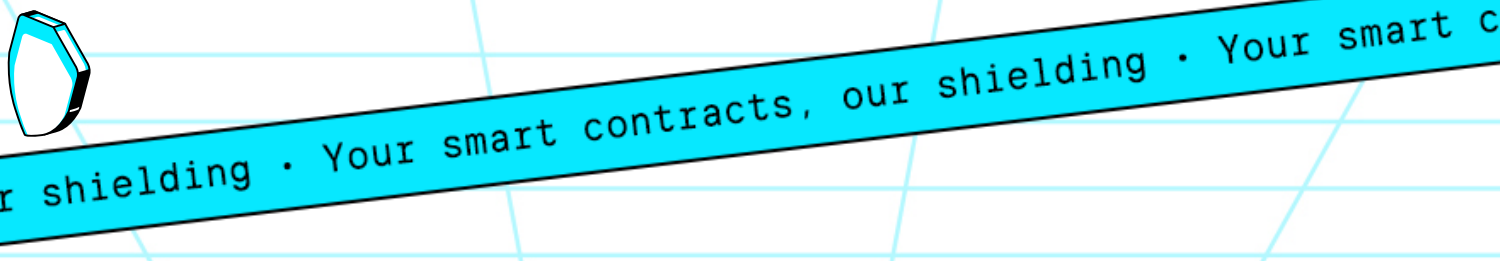
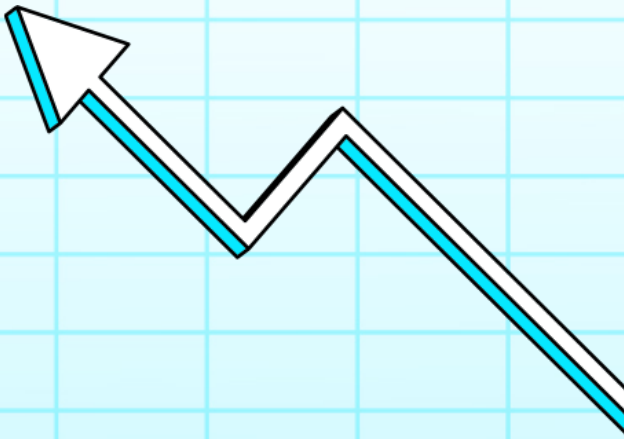


shieldify



Possum Labs

Portals V2

SECURITY REVIEW

Date: 28 March 2024

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach working with the best talents in the space.

Learn more about us at shieldify.org.

2. Disclaimer

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

3. About Possum Labs Portals V2

Possum Labs is a new DeFi primitive that aims to create a vast ecosystem of self-regulating financial products. This is achieved by allowing users to deposit yield-bearing assets and receive their yield up-front. Essentially, depositors can get months' worth of yield immediately after they deposit. Each depositor retains the flexibility to withdraw deposits at any time by paying back part of the upfront yield.

The beating heart of Possum is the Portal smart contract. When a specific Portal is ready to launch, it'll undergo an initial funding phase where funders can deposit `PSM` tokens up to a certain limit. In return, they're given receipt tokens based on a pre-determined reward rate. These receipt tokens, called `bTokens`, can later be redeemed for `PSM` – Possum's native token. As the funding pool grows, depositors will accumulate rights to more deposited `PSM` tokens. The application is expected to launch on Arbitrum Mainnet.

In Portals v2, users will take on the same roles – depositors, funders, and arbitrageurs – but a new architecture is implemented which will benefit each role. The primary upgrades in v2 are the shared liquidity pool and NFT depository receipts.

Take a deep dive into Possum's documentation here.

4. Risk Classification

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

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users

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

4.2 Likelihood

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

5. Security Review Summary

The bug bounty race was conducted in the span of two weeks (March 11th until March 24th) and a total of 30 smart contract security researchers participated. This SPP was the second layer in Possum Labs' planned security methodology for the codebase, which has been audited beforehand.

Overall, the code is written to by very high standards and implements very solid security practices. Shieldify's Private Pool race contributed by identifying several issues of varying severity, around value extraction and potentially higher than intended earniing, among other less severe findings which have been acknowledged by the Possum team.

Shieldify extends their gratitude to Possum Labs' team for their exceptional work communication and responsiveness.

