reward of AutopXGMX will be lost.

Old PirexGmx still can execute https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexGmx.sol#L824-L828

https://github.com/code-423n4/2022-11-redactedcartel/blob/03b71a8d395c02324cb9fdaf92401357da5b19d1/src/PirexGmx.sol#L940-L949

Recommended Mitigation Steps

There are two ways to solve the problem.

1. Set the producer of PirexRewards to the new PirexGmx in `completeMigration ()`.
2. In `#migrateReward ()`, set the old PirexGmx's "pirexRewards" to `address(0)`, so that you can't use the old PirexGmx to get rewards

Simply use the second, such as:

```
function migrateReward() external whenPaused {
    if (msg.sender != migratedTo) revert NotMigratedT
    if (gmxRewardRouterV2.pendingReceivers(address(th
```

```
            revert PendingMigration();

        // Transfer out any remaining base reward (ie. WE
        gmxBaseReward.safeTransfer(
            migratedTo,
            gmxBaseReward.balanceOf(address(this))
        );
+       pirexRewards ==address(0);   //*** set pirexRewar
    }
```

drahrealm (Redacted Cartel) confirmed

[M-12] Reward tokens mismanagement can cause users losing rewards

Submitted by Jeiwan, *also found by* pashov, cryptonue, Oxbepresent, __141345__, unforgiven, cryptoDave, Koolex, *and* datapunk

A user (which can also be one of the autocompounding contracts, `AutoPxGlp` or `AutoPxGmx`) can lose a reward as a result of reward tokens mismanagement by the owner.

Proof of Concept

The protocol defines a short list of reward tokens that are hard coded in the `claimRewards` function of the `PirexGmx` contract (PirexGmx.sol#L756-L759):

```
        rewardTokens[0] = gmxBaseReward;
        rewardTokens[1] = gmxBaseReward;
        rewardTokens[2] = ERC20(pxGmx); // esGMX rewards distribu
        rewardTokens[3] = ERC20(pxGmx);
```

The fact that these addresses are hard coded means that no other reward tokens will be supported by the protocol. However, the `PirexRewards`

contract maintains a different list of reward tokens, one per producer token (PirexRewards.sol#L19-L31):

```
    struct ProducerToken {
        ERC20[] rewardTokens;
        GlobalState globalState;
        mapping(address => UserState) userStates;
        mapping(ERC20 => uint256) rewardStates;
        mapping(address => mapping(ERC20 => address)) rewardF
    }

    // Producer tokens mapped to their data
    mapping(ERC20 => ProducerToken) public producerTokens;
```

These reward tokens can be added (PirexRewards.sol#L151) or removed (PirexRewards.sol#L179) by the owner, which creates the possibility of mismanagement:

1. The owner can mistakenly remove one of the reward tokens hard coded in the `PirexGmx` contract;
2. The owner can add reward tokens that are not supported by the `PirexGmx` contract.

Such mismanagement can cause users to lose rewards for two reasons:

1. Reward state of a user is updated *before* their rewards are claimed;
2. It's the reward token addresses set by the owner of the `PirexRewards` contract that are used to transfer rewards.

In the `claim` function:

1. `harvest` is called to pull rewards from GMX (PirexRewards.sol#L377):

```
        harvest();
```

2. `claimReward` is called on `PirexGmx` to pull rewards from GMX and get the hard coded lists of producer tokens, reward tokens, and amounts (PirexRewards.sol#L346-L347):

```
(_producerTokens, rewardTokens, rewardAmounts) = producer
.claimRewards();
```

3. Rewards are recorded for each of the hard coded reward token (PirexRewards.sol#L361):

```
if (r != 0) {
    producerState.rewardStates[rewardTokens[i]] += r;
}
```

4. Later in the `claim` function, owner-set reward tokens are read (PirexRewards.sol#L386-L387):

```
ERC20[] memory rewardTokens = p.rewardTokens;
uint256 rLen = rewardTokens.length;
```

5. User reward state is set to 0, which means they've claimed their entire share of rewards (PirexRewards.sol#L391), however this is done before a reward is actually claimed:

```
p.userStates[user].rewards = 0;
```

6. The owner-set reward tokens are iterated and the previously recorded rewards are distributed (PirexRewards.sol#L396-L415):

```
for (uint256 i; i < rLen; ++i) {
    ERC20 rewardToken = rewardTokens[i];
    address rewardRecipient = p.rewardRecipients[user][rewardTo
    address recipient = rewardRecipient != address(0)
```

```
                ? rewardRecipient
                : user;
            uint256 rewardState = p.rewardStates[rewardToken];
            uint256 amount = (rewardState * userRewards) / globalReward

            if (amount != 0) {
                // Update reward state (i.e. amount) to reflect reward
                p.rewardStates[rewardToken] = rewardState - amount;

                producer.claimUserReward(
                    address(rewardToken),
                    amount,
                    recipient
                );
            }
        }
```

In the above loop, there can be multiple reasons for rewards to not be sent:

1. One of the hard coded reward tokens is missing in the owner-set reward tokens list;
2. The owner-set reward token list contains a token that's not supported by `PirexGmx` (i.e. it's not in the hard coded reward tokens list);
3. The `rewardTokens` array of a producer token turns out to be empty due to mismanagement by the owner.

