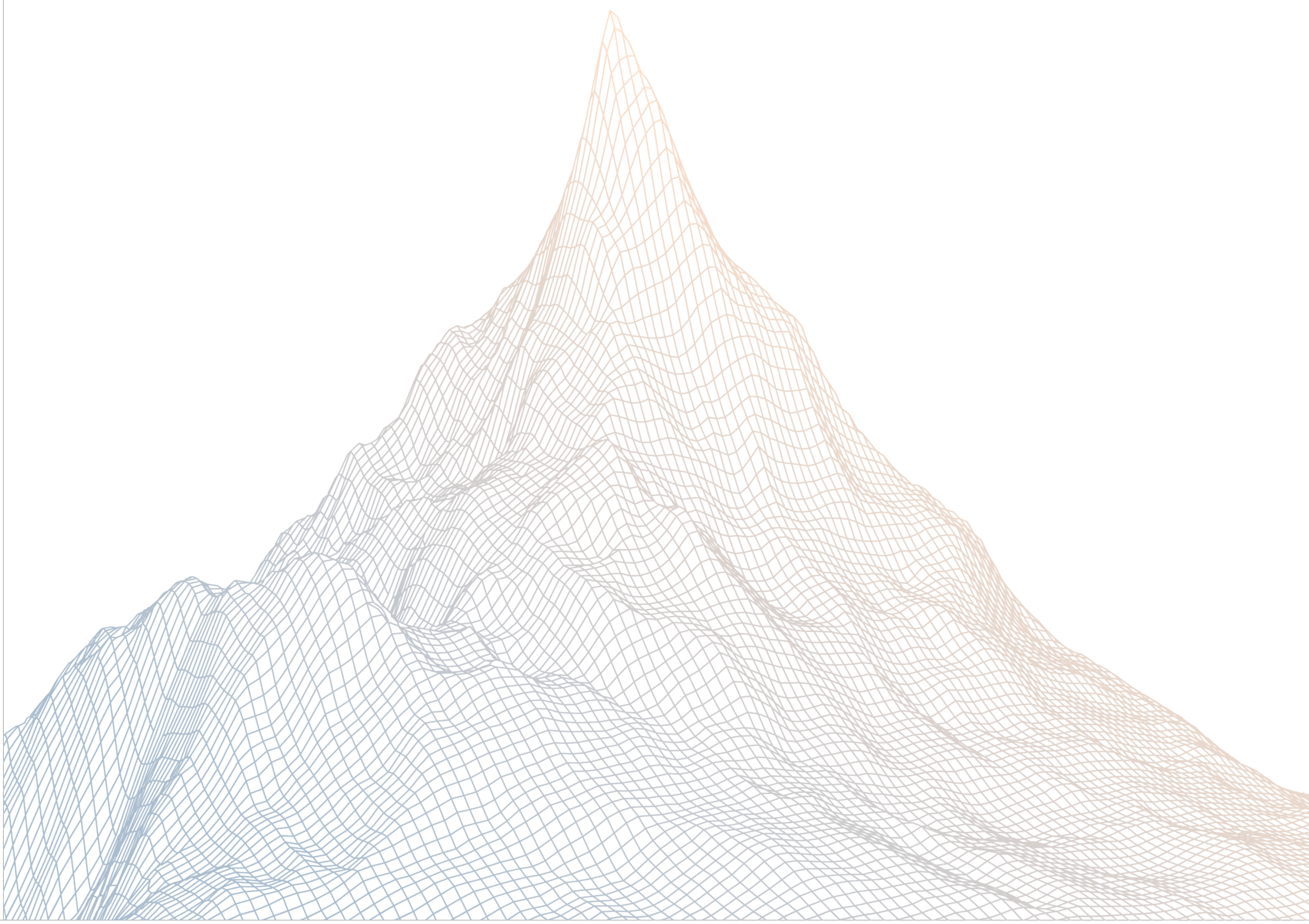
Zenith

VVET Protocol

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: June 24th to July 4th, 2025

AUDITED BY: Chinmay
zzykxx

Contents

1

Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2 Executive Summary

2.1 About VVET Protocol

VVET Protocol is a liquid staking platform for the Virtuals Protocol ecosystem. It lets users stake and earn from staking pools while receiving a liquid version of the non-transferable $veVIRTUAL. The protocol is governed by $VVET holders, who collectively vote on Virtuals Protocol governance using aggregated voting power.