5.1 Protocol Summary

Project Name	**Possum Labs V2**
Repository	Portals
Type of Project	DeFi, Stake yield-bearing assets and receive instant upfront yield
Audit Timeline	14 days
Review Commit Hash	810fd90645cccf6fd80d91f8050eff618f63af81
Fixes Review Commit Hash	65ceff60fd00ff657b882a56e56fd75626f59d61

5.2 Scope

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

File	nSLOC
src/V2MultiAsset/interfaces/IPortalV2MultiAsset.sol	3
src/V2MultiAsset/interfaces/ISingleStaking.sol	3
src/V2MultiAsset/interfaces/IVirtualLP.sol	3
src/V2MultiAsset/interfaces/IWater.sol	3

src/V2MultiAsset/interfaces/IDualStaking.sol	3
src/V2MultiAsset/VirtualLP.sol	363
src/V2MultiAsset/PortalNFT.sol	73
src/V2MultiAsset/MintBurnToken.sol	26
src/V2MultiAsset/PortalV2MultiAsset.sol	434
Total	**908**

6. Findings Summary

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

- **Critical** and **High** issues: **1**
- **Medium** issues: **2**
- **Low** issues: **6**

ID	Title	Severity	Status
[H-01]	Value Can Be Extracted From The Pool By Combining The Burning of **bToken** and the Buying/Selling of Portal Energy	High	Fixed
[M-01]	Investors Could Earn 10x More Than Intended	Medium	Fixed
[M-02]	Unfair **PSM** Token Gain via **PortalEnergy** Price Manipulation When Calling Convert	Medium	Mitigated
[L-01]	Late Withdrawals Incur Vault Losses If Debt Repayment Defaulted	Low	Acknowledged
[L-02]	Cannot Revoke Permit of **MintBurnToken**	Low	Acknowledged
[L-03]	Unnecessary Typecasting of **msg.sender** in Event Emission	Low	Fixed
[L-04]	Attacker Can Grief Users By Reverting Their **withdraw()** Function	Low	Acknowledged
[L-05]	Arbitrum Sequencer Timestamps Can Affect the **maxLockDuration** Update	Low	Acknowledged
[L-06]	The MintBurnToken Leaves Its Burn Function **public**	Low	Acknowledged

7. Findings

[H-01] Value Can Be Extracted From The Pool By Combining The Burning of `bToken` and the Buying/Selling of Portal Energy

Severity

High Risk

Reported By

carrotsmuggler

Description

Value can be extracted from the pool by combining the burning of **bToken** and the buying/selling of portal energy.

Impact

Users can profit from the liquidity in the pool.

Attack Scenario

A user who has provided liquidity to the **virtualLP** in PSM tokens via **contributeFunding** gets minted **bToken**. They can then call **burnBtokens** later when the pool is active to get back the PSM tokens and burn up their **bToken**.

However, the current amount of PSM tokens in the pool contract determines the price at which portal energy can be bought/sold as seen in the **quoteBuyPortalEnergy** function.

File: src/V2MultiAsset/PortalV2MultiAsset.sol#L672-L690

```
function quoteBuyPortalEnergy(
    uint256 _amountInputPSM
) public view activeLP returns (uint256 amountReceived) {
// @dev Calculate the PSM token reserve (input)
    uint256 reserve0 = IERC20(PSM_ADDRESS).balanceOf(VIRTUAL_LP);

// @dev Calculate the reserve of portalEnergy (output)
    uint256 reserve1 = CONSTANT_PRODUCT / reserve0;

// @dev Reduce amount by the LP Protection Hurdle to prevent sandwich
   attacks
    _amountInputPSM =
        (_amountInputPSM * (100 - LP_PROTECTION_HURDLE)) /
        100;

// @dev Calculate the amount of portalEnergy received based on the amount
    of PSM tokens sold
    amountReceived =
        (_amountInputPSM * reserve1) /
        (_amountInputPSM + reserve0);
}
```

So a user can burn their **bToken** to take out PSM tokens from the contract. This leads to a drop in the price of portal energy, as the PSM balance in the contract will drop. So a user can sell portal energy from the pool before, and then buy it immediately after for profit.

Proof of Concept (PoC)

Users can exploit it in the following way.