In all of the above situations, rewards won't be sent, however user's reward state will still be set to 0.

Also, notice that calling `claim` won't revert if reward tokens are misconfigured, and the `Claim` event will be emitted successfully, which makes reward tokens mismanagement hard to detect.

The amount of lost rewards can be different depending on how much GMX a user has staked and how often they claim rewards. Of course, if a mistake isn't detected quickly, multiple users can suffer from this issue. The autocompounding contracts (`AutoPxGlp` and `AutoPxGmx`) are also users of the protocol, and since they're intended to hold big amounts of real users' deposits (they'll probably be the biggest stakers), lost rewards can be big.

Recommended Mitigation Steps

Consider having one source of reward tokens. Since they're already hard coded in the `PirexGmx` contract, consider exposing them so that `PirexRewards` could read them in the `claim` function. This change will also mean that the `addRewardToken` and `removeRewardToken` functions won't be needed, which makes contract management simpler.

Also, in the `claim` function, consider updating global and user reward states only after ensuring that at least one reward token was distributed.

drahrealm (Redacted Cartel) disagreed with severity and commented:

> To make sure this won't be an issue, we will add the `whenNotPaused` modifier to `claimUserReward` method in `PirexGmx` . Also, as `migrateRewards` is going to be updated to also set the `pirexRewards` address to 0, it will defer any call to claim the rewards

Picodes (judge) commented:

> Seems to be the mitigation for **#249**.

Low Risk and Non-Critical Issues

For this contest, 60 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **0xSmartContract** received the top score from the judge.

The following wardens also submitted reports: brgltd, Deivitto, Awesome, eierina, jadezti, 0xNazgul, adriro, danyams, delfin454000, rotcivegaf, sakshamguruji, rbserver, Waze, Josiah, hansfriese, gz627, oyc_109, keccak123, B2, ch0bu, 0xbepresent, cryptostellar5, Diana, Funen, 0xfuje, pedr02b2, nameruse, deliriusz, Jeiwan, joestakey, unforgiven, xiaoming90, shark, erictee, JohnSmith, 0xPanda, btk, 0xAgro, gzeon, hihen, carrotsmuggler, R2, subtle77, codeislight, simon135, Rolezn,

aphak5010, csanuragjain, datapunk, martin, Sathish9098, yixxas, perseverancesuccess, fatherOfBlocks, rvierdiiev, codexploder, chaduke, Bnke0x0, *and* RaymondFam .

Low Risk Issues Summary

Number	Issues Details	Context
[L-01]	PirexERC4626's implmentation is not fully up to EIP-4626's specification	1
[L-02]	initialize() function can be called by anybody	1
[L-03]	Solmate's SafeTransferLib doesn't check whether the ERC20 contract exists	19
[L-04]	Use `Ownable2StepUpgradeable` instead of `OwnableUpgradeable` contract	1
[L-05]	Owner can renounce Ownership	1
[L-06]	Critical Address Changes Should Use Two-step Procedure	4
[L-07]	`DepositGlp` Event arguments names are confusing	2
[L-08]	Loss of precision due to rounding	1
[L-09]	First value check of argument of type enum in setFee function is missing	1
[L-10]	Hardcode the address causes no future updates	1
[L-11]	Lack of Input Validation	6

Total: 11 issues

[L-01] PirexERC4626's implmentation is not fully up to EIP-4626's specification

Must return the maximum amount of shares mint would allow to be deposited to receiver and not cause a revert, which must not be higher than the actual maximum that would be accepted (it should underestimate if necessary).

This assumes that the user has infinite assets, i.e. must not rely on `balanceOf` of asset.

```
src/vaults/PirexERC4626.sol:
  216
  217:     function maxDeposit(address) public view virtu
  218:         return type(uint256).max;
  219:     }
  220:
  221:     function maxMint(address) public view virtual
  222:         return type(uint256).max;
  223:     }
```

Could cause unexpected behavior in the future due to non-compliance with EIP-4626 standard.

Similar problem for Sentimentxyz:
https://github.com/sentimentxyz/protocol/pull/235/files

Recommended Mitigation Steps

`maxMint()` and `maxDeposit()` should reflect the limitation of maxSupply.

