PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Lynex (Scope Extension)

24 September 2024

paladinsec.co

info@paladinsec.co

Table of Contents

Disclaimer

1 Overview

This report has been prepared for Lynex on the Ethereum network. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

1.1 Summary

Project Name	Lynex
URL	https://www.lynex.fi/
Platform	Ethereum
Language	Solidity
Preliminary Contracts	`https://github.com/Lynexfi/lynex-contracts/commit/839328c7243f386bab874158199c19639a9d9d6e`
Resolution 1	`https://github.com/Lynexfi/lynex-contracts/tree/d127eb5e8fc5ade3b7eb4865605df337bb860ae3`

1.2 Contracts Assessed

Name	Contract	Live Code Match
VotingEscrowV2Upgradeable		
ERC5725Upgradeable		
Checkpoints		
EscrowDelegateCheckpoints		
EscrowDelegateStorage		

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
● Governance	0	-	-	-
● High	1	1	-	-
● Medium	3	2	-	1
● Low	6	3	-	3
● Informational	14	6	2	6
Total	**24**	**12**	**2**	**10**

Classification of Issues

Severity	Description
● Governance	Issues under this category are where the governance or owners of the protocol have certain privileges that users need to be aware of, some of which can result in the loss of user funds if the governance's private keys are lost or if they turn malicious, for example.
● High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
● Medium	Bugs or issues that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
● Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
● Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 VotingEscrowV2Upgradeable

ID	Severity	Summary	Status
01	HIGH	Malicious users can DOS token transfers	✓ RESOLVED
02	MEDIUM	Lack of minimum amount for a lock leads to users not getting any voting power	ACKNOWLEDGED
03	MEDIUM	Not all fields are set on a merged lock	✓ RESOLVED
04	LOW	The `MAX_TIME` lock restriction is obsolete	ACKNOWLEDGED
05	LOW	The logic behind the permanent lock is ambiguous	✓ RESOLVED
06	LOW	Expired locks can be reactivated	✓ RESOLVED
07	LOW	`claimablePayout` will fail with arithmetic error	✓ RESOLVED
08	INFO	The upgradeable version of `ReentrancyGuard` should be used	✓ RESOLVED
09	INFO	Lack of validation in the constructor	PARTIAL
10	INFO	Splits with zero or small weights should be prohibited	ACKNOWLEDGED
11	INFO	Inefficient `LockMismatch` check when merging	ACKNOWLEDGED
12	INFO	Tokens resulting from a split can not have a designated delegate	ACKNOWLEDGED
13	INFO	`getPastTotalSupply` can look for future total supply	ACKNOWLEDGED
14	INFO	No setter function for `artProxy`	ACKNOWLEDGED
15	INFO	Event not emitted properly	✓ RESOLVED
16	INFO	Typographical issues	✓ RESOLVED
17	INFO	Gas optimizations	✓ RESOLVED

1.3.2 ERC5725Upgradeable

ID	Severity	Summary	Status
18	MEDIUM	Contract will fail to initialize after deployment	✓ RESOLVED
19	LOW	Insufficient validation	ACKNOWLEDGED
20	INFO	Typographical issues and gas optimizations	✓ RESOLVED

1.3.3 Checkpoints

ID	Severity	Summary	Status
21	LOW	Array size can grow too big	ACKNOWLEDGED
22	INFO	Gas optimizations	ACKNOWLEDGED

1.3.4 EscrowDelegateCheckpoints

ID	Severity	Summary	Status
23	INFO	`checkpoint()` can potentially calculate permanent value of an escrow incorrectly	✓ RESOLVED
24	INFO	Typographical issues and gas optimizations	PARTIAL

1.3.5 EscrowDelegateStorage

No issues found.

2 Findings

2.1 VotingEscrowV2Upgradeable

`VotingEscrowV2Upgradeable` is the backbone of the architecture. It allows users to wrap/lock their Lynex token and receive an NFT. Depending on the locked amount and time, users will gain voting power (VP) which they can use for voting purposes within the `VoterV5` contract.

Users can execute the following actions:

- `createLock`: A user can create a simple lock with the desired amount and duration. This lock is inherently granted to the user without any delegation.

- `createLockFor`: A user can create a simple lock with the desired amount and duration for another user. This lock is inherently granted to the recipient without any delegation.

- `createDelegatedLockFor`: A user can create a simple lock with the desired amount and duration for another user. This lock is inherently granted to the recipient but with an optional delegation possibility.

- `increaseAmount`: A user can increase the lock amount of any `tokenId` by depositing on behalf of it.

- `increaseUnlockTime`: A user can increase the `unlockTime` of any approved `tokenId`.

- `unlockPermanent`: A user can remove the permanent status of any approved `tokenId`. This will set the `unlockTime` to `block.timestamp` + 2 years.

- `claim`: A user can claim any approved and expired lock. This will transfer out the amount and set the `tokenId` states to zero.

- `merge`: A user can merge one `tokenId` to another `tokenId`. This will simply merge amounts and use the longer of both `endTimes`. The user must be approved for both `tokenIds`.

- `split`: A user can split an approved `tokenId` to multiple new `tokenIds`. This will simply decrease the amount of the original `tokenId` and mint new `tokenIds` with the same `unlockTime` and the corresponding amounts.

- `burn`: A user can burn their own `tokenId` if the amount is zero.

- `delegate`: A user can delegate any approved `tokenId` to a delegatee.