1. Assume the user has acquired a large number of **bToken**.

2. Assume the pool has 100 PSM tokens, and the CONSTANT_PRODUCT is set to 1e4.
3. The user sells the 120 portal energy tokens for 54 PSM tokens (assuming 0 LP_PROTECTION_HURDLE). Pool now has 46 PSM tokens.
4. User burns up their **bToken** to remove 20 PSM tokens from the pool. Pool now has only 26 PSM tokens.
5. The user buys portal energy using the 54 PSM tokens and gets back 259.6 portal energy

The user has made a profit of 139.6 portal energy.

The protocol tries to mitigate this with an LP_PROTECTION_HURDLE factor. However, this is only a percentage change on the amount of tokens being exchanged. The profitability also depends on the amount of liquidity removed from the pool by burning **bToken**. So if a large amount of **bToken** can be acquired and burnt, the LP_PROTECTION_HURDLE factor becomes irrelevant.

Location of Affected Code

File: src/V2MultiAsset/VirtualLP.sol#L697-L723

```
function burnBtokens(uint256 _amount) external {
// @dev Check that the burn amount is not zero
    if (_amount == 0) {
        revert InvalidAmount();
    }

// @dev Check that the burn amount is not larger than what can be
   redeemed
    uint256 burnable = getBurnableBtokenAmount();
    if (_amount > burnable) {
        revert InvalidAmount();
    }

// @dev Calculate how many PSM the user receives based on the burn amount
    uint256 amountToReceive = getBurnValuePSM(_amount);

// @dev Reduce the funding reward pool by the amount of PSM payable to
   the user
    fundingRewardPool -= amountToReceive;

// @dev Burn the bTokens from the user's balance
    bToken.burnFrom(msg.sender, _amount);

// @dev Transfer the PSM to the user
    IERC20(PSM_ADDRESS).transfer(msg.sender, amountToReceive);

// @dev Event that informs about burn amount and received PSM by the
   caller
    emit RewardsRedeemed(msg.sender, _amount, amountToReceive);
}
```

Recommendation

Limit the amount of **bToken** that can be burnt in one go. A limit for **bToken** burnt which replenishes over time so that users cannot manipulate the pool too much would prevent the exploit combined with the LP_PROTECTION_HURDLE factor.

Team Response

Fixed by subtracting the rewardPool from the PSM balance of the virtual LP when calculating swaps.

[M-01] Investors Could Earn 10x More Than Intended

Severity

Medium Risk

Reported By

Mario Poneder

Description

Initial funders get 10x the amount of **bToken** than invested **PSM** in contributeFunding(...), since FUNDING_MAX_RETURN_PERCENT = 1000. This is intended.

```
function contributeFunding(uint256 _amount) external inactiveLP {
    ...

// @dev Calculate the amount of bTokens to be minted based on the maximum
     return
    uint256 mintableAmount = (_amount * FUNDING_MAX_RETURN_PERCENT) / 100;
// @audit bToken = PSM * 10
    ...

// @dev Transfer the PSM tokens from the user to the contract
    IERC20(PSM_ADDRESS).transferFrom(msg.sender, address(this), _amount);

// @dev Mint bTokens to the user
    bToken.mint(msg.sender, mintableAmount);

    ...
}
```

When burning **bToken** to get **PSM** after the funding period via burnBtokens(...), the getBurnValuePSM(...) method correctly computes the **minValue** to return the initial investment. However, the **maxValue**, which can be reached over time allows a **100x** increase from the initial investment although only a 10x increase is intended by the protocol.

```
function getBurnValuePSM(
    uint256 _amount
) public view activeLP returns (uint256 burnValue) {
// @dev Calculate the minimum burn value
    uint256 minValue = (_amount * 100) / FUNDING_MAX_RETURN_PERCENT;  //
       @audit PSM = bToken / 10

// @dev Calculate the time based burn value
    uint256 accruedValue = ...

// @dev Calculate the maximum and current burn value
    uint256 maxValue = (_amount * FUNDING_MAX_RETURN_PERCENT) / 100; //
       @audit PSM = bToken * 10
    uint256 currentValue = minValue + accruedValue;

    burnValue = (currentValue < maxValue) ? currentValue : maxValue;
}
```

Similarly, the convert(...) method is affected, where the **maxRewards** is given by **10x** the **bToken** supply although it's already 10x the invested **PSM**.

```
function convert(
    address _token,
    address _recipient,
    uint256 _minReceived,
    uint256 _deadline
) external nonReentrant activeLP {
    ...

// @dev initialize helper variables
    uint256 maxRewards = (bToken.totalSupply() *
        FUNDING_MAX_RETURN_PERCENT) / 100;      // @audit bToken supply
           already contains factor 10x
    ...

// @dev Check if rewards must be added, adjust reward pool accordingly
    if (fundingRewardPool + newRewards >= maxRewards) {
        fundingRewardPool = maxRewards;
    } else {
        fundingRewardPool += newRewards;
    }

   ...
}
```

Impact

All in all, investors could earn 10x more than intended via **burnBtokens(...)** and **convert(...)** which could have detrimental impacts on the solvency of the protocol.