Consider changing `maxMint()` and `maxDeposit()` to:

```
function maxMint(address) public view virtual returns (ui
    if (totalSupply >= maxSupply) {
        return 0;
    }
    return maxSupply - totalSupply;
}
function maxDeposit(address) public view virtual returns
    return convertToAssets(maxMint(address(0)));
}
```

[L-02] `initialize()` function can be called by anybody

`initialize()` function can be called by anybody when the contract is not initialized.

More importantly, if someone else runs this function, they will have full authority because of the `__Ownable_init()` function.

Here is a definition of `initialize()` function.

```
src/PirexRewards.sol:
  84
  85:     function initialize() public initializer {
  86:         __Ownable_init();
  87:     }
```

Recommended Mitigation Steps

Add a control that makes `initialize()` only call the Deployer Contract or EOA;

```
if (msg.sender != DEPLOYER_ADDRESS) {
            revert NotDeployer();
        }
```

[L-03] Solmate's SafeTransferLib doesn't check whether the ERC20 contract exists

Solmate's SafeTransferLib, which is often used to interact with non-compliant/unsafe ERC20 tokens, does not check whether the ERC20 contract exists. The following code will not revert in case the token doesn't exist (yet).

This is stated in the Solmate library: https://github.com/transmissions11/solmate/blob/main/src/utils/SafeTransferLib.sol#L9

```
19 results

src/PirexFees.sol:
    5: import {SafeTransferLib} from "solmate/utils/SafeT

  114:         token.safeTransfer(treasury, treasuryDistr
  115:         token.safeTransfer(contributors, contribut

src/PirexGmx.sol:
    7: import {SafeTransferLib} from "solmate/utils/SafeT

  392:         gmx.safeTransferFrom(msg.sender, address(t

  506:             t.safeTransferFrom(msg.sender, address

  643:             ERC20(pxGlp).safeTransferFrom(

  844:             gmxBaseReward.safeTransfer(receiver, p

  847:                 gmxBaseReward.safeTransfer(address

  946:         gmxBaseReward.safeTransfer(



src/vaults/AutoPxGlp.sol:
    6: import {SafeTransferLib} from "solmate/utils/SafeT


  258:                 asset.safeTransfer(msg.sender, pxG
  259
  260:             asset.safeTransfer(owner, totalPxGlpFe

  274:                 pxGmx.safeTransfer(msg.sender, pxG
  275
  276:             pxGmx.safeTransfer(owner, totalPxGmxFe

  344:         stakedGlp.safeTransferFrom(msg.sender, add
```

```
387:         erc20Token.safeTransferFrom(msg.sender, ac
388

src/vaults/AutoPxGmx.sol:
  7: import {SafeTransferLib} from "solmate/utils/SafeT

297:             if (incentive != 0) asset.safeTransfer
298
299:             asset.safeTransfer(owner, totalFee - i

336:         asset.safeTransfer(receiver, assets);

361:         asset.safeTransfer(receiver, assets);

382:         gmx.safeTransferFrom(msg.sender, address(t
```

Recommended Mitigation Steps

Add a contract exist control in functions;

```
pragma solidity >=0.8.0;

function isContract(address _addr) private returns (bool
    isContract = _addr.code.length > 0;
}
```

[L-04] Use `Ownable2StepUpgradeable` instead of `OwnableUpgradeable` contract

PirexRewards.sol#L4

`transferOwnership` function is used to change Ownership from `OwnableUpgradeable.sol` .

There is another Openzeppelin Ownable contract (Ownable2StepUpgradeable.sol) has `transferOwnership` function , use it is

more secure due to 2-stage ownership transfer.

[Ownable2StepUpgradeable.sol]

[L-05] Owner can renounce Ownership

[PirexRewards.sol#L4]

Typically, the contract's owner is the account that deploys the contract. As a result, the owner is able to perform certain privileged activities.

The Openzeppelin's Ownable used in this project contract implements renounceOwnership. This can represent a certain risk if the ownership is renounced for any other reason than by design. Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner.

`onlyOwner` functions;

```
src/PirexRewards.sol:
  93:     function setProducer(address _producer) externa
151:     function addRewardToken(ERC20 producerToken, ERC
```

Recommended Mitigation Steps

We recommend to either reimplement the function to disable it or to clearly specify if it is part of the contract design.

[L-06] Critical Address Changes Should Use Two-step Procedure

The critical procedures should be a two-step process.

```
src/PirexFees.sol:
  63:     function setFeeRecipient(FeeRecipient f, addres
```

```
src/PirexGmx.sol:
  313:     function setContract(Contracts c, address cont
  884:     function setVoteDelegate(address voteDelegate)

src/external/DelegateRegistry.sol:
  18:     function setDelegate(bytes32 id, address delega
```

Recommended Mitigation Steps

Lack of two-step procedure for critical operations leaves them error-prone. Consider adding two step procedure on the critical functions.