2.2 Scope

The engagement involved a review of the following targets:

Target	vvet
Repository	https://github.com/luiyongsheng/vvet
Commit Hash	`2b5a8d40294483b93dd2ac808cf92d0cf198df9c`

Files	`tokens/*`
	`FarmPool.sol`
	`FarmPoolFactory.sol`
	`FarmPoolUpgradeable.sol`
	`FarmPoolFactoryUpgradeable.sol`
	`XSwap.sol`

2.3 Audit Timeline

June 24, 2025	Audit start
July 4, 2025	Audit end
July 8, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	6
Medium Risk	7
Low Risk	4
Informational	10
Total Issues	**27**

3

Findings Summary

ID	Description	Status
H-1	xERC20.sol is vulnerable to inflation attacks	Resolved
H-2	Each individual FarmPool uses 1 B VVET tokens as maxSupply	Acknowledged
H-3	xERC20 rewards can be stolen as soon as they get transferred to the contract	Resolved
H-4	Missing voting functionality in XSwap wastes its aggregate voting power	Resolved
H-5	New deposits after executeEmergencyWithdrawal() can be stolen by previous depositors	Resolved
H-6	Executing updateEmissionParameters() leads to incorrect emissions calculations	Resolved
M-1	XSwap::_getAvailableLockIds() might skip some available locks	Resolved
M-2	FarmPool.sol and xERC20.sol will never distribute rewards accumulated when the total supply of shares is 0	Resolved
M-3	FarmPoolUpgradeable and xERC20Upgradeable can't be upgraded	Resolved
M-4	The function distributeRemainingRewards() doesn't transfer reward tokens	Resolved
M-5	xERC20 assumes the amount of rewards to be distributed is always exactly MAX_SUPPLY	Resolved
M-6	No consideration for adminUnlocked in XSwap's lock withdrawal flows	Resolved
M-7	Use of cached _maxWeeks in XSwap can lead to many problems	Resolved
L-1	xERC20::stake() rounds up the amount of shares minted	Resolved

ID	Description	Status
L-2	Emergency withdrawal flows lack check for MAX_LOCKS_PER_REQUEST	Resolved
L-3	Use of unbounded array in _getAvailableLockIDs()	Resolved
L-4	Loss of contract ownership on xVirtual and bxVirtual tokens	Resolved
I-1	MIN_STAKE_AMOUNT can be easily bypassed in stxVirtual	Resolved
I-2	Rewards distribution is 4x faster on Base than it is on Ethereum	Resolved
I-3	VVET token should not be burnable	Resolved
I-4	XSwap::_claimWithdrawal() implements an unnecessary gas check	Resolved
I-5	XSwap::executeEmergencyWithdrawal() always reverts after changing state variables	Resolved
I-6	Modifier whenNotPaused is not applied consistently to withdrawals in FarmPoolUpgradeable	Resolved
I-7	_rewardPerToken() performs multiplication before calling FullMath.mulDiv()	Resolved
I-8	Unused functions and variables in xERC20	Resolved
I-9	XSwap should be upgradeable	Acknowledged
I-10	renounceOwnership should be blocked across the codebase	Resolved

4 Findings

4.1 High Risk

A total of 6 high risk findings were identified.

[H-1] `xERC20.sol` is vulnerable to inflation attacks

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- xERC20.sol

Description:

The xERC20.sol contract calculates the amount of shares to mint on deposit based on the current total supply of shares, the balance of stake token in the contract and the amount of stake token deposited:

```
xERC20Amount = FullMath.mulDiv(
    stakeTokenAmount,
    PRECISION,
    getPricePerFullShare()
);
```

```
function getPricePerFullShare() public view returns (uint256) {
//...SNIP...
    uint256 stakeTokenBalance = stakeToken().balanceOf(address(this));

    uint256 currentEmissions = totalEmitted();
    uint256 virtualBalance = stakeTokenBalance + tokens
        currentEmissions -
        totalEmittedRewards;

    return FullMath.mulDiv(virtualBalance, PRECISION, totalShares);
}
```

This is vulnerable to inflation attacks as the following is possible:

1. The xERC20.sol has just been deployed, emissions started, and the contract holds no funds.
2. Alice calls xERC20::stake(), depositing `1e18` units of stake token. This transaction is now in the mempool.
3. Eve notices Alice's transaction and calls xERC20::stake(), depositing `1` single unit of stake token. This mints `1` share to Eve.
4. Eve transfers `1e18` units of stake token directly into the xERC20.sol contract. This mints no shares. The total supply of shares is now `1` and the balance of stake tokens in the contract is `1e18 + 1`.
5. Alice's transaction is executed. The price per share is currently `(1e18 + 1)/1` and the amount deposited by Alice is `1e18`. This implies `xERC20Amount` will round down to `0` and mint Alice `0` shares.
6. Eve still has the only share minted, she can now execute xERC20::withdraw() to steal Alice's deposit and withdraw all funds from the contract.