Location of Affected Code

src/V2MultiAsset/VirtualLP.sol#L666-L667

src/V2MultiAsset/VirtualLP.sol#L536-L538

Recommendation

Reduce the implicit 100x gain to the intended 10x gain in both instances:

```
diff --git a/src/V2MultiAsset/VirtualLP.sol b/src/V2MultiAsset/VirtualLP.
   sol
index 30c71b8..8531f0a 100644
--- a/src/V2MultiAsset/VirtualLP.sol
+++ b/src/V2MultiAsset/VirtualLP.sol
@@ -534,8 +534,7 @@ contract VirtualLP is ReentrancyGuard {
     }

// @dev initialize helper variables
-    uint256 maxRewards = (bToken.totalSupply() *
-    FUNDING_MAX_RETURN_PERCENT) / 100;
+    uint256 maxRewards = bToken.totalSupply();
     uint256 newRewards = (AMOUNT_TO_CONVERT * FUNDING_REWARD_SHARE) /
         100;

// @dev Check if rewards must be added, adjust reward pool accordingly
@@ -664,7 +663,7 @@ contract VirtualLP is ReentrancyGuard {
     FUNDING_APR) / (100 * SECONDS_PER_YEAR);

// @dev Calculate the maximum and current burn value
-    uint256 maxValue = (_amount * FUNDING_MAX_RETURN_PERCENT) / 100;
+    uint256 maxValue = _amount;
     uint256 currentValue = minValue + accruedValue;

     burnValue = (currentValue < maxValue) ? currentValue : maxValue;
```

Team Response

Fixed as proposed.

[M-02] Unfair PSM Token Gain via PortalEnergy Price Manipulation When Calling Convert

Severity

Medium Risk

Reported By

Elhaj

Description

- A user can manipulate the system to acquire more PSM tokens than they rightfully deserve when calling th**convert** function. They achieve this by purchasing **portalEnergy** tokens before calling **convert**, then selling these **portalEnergy** tokens after calling **convert** for PSM tokens, thus gaining additional PSM tokens unfairly.

Impact

- This exploitation allows the user to extract more PSM tokens, potentially affecting the price of **portalEnergy** tokens, making them cheaper. This manipulation can impact honest stakers within the protocol.

Attack Scenario

1. The **convert()** function is designed to permit anyone to convert a specific amount of PSM tokens for any yield token held by the **virtualLP** contract. This creates an arbitrage opportunity and funds the **virtualLP** contract with additional PSM tokens, which helps maintain the functionality of upfront yield.
2. However, a flaw exists in this design. The converter of PSM can immediately extract this value by:
    - Before calling **convert**, they execute the **buyPortalEnergy** function to purchase as many **portalEnergy** tokens as possible.
    - The user then calls **convert** to convert a specific amount of PSM tokens for a yield token earned by the **virtualLP** contract.
    - Finally, the user sells the amount of **portalEnergy** they purchased for more PSM tokens than they initially bought.

Proof of Concept (PoC)

- here is a PoC in the test file demonstrating how an attacker can exploit this to acquire more PSM tokens than they should:

```
function test_poc() public {
    helper_prepareSystem();
    uint256 portalEnergy;
    (,,,,, portalEnergy,) = portal_USDC.getUpdateAccount(Alice, 0, true);
// buy some portal energy :
    vm.startPrank(Alice);
    psm.approve(address(portal_USDC), 1e55);
    uint256 balPsmBefore = psm.balanceOf(Alice);
    portal_USDC.buyPortalEnergy(Alice, balPsmBefore, 1, block.timestamp);
    uint256 psmSpent = balPsmBefore - psm.balanceOf(Alice);
    vm.stopPrank();
// convert :
    helper_executeConvert();
//sell the same amount of portal energy :
    vm.startPrank(Alice);
    uint256 reserve1 = _TARGET_CONSTANT_USDC / _FUNDING_MIN_AMOUNT;
    uint256 netPSMinput = (balPsmBefore * 99) / 100;
    uint256 result = (netPSMinput * reserve1) / (netPSMinput +
       _FUNDING_MIN_AMOUNT);
    portal_USDC.sellPortalEnergy(Alice, result, 1, block.timestamp);
// extracted value :
    uint256 valueExtracted = psm.balanceOf(Alice) - balPsmBefore;
    console2.log("value spend %e", psmSpent);
    console2.log("value extracted %e", valueExtracted);
}
```