[L-07] `DepositGlp` Event arguments names are confusing

The `uint256 amount` argument of the `depositFsGlp` function in the PirexGmx.sol contract is named `deposited` in the `event DepositGlp` parameter. In `emit DepositGlp` this causes confusion in terms of user, code readability and web pages.

```
src/PirexGmx.sol:
  421       */
  422:     function depositFsGlp(uint256 amount, address
  423:         external
  424:         whenNotPaused
  425:         nonReentrant
  426:         returns (
  427:             uint256,
  428:             uint256,
  429:             uint256
  430:         )

  452:         emit DepositGlp(
  453:             msg.sender,                    // caller
  454:             receiver,                      // receiv
  455:             address(stakedGlp),            // token
  456:             0,                                    /
  457:             0,                              // minUs
```

```
458:             0,                                          /
459:             amount,                              // depos
460:             postFeeAmount,           // postFeeAmount
461:             feeAmount                // feeAmount
462:         );
```

Recommended Mitigation Steps

Event-Emit parameter names must match the function arguments or the argument they take value from.

[L-08] Loss of precision due to rounding

```
src/PirexGmx.sol:
  221      {
  222:         feeAmount = (assets * fees[f]) / FEE_DENOM
  223          postFeeAmount = assets - feeAmount;;
```

[L-09] First value check of argument of type enum in setFee function is missing

```
src/PirexGmx.sol:
  298          @param  fee  uint256  Fee amount
  299:      */
  300:     function setFee(Fees f, uint256 fee) external
  301:         if (fee > FEE_MAX) revert InvalidFee();
  302:
  303:         fees[f] = fee;
  304:
  305:         emit SetFee(f, fee);
  306:     }
```

Leaving the enum type argument `Fees` in the setFee function empty and returning the value `0` gives the same result as returning the value `0`, this is

a property of the enum type and therefore error-prone.

Recommended Mitigation Steps

Use struck instead of enum.

[L-10] Hardcode the address causes no future updates

```
src/vaults/AutoPxGmx.sol:
  18      IV3SwapRouter public constant SWAP_ROUTER =
  19:         IV3SwapRouter(0x68b3465833fb72A70ecDF485E0e
```

Router etc. In case the addresses change due to reasons such as updating their versions in the future, addresses coded as constants cannot be updated, so it is recommended to add the update option with the onlyOwner modifier.

[L-11] Lack of Input Validation

For defence-in-depth purpose, it is recommended to perform additional validation against the amount that the user is attempting to deposit, mint, withdraw and redeem to ensure that the submitted amount is valid.

[OpenZeppelinTokenizedVault.sol#L9]

```
src/PirexGmx.sol:
  429       */
  430:     function depositGmx(uint256 amount, address re
  431:         external
  432:         whenNotPaused
  433:         nonReentrant
  434:         returns (
  435:             uint256,
  436:             uint256,
  437:             uint256
  438:         )
```

```
  439:     {
+                require(amount <= maxDeposit(receiver),
```

```
src/vaults/PirexERC4626.sol:
  79
  80:     function mint(uint256 shares, address receiver)
  81:         public
  82:         virtual
  83:         returns (uint256 assets)
  84:     {
+              require(shares <= maxMint(receiver), "mint
```

```
src/vaults/AutoPxGlp.sol:
  438:     function withdraw(
  439:         uint256 assets,
  440:         address receiver,
  441:         address owner
  442:     ) public override returns (uint256 shares) {
+                require(assets <= maxWithdraw(owner), "w
```

```
src/vaults/AutoPxGmx.sol:
  317:     function withdraw(
  318:         uint256 assets,
  319:         address receiver,
  320:         address owner
  321:     ) public override returns (uint256 shares) {
+                require(assets <= maxWithdraw(owner), "wi
```

```
src/vaults/AutoPxGlp.sol:
  450       */
  451:     function redeem(
  452:         uint256 shares,
  453:         address receiver,
  454:         address owner
  455:     ) public override returns (uint256 assets) {
+                require(shares <= maxRedeem(owner), "rede
```

```
src/vaults/AutoPxGmx.sol:
  340
  341:     function redeem(
  342:         uint256 shares,
  343:         address receiver,
  344:         address owner
  345:     ) public override returns (uint256 assets) {
+                 require(shares <= maxRedeem(owner), "rec
```

Non-Critical Issues Summary

Number	Issues Details	Context
[N-01]	Insufficient coverage	1
[N-02]	Not using the named return variables anywhere in the function is confusing	1
[N-03]	Some Constant redefined elsewhere	4
[N-04]	Omissions in Events	8
[N-05]	Add parameter to Event-Emit	1
[N-06]	NatSpec is missing	1
[N-07]	Use `require` instead of `assert`	1
[N-08]	Implement some type of version counter that will be incremented automatically for contract upgrades	1
[N-09]	Constant values such as a call to keccak256(), should used to immutable rather than constant	2
[N-10]	For functions, follow Solidity standard naming conventions	4
[N-11]	Mark visibility of initialize(...) functions as `external`	1
[N-12]	No same value input control	8
[N-13]	Include `return parameters` in *NatSpec comments*	All
[N-14]	`0 address` check for `asset`	1
[N-15]	Use a single file for all system-wide constants	6
[N-16]	`Function writing` that does not comply with the `Solidity Style Guide`	All