Recommendations:

Use a system of virtual shares and decimal offsets as it's done in Openzeppelin ERC4626 vault contract

Virtuals: Resolved with @80ebbf7...

Zenith: Verified.

[H-2] Each individual FarmPool uses 1 B VVET tokens as maxSupply

SEVERITY: High	IMPACT: High
STATUS: Acknowledged	LIKELIHOOD: High

Target

- FarmPool.sol

Description:

A FarmPool instance uses 1 B as the maxSupply of VVET reward tokens for the emission curve, which means the emission rate schedules will work as if all 1 Billion VVET tokens are to be distributed in this individual FarmPool.

This is problematic because each FarmPool staking instance has to use VVET as its reward token, and each one will distribute 1 B reward tokens in its lifetime.

While as per docs, the intended entire totalSupply of VVET should be 1 Billion. This means if there are more than one FarmPool instances, the emissions will not work.

Recommendations:

The maxSupply used for emission calculations in a single FarmPool instance should be a fraction of the entire VVET totalSupply, depending on how many FarmPools are planned, or the total amount of VVET reward tokens should be planned to be much greater. Also consider FarmPoolUpgradeable instances and their planned maxSupply values.

Virtuals: Acknowledged. We will only have 1 active farm pool.

[H-3] `xERC20` rewards can be stolen as soon as they get transferred to the contract

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- xERC20

Description:

The xERC20 contract allows staking tokens to get rewards. The rewards are given out in the same token that's being staked. To achieve this the contract implements a vault-like system that mints shares to stakers whose value is supposed to increase when rewards are transferred directly to the contract.

This is implemented via the getPricePerFullShare() function:

```
function getPricePerFullShare() public view returns (uint256) {
    uint256 totalShares = totalSupply;
    if (totalShares == 0) {
        return PRECISION;
    }

    // Base staked token balance
    uint256 stakeTokenBalance = stakeToken().balanceOf(address(this));

    // Add emitted rewards to the virtual balance
    uint256 currentEmissions = totalEmitted();
    uint256 virtualBalance = stakeTokenBalance +
        currentEmissions -
        totalEmittedRewards;

    return FullMath.mulDiv(virtualBalance, PRECISION, totalShares);
}
```

This implementation allows to steal rewards because it uses the stake token balance of the contract to calculate the price per share.

Here's an example:

1. Admins send a transaction to transfer reward tokens directly to xERC20.
2. Eve monitors the mempool and sees the transaction that sends rewards tokens directly to xERC20.
3. Eve front-runs the transfer by depositing a big amount of stake tokens in order to receive shares.
4. The transaction that sends rewards to xERC20 is executed. Because getPricePerFullShare() prices shares based on the balance of stake tokens in the contract the price per share increases instantly.
5. Eve shares are now worth more and she can withdraw them for a profit, stealing rewards from legitimate users.

Recommendations:

The getPricePerFullShare() function should calculate the value per share by subtracting the amount of rewards that have **not** been distributed yet from the balance of the stake token in the contract.

By doing this the price per share results in the amount of tokens staked plus the amount of rewards that have been distributed already.

Virtuals: Resolved with @80ebbf7...

Zenith: Verified.

[H-4] Missing voting functionality in `XSwap` wastes its aggregate voting power

`SEVERITY:` High	`IMPACT:` High
`STATUS:` Resolved	`LIKELIHOOD:` High

Target

- XSwap.sol

Description:

When users deposit their `VIRTUAL` tokens into `XSwap`, they are giving up their individual voting weights (in the virtuals ecosystem) in return for a liquid xVirtual token. All these `VIRTUAL` locks on `veVirtual` contract are done by `XSwap` on behalf of every user. This leaves the `XSwap` contract owning all the locks.

As per the protocol team, combined balance of all these locks of `XSwap` contract will be used to exercise the associated aggregate voting power, which the protocol intends to use.

But the problem is the Xswap contract does not have voting functionality.

We can look at how the voting process in `VirtualsProtocolDaoV2.sol` uses Openzeppelin's standard `castVote()` interface for allowing lock holders to cast votes.

As a result, all this aggregate voting power of the `XSwap` contract will be wasted as there is no way for the admin to use `XSwap` to interact with VirtualDAO's voting systems.