- after running test :

```
Ran 1 test for test/V2MultiAsset/PortalV2MultiAssetTest.t.sol:
   PortalV2MultiAssetTest
[PASS] test_poc() (gas: 1709548)
Logs:
  transfer
  value spend 1e25
  value extracted 4.974748109248976104 9104e22
```

Location of Affected Code

src/V2MultiAsset/PortalV2MultiAsset.sol#L556

src/V2MultiAsset/PortalV2MultiAsset.sol#L608

src/V2MultiAsset/VirtualLP.sol#L502

Team Response

Mitigated, roughly 42k of the 100k from incoming converts would be extracted with the given testing parameters, meaning only 58k arrives in the Portal, but won't be fixed since it's a trade-off decision with the following arguments:

1. Not 100% of the incoming value can be extracted under any circumstance which was the intention of introducing the LP_PROTECTION_HURDLE. Essentially, this is a trading fee that accrues to the LP when someone buys PE. In that sense, it works as intended.

2. The testing values are exemplary and not realistic. For example, in the test setup it is assumed that the LP will be funded with 10M PSM and users (Alice, Bob) have the same amount as well. In reality, the LP will have a minimum funding of 500M PSM which is 5% of the total supply. If we scale the POC to the real numbers and keep the constant K stable (because it was not adjusted to the test values in the first place), reserves and prices of the LP change dramatically and with it also the potential for value extraction. Even if the attacker uses more than 800M PSM to buy PE, the result is still a net negative.

[L-01] Late Withdrawals Incur Vault Losses If Debt Repayment Defaulted

Severity

Low Risk

Reported By

carrotsmuggler

Impact

Early withdrawals will be able to withdraw the full amount, but late withdrawals will eat the entire loss if the WATER vault incurs losses due to lending

Attack Scenario

The WATER vault contract loans out tokens to other vaults to use them and asks for yield in return. This is done via the **lend** and **repayDebt** functions. In the **repayDebt** function, 2 parameters are passed, _debtAmount and _amountPaid. So if the target vaults incur a loss, they can erase the debt by passing a lower _amountPaid than the _debtAmount. This will cause the WATER vault to remove more debt than the amount of USDC tokens they receive. This causes the **totalAssets()** value in the WATER vault to go down.

Due to this, the amount of shares needed to take out some **asset** amount will increase. This means if a user who deposited an **asset** amount in the Portal decides to take out the same **asset** amount, the contract will use more shares than the user deposited. So initial users will be able to take out the full **asset** amount, while final users won't be able to withdraw anything since the contract would have burnt up all its shares and would thus be insolvent.

Proof of Concept (PoC)

This triggers if a final vault incurs a loss and writes off that loss by calling **repayDebt**. Normal ERC4626 vaults can cope with this since they do their accounting in terms of shares, not assets. However since here the accounting is done with the assets, the contract will go insolvent.

Location of Affected Code

File src/V2MultiAsset/VirtualLP.sol#L271-L316

Recommendation

If the WATER vault incurs a loss, users should be given back a proportional amount of tokens.

Team Response

Acknowledged, since it's theoretically possible to occur but it is not worth the added complexity (and gas for everyone) to keep track of shares owned by individual stakers and separate those from the shares that are "profit" of the Portal.

[L-02] Cannot Revoke Permit of MintBurnToken

Severity

Low Risk

Reported By

Mario Poneder

Description

The **MintBurnToken** inherits from **ERC20Permit** which provides users with a gasless way to give other users a token approval by generating a permit signature.
However, this signature is valid until the **deadline** and cannot be revoked beforehand if desired.

Impact

The **MintBurnToken** does not implement a public entry-point for ERC20Permit._useNonce(...) which could be used to invalidate a token approval signature by incrementing the nonce.