Number	Issues Details	Context
[N-17]	Missing Upgrade Path for `PirexRewards` Implementation	1
[N-18]	No need `assert` check in `_computeAssetAmounts()`	1
[N-19]	Lack of event emission after critical `initialize()` functions	1
[N-20]	Add a timelock to critical functions	11

Total 19 issues

[N-01] Insufficient coverage

Testing all functions is best practice in terms of security criteria.

```
| File                                | % Lines            |
|-------------------------------------|--------------------|
| src/PirexRewards.sol                | 98.98% (97/98)     |
| src/PxGmx.sol                       | 100.00% (0/0)      |
| src/vaults/AutoPxGlp.sol            | 91.11% (82/90)     |
| src/vaults/AutoPxGmx.sol            | 89.39% (59/66)     |
| src/vaults/PirexERC4626.sol         | 75.00% (39/52)     |
| src/vaults/PxGmxReward.sol          | 78.12% (25/32)     |
| Total                               | 56.72% (523/922)   |
```

Due to its capacity, test coverage is expected to be 100%.

[N-02] Not using the named return variables anywhere in the function is confusing

```
function getUserState(ERC20 producerToken, address user)
        external
        view
        returns (
            uint256 lastUpdate,
            uint256 lastBalance,
            uint256 rewards
```

```
        )
    {
        UserState memory userState = producerTokens[produ
            user
        ];

        return (userState.lastUpdate, userState.lastBalar
    }
```

Recommended Mitigation Steps

Consider adopting a consistent approach to return values throughout the codebase by removing all named return variables, explicitly declaring them as local variables, and adding the necessary return statements where appropriate. This would improve both the explicitness and readability of the code, and it may also help reduce regressions during future code refactors.

[N-03] Same Constant redefined elsewhere

Keeping the same constants in different files may cause some problems or errors, reading constants from a single file is preferable. This should also be preferred in gas optimization

```
src/vaults/AutoPxGlp.sol:
  17
  18:     uint256 public constant MAX_WITHDRAWAL_PENALTY
  19:     uint256 public constant MAX_PLATFORM_FEE = 200
  20:     uint256 public constant FEE_DENOMINATOR = 1000
  21:     uint256 public constant MAX_COMPOUND_INCENTIVE

src/vaults/AutoPxGmx.sol:

  20:     uint256 public constant MAX_WITHDRAWAL_PENALTY
  21:     uint256 public constant MAX_PLATFORM_FEE = 200
  22:     uint256 public constant FEE_DENOMINATOR = 1000
  23:     uint256 public constant MAX_COMPOUND_INCENTIVE
```

[N-04] Omissions in Events

Throughout the codebase, events are generally emitted when sensitive changes are made to the contracts.

However, some events are missing important parameters. The events should include the new value and old value where possible:

Events with no old value;

```
8 results

src/PirexFees.sol:
  63:     function setFeeRecipient(FeeRecipient f, addres

  83:     function setTreasuryFeePercent(uint8 _treasuryF

src/PirexGmx.sol:
  300:     function setFee(Fees f, uint256 fee) external
  301          if (fee > FEE_MAX) revert InvalidFee();

  313:     function setContract(Contracts c, address cont

  862:     function setDelegationSpace(
  863          string memory _delegationSpace,

  884:     function setVoteDelegate(address voteDelegate)
  885          if (voteDelegate == address(0)) revert Zer

  909:     function setPauseState(bool state) external or
  910          if (state) {

src/PirexRewards.sol:
   93:     function setProducer(address _producer) exterr
   94          if (_producer == address(0)) revert ZeroAc
```

[N-05] Add parameter to Event-Emit

Some event-emit description has no parameter. Add to parameter for front-end website or client app, they can see that something has happened on the blockchain.

Events with no old value;

```
src/PirexGmx.sol:
  894        */
  895:      function clearVoteDelegate() public onlyOwner
  896:          emit ClearVoteDelegate();
```

[N-06] NatSpec is missing

NatSpec is missing for the following functions, constructor and modifier:

```
src/vaults/PxGmxReward.sol:
  15:     ERC20 public pxGmx;
  17:     GlobalState public globalState;
  18:     uint256 public rewardState;
  19:     mapping(address => UserState) public userRewarc
```

[N-07] Use `require` instead of `assert`

```
src/PirexGmx.sol:
  224
  225:          assert(feeAmount + postFeeAmount == assets
  226       }
```

Assert should not be used except for tests, `require` should be used

Prior to Solidity 0.8.0, pressing a confirm consumes the remainder of the process's available gas instead of returning it, as request()/revert() did.