Recommendations:

Add a guarded `castVote()` / `castVoteWithReason()` function to `XSwap` contract, so that the total voting power can be used by the protocol admin. Keep in mind that there might be multiple places in Virtuals ecosystem where this voting power can be used, add all necessary functions as per requirement.

Virtuals: Resolved with @83e54bd...

Zenith: Verified.

[H-5] New deposits after `executeEmergencyWithdrawal()` can be stolen by previous depositors

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- XSwap.sol

Description:

When an emergency withdrawal gets executed, all the active positions from `XSwap` (as per `_getAvailablelockIds()`) get sweeped out in a single withdrawal request, and none of the existing positions are active after that.

This flow does not burn associated `xVirtual` tokens unlike `requestWithdrawal()`, because those are held by the users.

This allows stealing of user deposits in the following situation :

- Admin calls `executeEmergencyWithdrawal()`, all active positions are marked with `autoRenew == false`, such that calling `_getAvailableLockIds` would return zero locks after this
- A new user comes and deposits for a new lock, now this position will be marked as `autoRenew == true`, which means it will be considered active going forward
- All `xVirtual` balances of previous positions are still with the users, as they were not burned in emergency withdrawal flows
- Attacker (who has xVirtual balance from previous deposit) sees this new deposit, immediately calls `requestWithdrawal()`
- The `_getAvailableLockIds()` logic now considers this new lock as withdrawable for the attacker and the withdrawal request gets placed successfully

The actual depositor also has the `xVirtual` balance corresponding to his deposit, but now he cannot withdraw that position as it has been sweeped by the attacker. This leads to loss of VIRTUAL tokens for an honest user.

This is possible because `deposit()` and `requestWithdrawal()` are still possible after emergency withdrawal has been executed.

Recommendations:

If `executeEmergencyWithdrawal()` is only expected to be called once, after it is called => `deposit()` and `requestWithdrawal()` should be blocked.

Virtuals: Resolved with @0d26ee...

Zenith: Verified.

[H-6] Executing `updateEmissionParameters()` leads to incorrect emissions calculations

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- FarmPoolUpgradeable
- xERC20

Description:

The upgradable contracts FarmPoolUpgradeable and xERC20 both implement a updateEmissionParameters() function that allows to change the `decayConstant` and `maxSupply` state variables.

These parameters are used to calculate emissions and the protocol doesn't cache historical emissions, which implies:

1. Both xERC20Upgradeable::totalEmitted() and FarmPoolUpgradeable::totalEmitted() will return a wrong amount of emitted tokens.
2. Because xERC20Upgradeable::totalEmitted() is used in xERC20Upgradeable::getPricePerFullShare(), the price per share will be incorrect.
3. The function FarmPoolUpgradeable::_totalEmittedAtBlock() will return the wrong amount of tokens.

Recommendations:

The contracts should calculate emissions correctly in situations where either `decayConstant` or `maxSupply` are updated:

1. The contracts should update `rewardPerTokenStored` and `lastUpdateBlock` variables before updating `decayConstant` and/or `maxSupply`
2. The contract should cache current emissions values and current block and use the cached parameters when calculating total emissions or emissions at a specific block.

Keep in mind that the function FarmPoolUpgradeable::_rewardPerToken() uses a subtraction between FarmPoolUpgradeable::totalEmitted() and

FarmPoolUpgradeable::_totalEmittedAtBlock() to calculate the reward per token, this should work even in the current contracts because it's a delta but might not work correctly if other functions are updated.

Virtuals: Resolved with @268437

Zenith: Verified.

4.2 Medium Risk

A total of 7 medium risk findings were identified.

[M-1] `XSwap::_getAvailableLockIds()` might skip some available locks

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- XSwap.sol

Description:

The function XSwap::_getAvailableLockIds() is supposed to retrieve all locks currently held by `XSwap.sol` that have the `autoRenew` parameter set to `true`.

This is done here via the following code:

```
uint256 activeAutoRenewCount = 0;
for (uint256 i; i < totalPositions; ) {
    if (
        allPositions[i].autoRenew &&
        allPositions[i].end > block.timestamp
    ) {
        unchecked {
            ++activeAutoRenewCount;
        }
    }
    unchecked {
        ++i;
    }
}

uint256[] memory availableLocks = new uint256[](activeAutoRenewCount);
uint256 index = 0;

for (uint256 i; i < totalPositions; ) {
```

```
        if (
            allPositions[i].autoRenew &&
            allPositions[i].end > block.timestamp
        ) {
            availableLocks[index] = allPositions[i].id;
            unchecked {
                ++index;
            }
        }
        unchecked {
            ++i;
        }
    }
```

This considers a lock available if:

1. The `autoRenew` parameter is set to `true`
2. The lock `end` parameter is in the future

There are a couple of issues with this:

1. If a lock `autoRenew` is set to `true` the `end` parameter **should** always be in a future date (ie. `maxWeeks` from now)
2. The function used to retrieve the locks, veVirtual.getPositions(), returns the lock with a **stale** `end` parameter (ie. if `autoRenew` is `true` the parameter `end` is not adjusted to be `maxWeeks` from now). This implies an edge case where a lock has `autoRenew` set to `true` but `end` is in the past, if this happens the lock won't be considered as available and users won't be able to retrieve all the funds in the contract as the protocol assumes there are no available locks.