Attack Scenario

1. Alice creates a token approval signature for Bob.
2. Bob gets compromised which exposes his wallet and signature to the risk of being used by an adversary.
3. Alice needs to immediately invalidate the signature **before** the deadline to avoid even further losses.

Location of Affected Code

File: src/V2MultiAsset/MintBurnToken.sol#L10

Recommendation

Add a public entry-point of _useNonce(...)}to the **MintBurnToken**:

```
function useNonce() external returns (uint256 current) {
    current = _useNonce(msg.sender);
}
```

Team Response

Acknowledged.

[L-03] Unnecessary Typecasting of msg.sender in Event Emission

Severity

Low Risk

Reported By

Pelz

Description

The function **mintPortalEnergyToken** emits an event using **msg.sender** and typecasts it to an address, which is unnecessary since **msg.sender** already returns an address.

Impact

While this issue does not pose a security risk, it introduces unnecessary complexity to the code and may lead to confusion for developers reviewing the contract.

Location of Affected Code

File: src/V2MultiAsset/PortalV2MultiAsset.sol#L820

```
emit PortalEnergyMinted(address(msg.sender), _recipient, _amount);
```

Recommendation

Remove the unnecessary typecast of **msg.sender** to an address in the **emit** statement. Simply use **msg.sender** directly as it already returns an address. This will improve code readability and maintainability.

Team Response

Fixed.

[L-04] Attacker Can Grief Users By Reverting Their withdraw() Function

Severity

Low Risk

Reported By

djxploit

Description

A highly dedicated attacker can prevent users from withdrawing their stakes, by making their withdrawal call fail. If an attacker calls the stake function continuously, he can block the victim users from withdrawing. Because they may fall in the same block and may have the same timestamp. As the timelock variable in the vault contract is set while calling the deposit function, if they have the same timestamp, the withdrawn call will fail, because of the **require** statement.

The issue happens because of the timelock check in Vaultka contracts.

Impact

Grief victim users from withdrawing their stakes.

Location of Affected Code

File: src/V2MultiAsset/PortalV2MultiAsset.sol#L340

Recommendation

This needs to be fixed in the vault contract.

Team Response

Unfortunately, we can't do anything on our side to prevent this. If Vaultka used >= instead of > in the timelock check of their vault this would not be a problem at all. We'll make them aware of this issue and hope they fix it with their next vault upgrade to improve comparability.

[L-05] Arbitrum Sequencer Timestamps Can Affect the maxLockDuration Update

Severity

Low Risk

Reported By

Plamen Tsanev

Description

The Arbitrum L2 block timestamp and block number are derived following a set of specific rules, that could impact the update of the max lock.

Impact

The Arbitrum timestamp is updated every L2 block (which block is delayed due to the sequencer) and depending on the delay from the parent chain, can have fluctuating values from 24 hours earlier to 1 hour into the future, which could lead to fluctuations in the calculations involving the max lock duration. Such times when the sequencer is down or highly delayed are monitorable and could be used.

Attack Scenario

Set the **maxLockDuration** to a slightly inflated/deflated value than expected, which could be a part of a front-run.

Recommendation

There is no real recommendation that can prevent the sequencer's behavior. Block numbers generally hold their value better, when disregarding the 1 minute delay, so something around them could be thought up.

Team Response

Acknowledged.

[L-06] The MintBurnToken Leaves Its Burn Function public

Severity

Low Risk

Reported By

Plamen Tsanev

Description

The **MintBurnToken** contract is the contract behind the PortalEnergyToken deployed for each portal and could be minted/sold probably for trading purposes or transfer purposes, similarly to the NFT functionality. It however inherits ERC20Burnable and leaves its implementation open.

Impact

The ERC20Burnable contract is inherited in order to make use of the **burnFrom** function when selling energy. However the methods b**burn** and **burnFrom** are left public, so any left-over allowances between users and users themselves could burn tokens.

Location of Affected Code

File src/V2MultiAsset/MintBurnToken.sol

Recommendation

Override the methods with the **onlyOwner** modifier like the **mint** method.

Team Response

There are legit cases where a token owner (such as our treasury) will want to burn **bTokens** without receiving the reward for whatever reason by directly calling the burn function. This won't be possible if implemented as an owner function, hence won't fix.

shieldify

Thank you!