`assert()` and `reqire()`;

The big difference between the two is that the `assert()` function when false, uses up all the remaining gas and reverts all the changes made.

Meanwhile, a `require()` function when false, also reverts back all the changes made to the contract but does refund all the remaining gas fees we offered to pay.This is the most common Solidity function used by developers for debugging and error handling.

`Assertion()` should be avoided even after solidity version 0.8.0, because its documentation states “The Assert function generates an error of type Panic(uint256). Code that works properly should never Panic, even on invalid external input. If this happens, you need to fix it in your contract. there’s a mistake”.

[N-08] Implement some type of version counter that will be incremented automatically for contract upgrades

[PirexRewards.sol#L85]

I suggest implementing some kind of version counter that will be incremented automatically when you upgrade the contract.

Recommended Mitigation Steps

```
    uint256 public authorizeUpgradeCounter;

     function upgradeTo(address _newImplementation) external
            _setPendingImplementation(_newImplementation);
          authorizeUpgradeCounter+=1;

        }
```

[N-09] Constant values such as a call to `keccak256()`, should used to immutable rather than constant

There is a difference between constant variables and immutable variables, and they should each be used in their appropriate contexts.

While it doesn't save any gas because the compiler knows that developers often make this mistake, it's still best to use the right tool for the task at hand.

Constants should be used for literal values written into the code, and immutable variables should be used for expressions, or values calculated in, or passed into the constructor.

```
2 results

src/PxERC20.sol:
    9:     bytes32 public constant MINTER_ROLE = keccak25
   10:     bytes32 public constant BURNER_ROLE = keccak256
```

[N-10] For functions, follow Solidity standard naming conventions

[AutoPxGlp.sol#L487](#)

[AutoPxGlp.sol#L501](#)

[AutoPxGlp.sol#L474](#)

[AutoPxGlp.sol#L462](#)

The above codes don't follow Solidity's standard naming convention:

internal and private functions: the mixedCase format starting with an underscore (_mixedCase starting with an underscore)

[N-11] Mark visibility of initialize(...) functions as `external`

L1GraphTokenGateway.sol#L99

If someone wants to extend via inheritance, it might make more sense that the overridden initialize(...) function calls the internal {...}_init function, not the parent public initialize(...) function.

External instead of public would give more the sense of the initialize(...) functions to behave like a constructor (only called on deployment, so should only be called externally)

From a security point of view, it might be safer so that it cannot be called internally by accident in the child contract.

It might cost a bit less gas to use external over public.

It is possible to override a function from external to public (= “opening it up”) ✅

But it is not possible to override a function from public to external (= “narrow it down”). ❌

For above reasons you can change initialize(...) to external

https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3750

[N-12] No same value input control

```
src/PirexFees.sol:
  63:     function setFeeRecipient(FeeRecipient f, addres

  83:     function setTreasuryFeePercent(uint8 _treasuryF

src/PirexGmx.sol:
```

```
313:     function setContract(Contracts c, address cont

884:     function setVoteDelegate(address voteDelegate)

909:     function setPauseState(bool state) external or

src/PirexRewards.sol:
 93:     function setProducer(address _producer) exterr

107:     function setRewardRecipient(

432:     function setRewardRecipientPrivileged(
```

Recommended Mitigation Steps

Add code like this: `if (oracle == _oracle revert ADDRESS_SAME();`

[N-13] Include `return parameters` in *NatSpec comments*

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI). It is clearly stated in the Solidity official documentation. In complex projects such as Defi, the interpretation of all functions and their arguments and returns is important for code readability and auditability.

https://docs.soliditylang.org/en/v0.8.15/natspec-format.html

If Return parameters are declared, you must prefix them with "/// @return".

Some code analysis programs do analysis by reading NatSpec details, if they can't see the "@return" tag, they do incomplete analysis.

Recommended Mitigation Steps

Include return parameters in NatSpec comments

Recommendation Code Style:

```
    /// @notice information about what a function does
    /// @param pageId The id of the page to get the URI fc
    /// @return Returns a page's URI if it has been mintec
    function tokenURI(uint256 pageId) public view virtual
        if (pageId == 0 || pageId > currentId) revert("NOT

        return string.concat(BASE_URI, pageId.toString());
    }
```

[N-14] 0 address check for asset

```
src/vaults/PirexERC4626.sol:
  47
  48:     constructor(
  49:         ERC20 _asset,
  50:         string memory _name,
  51:         string memory _symbol
  52:     ) ERC20(_name, _symbol, _asset.decimals()) {
  53:         asset = _asset;
  54:     }
```

Also check of the address to protect the code from 0x0 address problem just in case. This is best practice or instead of suggesting that they verify address != 0x0, you could add some good NatSpec comments explaining what is valid and what is invalid and what are the implications of accidentally using an invalid address.