Recommendations:

In XSwap::_getAvailableLockIds() consider a lock available if `autoRenew` is `true` and ignore the `end` parameter.

Virtuals: Resolved with @d4cdf8...

Zenith: Verified.

[M-2] `FarmPool.sol` and `xERC20.sol` will never distribute rewards accumulated when the total supply of shares is 0

SEVERITY: Medium	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- xERC20.sol
- FarmPool.sol

Description:

Both xERC20.sol and FarmPool.sol distribute rewards based on the amount of blocks finalized since emissions started.

The rewards are accounted for and distributed even if the total supply of shares is `0`, which implies the contracts will hold reward tokens that will never be claimed because nobody owns shares:

1. FarmPool.sol is deployed, emissions started and reward tokens added to the contract.
2. Nobody deposits and 1000 blocks passes.
3. The contract FarmPool::totalEmitted() will return `48022781188186030622289l`, which nobody will be able to claim.

Recommendations:

Virtuals: Resolved with @120779..., @f1520e and the owner will mint `1` share to themselves in order to prevent stuck rewards.

Zenith: Verified.

[M-3] `FarmPoolUpgradeable` and `xERC20Upgradeable` can't be upgraded

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- FarmPoolUpgradeable.sol
- xERC20Upgradeable.sol
- FarmPoolFactoryUpgradeable.sol

Description:

The FarmPoolUpgradeable.sol and xERC20Upgradeable.sol contracts both:

- Are deployed behind a minimal EIP1167-like proxy
- Inherits UUPSUpgradeable to render the contracts upgradable

This won't work, both FarmPoolUpgradeable.sol and xERC20Upgradeable.sol won't be upgradable because the UUPSUpgradeable upgradability pattern is meant to be used on implementations sitting behind a ERC1967 proxy.

The UUPSUpgradeable requires the implementation address to be stored in storage slots but EIP1167-like proxies stores the implementation address in the bytecode which will result in the upgrade call reverting on this line.

Recommendations:

- In FarmPoolFactoryUpgradeable deploy FarmPoolUpgradeable.sol and xERC20Upgradeable.sol behind ERC1967 proxy.
- Update FarmPoolUpgradeable.sol and xERC20Upgradeable.sol to remove usage of `CloneERC20` and `Clone`.

Virtuals: Resolved with @e1cdaf..., @f1520e... and @5f2ab6...

Zenith: Verified.

[M-4] The function `distributeRemainingRewards()` doesn't transfer reward tokens

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- FarmPoolUpgradeable.sol

Description:

FarmPoolUpgradeable::distributeRemainingRewards() is an emergency function that's supposed to distribute rewards when the FarmPoolUpgradeable.sol contract is paused but the function doesn't actually transfer tokens.

Recommendations:

Adjust FarmPoolUpgradeable::distributeRemainingRewards() to transfer reward tokens.

The naive fix is to loop over all the users to transfer reward tokens but this won't work because the function can run out of gas if there are too many users. Alternative fixes:

- Adjust FarmPoolUpgradeable::distributeRemainingRewards() to take as input an array of users that will receive rewards
- Implement a function that users can call when the contract is paused to receive rewards, this way users are responsible for the gas cost of transferring rewards

Virtuals: Resolved with @be89b5... and @2dfda4ddb...

Zenith: Verified.

[M-5] `xERC20` assumes the amount of rewards to be distributed is always exactly `MAX_SUPPLY`

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- xERC20

Description:

The xERC20 contract assumes the amount of rewards to be distributed is `MAX_SUPPLY`.

If the amount of rewards to be distributed is not exactly `MAX_SUPPLY`, the contract will distribute the wrong amount of rewards which can be either more or less than intended. Because `MAX_SUPPLY` is expressed in 18 decimals this also assumes the stake token has exactly 18 decimals.

Recommendations:

Manually set `MAX_SUPPLY` on the initialize() function based on the amount of rewards that will be distributed.