Recommended Mitigation Steps

like this; `if (_asset == address(0)) revert ADDRESS_ZERO();`

[N-15] Use a single file for all system-wide constants

There are many addresses and constants used in the system. It is recommended to put the most used ones in one file (for example

constants.sol, use inheritance to access these values)

This will help with readability and easier maintenance for future changes. This also helps with any issues, as some of these hard-coded values are admin addresses.

constants.left

Use and import this file in contracts that require access to these values. This is just a suggestion, in some use cases this may result in higher gas usage in the distribution.

```
21 results - 6 files

src/PirexFees.sol:
  20:     uint8 public constant FEE_PERCENT_DENOMINATOR =
  23:     uint8 public constant MAX_TREASURY_FEE_PERCENT

src/PirexGmx.sol:
  44:     uint256 public constant FEE_DENOMINATOR = 1_000
  47:     uint256 public constant FEE_MAX = 200_000;

src/PxERC20.sol:
   9:     bytes32 public constant MINTER_ROLE = keccak256
  10:     bytes32 public constant BURNER_ROLE = keccak256

src/external/RewardTracker.sol:
  772:     uint256 public constant BASIS_POINTS_DIVISOR =
  773:     uint256 public constant PRECISION = 1e30;
  775:     uint8 public constant decimals = 18;

src/vaults/AutoPxGlp.sol:
  18:     uint256 public constant MAX_WITHDRAWAL_PENALTY
  19:     uint256 public constant MAX_PLATFORM_FEE = 2000
  20:     uint256 public constant FEE_DENOMINATOR = 10000
  21:     uint256 public constant MAX_COMPOUND_INCENTIVE
  22:     uint256 public constant EXPANDED_DECIMALS = 1e3
  23

src/vaults/AutoPxGmx.sol:
```

```
18:     IV3SwapRouter public constant SWAP_ROUTER =
20:     uint256 public constant MAX_WITHDRAWAL_PENALTY
21:     uint256 public constant MAX_PLATFORM_FEE = 200(
22:     uint256 public constant FEE_DENOMINATOR = 1000(
23:     uint256 public constant MAX_COMPOUND_INCENTIVE
```

[NC-16] Function writing that does not comply with the Solidity Style Guide

Order of Functions; ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier. But there are contracts in the project that do not comply with this.

https://docs.soliditylang.org/en/v0.8.17/style-guide.html

Functions should be grouped according to their visibility and ordered:

constructor

receive function (if exists)

fallback function (if exists)

external

public

internal

private

within a grouping, place the view and pure functions last.

[N-17] Missing Upgrade Path for PirexRewards Implementation

```
src/PirexRewards.sol:
   4: import {OwnableUpgradeable} from "openzeppelin-cont
  85:     function initialize() public initializer {
```

With the current and contracts, it is not possible to upgrade the contract.

Recommended Mitigation Steps

Document the operational plan for contract upgrades. Also, consider using the UUPS proxy pattern to upgrade contract implementation.

[N-18] No need `assert` check in `_computeAssetAmounts()`

The `assert` check in this function is not needed. Because this check will always be true because of this line: `postFeeAmount = assets - feeAmount;`

Context:

```
src/PirexGmx.sol:
  216       */
  217:     function _computeAssetAmounts(Fees f, uint256
  218:         internal
  219:         view
  220:         returns (uint256 postFeeAmount, uint256 fe
  221:     {
  222:
  223:
  224:         feeAmount = (assets * fees[f]) / FEE_DENOM
  225:         postFeeAmount = assets - feeAmount;
  226:
- 227:       assert(feeAmount + postFeeAmount == assets);
  228:     }
```

[N-19] Lack of event emission after critical `initialize()` functions

To record the init parameters for off-chain monitoring and transparency reasons, please consider emitting an event after the `initialize()` functions:

```
src/PirexRewards.sol:
  84
  85:     function initialize() public initializer {
  86:         __Ownable_init();
  87:     }
```

[N-20] Add a timelock to critical functions

It is a good practice to give time for users to react and adjust to critical changes. A timelock provides more guarantees and reduces the level of trust required, thus decreasing risk for users. It also indicates that the project is legitimate (less risk of a malicious owner making a sandwich attack on a user).

Consider adding a timelock to:

```
11 results


src/PirexGmx.sol:
  351       */
  352:     function setFee(Fees f, uint256 fee) external
  353         if (fee > FEE_MAX) revert InvalidFee();

  364       */
  365:     function setContract(Contracts c, address cont
  366         external

  918       */
  919:     function setDelegationSpace(
  920         string memory _delegationSpace,
```

```
940         */
941:        function setVoteDelegate(address voteDelegate)
942             if (voteDelegate == address(0)) revert Zer

965         */
966:        function setPauseState(bool state) external or
967             if (state) {
```

```
src/vaults/AutoPxGlp.sol:
 94:        function setWithdrawalPenalty(uint256 penalty)
 95             if (penalty > MAX_WITHDRAWAL_PENALTY) reve

106:        function setPlatformFee(uint256 fee) external
107             if (fee > MAX_PLATFORM_FEE) revert Exceeds
```

```
src/vaults/AutoPxGmx.sol:
104:        function setPoolFee(uint24 _poolFee) external
105             if (_poolFee == 0) revert ZeroAmount();

116:        function setWithdrawalPenalty(uint256 penalty)
117             if (penalty > MAX_WITHDRAWAL_PENALTY) reve

128:        function setPlatformFee(uint256 fee) external
129             if (fee > MAX_PLATFORM_FEE) revert Exceeds

152:        function setPlatform(address _platform) exterr
153             if (_platform == address(0)) revert ZeroAc
```

Suggestions Summary

Number	Suggestion Details	
[S-01]	Generate perfect code headers every time	
[S-02]	Add NatSpec comments to the variables defined in Storage	

Total: 2 suggestions

[S-01] Generate perfect code headers every time

I recommend using header for Solidity code layout and readability:

https://github.com/transmissions11/headers

```
/*////////////////////////////////////////////////////////
                          TESTING 123
////////////////////////////////////////////////////////*/
```

[S-02] Add NatSpec comments to the variables defined in Storage

I recommend adding NatSpec comments explaining the variables defined in Storage, their slots, their contents and definitions.

This improves code readability and control quality

Current Code;

```
src/vaults/AutoPxGlp.sol:
  18:     uint256 public constant MAX_WITHDRAWAL_PENALTY
  19:     uint256 public constant MAX_PLATFORM_FEE = 200(
  20:     uint256 public constant FEE_DENOMINATOR = 1000(
  21:     uint256 public constant MAX_COMPOUND_INCENTIVE
  22:     uint256 public constant EXPANDED_DECIMALS = 1e3
  24:     uint256 public withdrawalPenalty = 300;
  25:     uint256 public platformFee = 1000;
  26:     uint256 public compoundIncentive = 1000;
  27:     address public platform;
  30:     address public immutable rewardsModule;
```

Recommendation Code;

```
//****** Slot 0 ******//
```

```
26:     uint256 public constant MAX_WITHDRAWAL_PENALTY =

/ /****** Slot 1 ******//
30:     uint256 public constant MAX_PLATFORM_FEE = 2000;

 //****** Slot 2 ******//
33:     uint256 public constant FEE_DENOMINATOR = 10000;
...

 /****** End of storage ******/
```

Gas Optimizations

For this contest, 33 reports were submitted by wardens detailing gas optimizations. The **report highlighted below** by **gzeon** received the top score from the judge.

The following wardens also submitted reports: Deivitto, adriro, Tomio, halden, c3phas, cryptonue, B2, ajtra, oyc_109, karanctf, __141345__, Diana, dharma09, unforgiven, PaludoX0, sakshamguruji, keccak123, saneryee, JohnSmith, 0xPanda, 0xSmartContract, ReyAdmirado, codeislight, Rolezn, aphak5010, datapunk, Rahoz, chaduke, Schlagatron, Secureverse, pavankv, *and* RaymondFam .

Since GMX is on Arbitrum, a L2 rollup, it is much more important to optimize for calldata when considering gas optimization. In fact, EVM gas optimizations have minimal effect when compared to calldata optimizations. For example at 9 gwei L1 gas price and 0.1 gwei Arbitrum gas price, each byte of calldata after compression cost ~640 calldata gas. Here are some suggestions to reduce the calldata size:

1. For all the functions with a `receiver` parameter (e.g. `depositGmx(uint256 amount, address receiver)`), add a helper function which default it with msg.sender. This saves 20 bytes (or 32 bytes, but those 0 will mostly be compressed away) of calldata ~= 12800 gas.

2. For deposit and withdraw functions, allow the user to specify some magic value to deposit/withdraw max. This saves 32 bytes of calldata ~= 20480 gas.
3. On the UI level, use more compressible value (e.g. 1024(0x400) instead of 1000(0x3E8)) for amounts like minGlp
4. Consider integrating with ArbAddressTable for token addresses and other common addresses.

Resources

- https://developer.offchainlabs.com/
- https://l2fees.info/blog/rollup-calldata-compression

drahrealm (Redacted Cartel) commented:

> These are interesting tips 👍 .

Picodes (judge) commented:

> Flagging as best as I believe this was the submission providing the most value to the sponsor.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and

responsibility of users.

Top