Virtuals: Resolved with @80ebbf7...

Zenith: Verified.

[M-6] No consideration for `adminUnlocked` in `XSwap`'s lock withdrawal flows

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- XSwap.sol

Description:

The `veVirtual` contract has a mechanism to make all locks instantly withdrawable, if the admin of `veVirtual` sets `adminUnlocked` to true. This mechanism is most probably present to support instant withdrawals of active locks in case of an emergency.

But the problem is this is not supported for `XSwap` locks, as both regular withdrawals and emergency withdrawals in `XSwap` have no consideration for current `adminUnlocked` status when users execute withdrawal request/ XSwap admin executes emergency withdrawal request.

In both these cases, when placing the request, the `claimableTime` for that withdrawal request is by default set to : (block.timestamp + maxWeeks). And later in `_claimWithdrawal()`, the timestamp is required to be past this `claimableTime`.

This means even if the `veVirtual` admin sets `adminUnlocked` to true in case of emergency, and all locks are instantly withdrawable, the `claimWithdrawal()` function in XSwap will not allow anyone to execute the withdrawal instantly.

Suppose :

- User1 opens a new deposit lock via XSwap
- `veVirtual` admin sets `adminUnlocked` to true, allowing instant withdrawal for everyone
- Seeing this, User1 places a withdrawal request for his lock, his claimable time will be (block.timestamp + maxWeeks)
- He can not call `claimWithdrawal()` until this `claimableTime` passes
- The lock (VIRTUAL tokens) will remain in the `veVirtual` contract

This might pose a risk to the VIRTUAL tokens deposited in `veVirtual` contract as XSwap users/ admin are unable to act on the emergency.

Recommendations:

In `_claimWithdrawal()` flow, check if `veVirtual.adminUnlocked` status is set to true at that time, if true bypass the `claimableTime` check and allow it to call `veVirtual.withdraw()`. Also bypass Emergency Delay in executeEmergencyWithdrawal() if adminUnlocked == true, allowing to execute instantly.

Virtuals: Resolved with @e07263... and @f0b5fcc...

Zenith: Verified.

[M-7] Use of cached `_maxWeeks` in XSwap can lead to many problems

`SEVERITY`: Medium	`IMPACT`: Medium
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- XSwap.sol

Description:

When `XSwap` contract gets deployed, in its constructor, the current value of maxWeeks is fetched from `veVirtual` contract and is cached into a variable `_maxWeeks`. But it is possible that the `maxWeeks` value changes on `veVirtual` end because there is a `setMaxWeeks()` function there.

This can give rise to different set of problems depending on if the `maxWeeks` value is decreased/ increased on `veVirtual` side.

If `maxWeeks` is decreased from lets say 104 weeks (initial value, also cached in XSwap contract) to anything lesser :

- XSwap.deposit() will be bricked because it will pass 104 weeks as the numWeeks parameter (the cached fixed value) => leading to a revert in `veVirtual.stake()` here

Recommendations:

Add a function to `XSwap` contract such that admin can keep the `_maxWeeks` value in sync with the `veVirtual` contract.

Virtuals: Resolved with @3e4c96...

Zenith: Verified.

4.3 Low Risk

A total of 4 low risk findings were identified.

[L-1] `xERC20::stake()` rounds up the amount of shares minted

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- xERC20.sol

Description:

The function xERC20::stake() uses the value returned by xERC20::getPricePerFullShare() as a denominator when calculating the amount of shares to mint:

```
xERC20Amount = FullMath.mulDiv(
    stakeTokenAmount,
    PRECISION,
    getPricePerFullShare()
);
```

Because xERC20::getPricePerFullShare() always rounds down and it's used as a divisor the resulting amount of shares will be rounded up minting more shares than intended and favoring the user instead of the protocol.

Recommendations:

1. Adjust xERC20::getPricePerFullShare() to take an input parameter that specifies the rounding direction of the returned value.
2. Adjust xERC20::stake() to use the version of xERC20::getPricePerFullShare() that rounds up.

Virtuals: Resolved with @80ebbf7...

Zenith: Verified.

[L-2] Emergency withdrawal flows lack check for `MAX_LOCKS_PER_REQUEST`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- XSwap.sol

Description:

When an emergency withdrawal gets executed, all the active positions from XSwap (as per `_getAvailablelockIds()`) get sweeped out in a single withdrawal request.

There is no consideration for `MAX_LOCKS_PER_REQUEST`, unlike `requestWithdrawal()`. This means an unbounded list of locks will have to be claimed in a single transaction later when `claimEmergencyWithdrawal()` gets called by the admin.

This can lead to unbounded gas costs and the transaction might fail.

Recommendations:

There should be a limit on how many locks are included in a single emergency withdrawal request, if needed the request should be split up according to `MAX_LOCKS_PER_REQUEST`.

Virtuals: Resolved with @0d26ee...

Zenith: Verified.

[L-3] Use of unbounded array in `_getAvailableLockIDs()`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- XSwap.sol

Description:

When a user calls `requestWithdrawal()` to get his `VIRTUAL` tokens back, active positions are fetched using `_getAvailableLockIds()`.

But the code in `_getAvailableLockIds()` can get gas-extensive and make the user spend a lot of gas, as it first copies all the positions `XSwap` has, then loops through these to check which ones are active, and then copies all the active ones again to a memory array.

All these operations are done within memory arrays and all of these lists are unbounded, and we know that gas consumption in EVM memory can become exponential.

This can lead to a lot of gas cost for user for simply withdrawing their position.

Recommendations:

Consider refactoring the related code.

Virtuals: Resolved with @0d26ee..., @5cc5a5... and @c6ebb7...

Zenith: Verified.

[L-4] Loss of contract ownership on xVirtual and bxVirtual tokens

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- xVirtual.sol
- bxVirtual.sol

Description:

`xVirtual` and `bxVirtual` tokens are deployed from the constructor of `XSwap` contract. Both these tokens mark the deployer as the owner of these contracts.

Note that this owner address has the ability to set a new swapX address using `setSwapX()` function, as well as the usual transferOwnership mechanism. But this does not work correctly because there is no way to call `xVirtual.transferOwnership()` or `xVirtual.setSwapX()` except out of the constructor.

Because of these missing functions, ownership of these two tokens will be locked forever on deployment and can't be changed. `xVirtual` and `bxVirtual` have a consideration for `oldSwapX` which means it should be modifiable later.

Recommendations:

Add relevant functions to `XSwap` contract using which `XSwap` contract owner can route these calls to `xVirtual` and `bxVirtual` contracts.

Virtuals: Resolved with @eb88841...

Zenith: Verified.

4.4 Informational

A total of 10 informational findings were identified.

[I-1] `MIN_STAKE_AMOUNT` can be easily bypassed in `stxVirtual`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- stxVirtual.sol

Description:

`stxVirtual` contract is used to stake `xVirtual` balances of users. `stxVirtual.stake()` function only allows staking an amount >= `MIN_STAKE_AMOUNT`.

The reason mentioned in comments is "Minimum stake amount to prevent dust attacks", but this can be easily bypassed by first staking `MIN_STAKE_AMOUNT` then unstaking everything except desired dust.

This happens because there are no related checks in the `unstake() function`, so a user can reach a dust position this way.

Recommendations:

Add `MIN_STAKE_AMOUNT` check to `unstake()` function.

Virtuals: Resolved with @89a5e6....

Zenith: Verified.

[I-2] Rewards distribution is 4x faster on Base than it is on Ethereum

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- FarmPool.sol
- xERC20.sol

Description:

Both the FarmPool.sol and xERC20.sol distribute rewards per-block.

Blocks are processed in ~3 seconds on Base and ~12 seconds on Ethereum, implying rewards will be distributed 4x faster on Base than on Ethereum.

Recommendations:

Adjust the rewards distribution formula based on the block production time of the chain on which the contracts are deployed. Adjusting `DECAY_CONSTANT` should work.

Virtuals: Resolved with @68ae09...

Zenith: Verified.

[I-3] VVET token should not be burnable

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- VvetToken.sol

Description:

VVET token rewards will be sent to the relevant FarmPool instances, following which emissions are started and funds drip slowly to the stakers.

The VVET token has a burn function, which can create problems if the admin mistakenly burns VVET tokens from a FarmPool. The admin even has the power to burn VVET rewards from any user address.

Recommendations:

Remove the burn functionality.

Virtuals: Resolved with @b21f7dd...

Zenith: Verified.

[I-4] `XSwap::_claimWithdrawal()` implements an unnecessary gas check

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- XSwap.sol

Description:

The function XSwap::_claimWithdrawal() includes a gas check and reverts if the remaining gas is not enough:

```
for (uint256 i; i < lockIdsLength; ) {
    // Gas check with documented constants
    if (
        i != 0 &&
        i % GAS_CHECK_INTERVAL == 0 &&
        gasleft() < MIN_GAS_REMAINING
    ) {
        revert InsufficientGas();
    }
    //...SNIP...
}
```

This check is not useful because, if the gas is not enough, the function will revert. The other reason is - The XSwap::requestWithdrawal() function, which must be called before XSwap::_claimWithdrawal(), already implements a "gas check" by only allowing a maximum of `MAX_LOCKS_PER_REQUEST` to be withdrawn per-request

Recommendations:

Remove the gas check in XSwap::_claimWithdrawal().

Virtuals: Resolved with @5cc5a5...

Zenith: Verified.

[I-5] `XSwap::executeEmergencyWithdrawal()` always reverts after changing state variables

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- XSwap.sol

Description:

The XSwap::executeEmergencyWithdrawal() function always reverts if the XSwap.sol doesn't hold any lock whose `autoRenew` parameter is set to `true`:

```
if (allLockIds.length == 0) {
    uint256 totalPositions = veVirtual.numPositions(address(this));
    if (totalPositions == 0) {
        revert ZeroAmount(); // Truly no locks to withdraw
    }

    // There are positions but none are auto-renewing
    // This means all locks are already in withdrawal requests
    // Reset emergency state and revert
    emergencyWithdrawalPending = false;
    emergencyWithdrawalTime = 0;
    revert ZeroAmount();
}
```

Before reverting it resets the `emergencyWithdrawalPending` and `emergencyWithdrawalTime` variables but this won't have any effect as all state variables changes will be reverted on the next line.

Recommendations:

Return instead of reverting if the state variable changes need to be persisted.

Virtuals: Resolved with @1ea14b... and @f0b5fcc...

Zenith: Verified.

[I-6] Modifier `whenNotPaused` is not applied consistently to withdrawals in `FarmPoolUpgradeable`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- FarmPoolUpgradeable.sol

Description:

`FarmPoolUpgradeable` has a bunch of extra features that allow the pool admin to adjust things. One of these features is the ability of the admin to pause interactions within the pool, including staking and withdrawals.

However, this guard can be bypassed for withdrawals because the `exit()` function does not have `whenNotPaused` modifier.

Recommendations:

Since `withdraw()` has the `whenNotPaused` modifier, `exit()` function should have the same.

Virtuals: Resolved with @f8e48e...

Zenith: Verified.

[I-7] `_rewardPerToken()` performs multiplication before calling `FullMath.mulDiv()`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- FarmPool.sol
- FarmPoolUpgradeable.sol

Description:

The functions FarmPoolUpgradeable::_rewardPerToken() and FarmPool::_rewardPerToken() both use the `FullMath` library to perform multiplications in order to avoid reverts if the calculations overflow on an intermediate value:

```
return rewardPerTokenStored + FullMath.mulDiv(newEmissions * PRECISION, 1,
    totalSupply_);
```

The functions multiply the `newEmissions` and `PRECISION` values outside of the `FullMath.mulDiv()` function, rendering the use of the library useless.

Recommendations:

Use the library as intended:

```
return rewardPerTokenStored +
            FullMath.mulDiv(newEmissions, PRECISION, totalSupply_);
```

Virtuals: Resolved with @6164ee...

Zenith: Verified.

[I-8] Unused functions and variables in `xERC20`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- xERC20

Description:

The xERC20 contract:

- Implements the internal function _totalEmittedAtBlock() which is never used.
- Has a lastUpdateBlock state variable which is never used for anything meaningful.

Recommendations:

Remove the _totalEmittedAtBlock() function and the lastUpdateBlock state variable.

Virtuals: Resolved with @80ebbf7...

Zenith: Verified.

[I-9] XSwap should be upgradeable

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Acknowledged	`LIKELIHOOD:` Low

Target

- XSwap.sol

Description:

The `XSwap` code interacts with veVirtual's stake and withdraw logic. We can see that the `veVirtual` contract is upgradeable and thus its logic may change.

In case the logic gets upgraded, it is possible that the XSwap's integration code may not work properly as intended after that.

Recommendations:

Consider making XSwap upgradeable.

Virtuals: Acknowledged. veVirtual is unlikely to change

[I-10] renounceOwnership should be blocked across the codebase

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- XSwap.sol

Description:

`XSwap` contract inherits from `openzeppelin's Ownable` library. This has two potential scenarios that may lead to loss of contract ownership for the admins :

- `renounceOwnership()` is not overriden and blocked
- It uses a single-step ownership transfer functionality, if an inaccessible address is made the new owner, emergency withdrawals will become unusable.

Management of contract ownership can be improved. The same issue exists in FarmPoolUpgradeable.sol as well.

Recommendations:

Consider applying the modifications mentioned above.

Virtuals: Resolved with @d4cdf8... and @ff8e48...

Zenith: Verified.