2.1.1 Privileged Functions

- `increaseUnlockTime [OWNER or APPROVED]`
- `unlockPermanent [OWNER or APPROVED]`
- `claim [OWNER or APPROVED]`
- `merge [OWNER or APPROVED]`
- `split [OWNER or APPROVED]`
- `burn [OWNER]`
- `delegate [OWNER or APPROVED]`

2.1.2 Issues & Recommendations

Issue #01	Malicious users can DOS token transfers
Severity	HIGH

Description	After the initial audit, the team implemented the following check within `_beforeTokenTransfer()`:
	`if(lockModifiedAt[tokenId] > block.timestamp - 60) {`
	`    revert LockModifiedDelay();`
	`}`
	It was included as a resolution to a front-running exploit through `split()` and `claim()`.
	Now all modifications to the NFT lock activate the lock and prevent transfers for some time as a safety mechanism.
	`IncreaseAmount()` is one such function that increases the vesting amount of a locked NFT. The issue is that anyone can call it with a minimum amount of 1 wei, which allows a malicious user to constantly reactivate the lock and practically DOS the token transfers at almost no cost.

Recommendation	Consider making the `increaseAmount` function permissioned with `isAuthorized` so that only approved parties or the owner can actually increase the amount. We do not see any benefit for this function to be permissionless. Another possible solution is to define a minimum amount that the caller can send. Additionally, if applicable, consider not updating the the lock in `increaseAmount()` since it only increases the amounts and front-running will not be as detrimental as in the case with `claim()` and `split()` (which reduces the amounts).
Resolution	✓ RESOLVED

Issue #02	Lack of minimum amount for a lock leads to users not getting any voting power
Severity	● MEDIUM SEVERITY

Description	There are two instances where the lack of a minimum amount for a lock will result in no voting power for a lock.
	
	<u>When creating a new lock</u>
	
	When creating a new lock, the user deposits an amount of the underlying token, based on which his total voting power gets calculated and checkpointed.
	
	Due to the calculation when checkpointing the voting power, the minimum amount of tokens that should be deposited so that the calculation does not round down to 0 is ~63e8.
	
	`if (uNewEndTime > block.timestamp && uNewAmount > 0) {`
	`    uNewPoint.slope = (uNewAmount) / MAX_TIME;`
	`    uNewPoint.bias = (uNewPoint.slope * (uNewEndTime - block.timestamp).toInt128());`
	`}`
	
	If, for example, the underlying `_token` is USDC, this would mean that all deposits below 63 USDC will not get any voting power even though they should. And if another 8 decimals token is used that is more valuable than USDC, the value of the tokens could be much higher. We do see that the team has stated this in a comment but there is nothing that stops a user from making such a deposit and locking his assets in the contract without getting anything in return.
	
	<u>During split</u>
	
	When splitting, a user splits a certain lock into multiple sub-locks by dividing the lock value into smaller chunks. During this action, as described above, if a value for a sub-lock is lower than the minimum amount (~63e8), then the sub-lock will have 0 voting power due to the round down.
	
	The issue is marked as medium because it can be mitigated after it occurs by increasing the locked amount via the `increaseLockAmount` function until it does not round down.

Recommendation	Consider adding a `MIN_AMOUNT` parameter within `createLock()` that would revert the transaction in case the deposit is below the threshold for receiving any voting power. Consider adding a `MIN_AMOUNT` check within `split()` after computing every sub-lock amount.
Resolution	● ACKNOWLEDGED The team stated that due to contract size limit this change was not included in the fix as users can always increase their lock.

Issue #03	**Not all fields are set on a merged lock**
Severity	● MEDIUM SEVERITY

Description	Within `merge()`, the `startTime` field is omitted from the parameters of the lock being merged so it does not get updated.
	`// Calculate the new lock details`
	`LockDetails memory newLockedTo;`
	`newLockedTo.amount = oldLockedTo.amount +`
	`oldLockedFrom.amount;`
	`newLockedTo.isPermanent = oldLockedTo.isPermanent;`
	`if (!newLockedTo.isPermanent) {`
	`    newLockedTo.endTime = end;`
	`}`
	Since this lock has already been created, its `startTime` should be copied into `newLockedTo`, otherwise the struct is not updated properly and can lead to unintended behavior.
	For example, the `_startTime()` function will return zero which will not be true for that lock. `_startTime()` is used within the crucial `vestingPeriod()` function of `ERC5725Upgradeable`.
	We marked this issue as low due to the fact that the start time is used only in `view` functions, but this can become a high issue if a third party relies on the `vestingPeriod` function to perform further actions.

Recommendation	When merging the data for the two NFTs, consider setting the start time as `min(from.startTime, to.startTime)`.
Resolution	✓ RESOLVED

Issue #04	The MAX_TIME lock restriction is obsolete
Severity	● LOW SEVERITY
Description	When creating a lock via `_createLock()`, there is a check that ensures it cannot be extended past the limit: `if (unlockTime > block.timestamp + MAX_TIME) revert LockDurationTooLong();` However, in a separate function `increaseUnlockTime()`, the same check from above is implemented, which practically means that the owner can increase the lock indefinitely and defeats the purpose of the restriction when creating the lock. Furthermore, switching from non-permanent to permanent to non-permanent also bypasses the initial `MAX_TIME`.
Recommendation	If the idea is to to have an absolute limit on the lock time, consider modifying the check inside `increaseUnlockTime()` to use `lock.startTime` instead of `block.timestamp`: `if (unlockTime > lock.startTime + MAX_TIME) revert LockDurationTooLong();` If the purpose of `MAX_TIME` is to just be a guardrail to avoid accidentally updating a lock time to a value very far in the future, then the feature works as expected and should not be changed.
Resolution	● ACKNOWLEDGED The team stated: "The `MAX_TIME` restriction is due to the math in `EscrowDelegateCheckpoints.checkpoint()`. Being able to continually increase your lock is intended behavior."

Issue #05	The logic behind the permanent lock is ambiguous
Severity	● LOW SEVERITY
Description	A lock can be defined as a permanent lock, which means it never expires. The current implementation for this sets a special parameter, called `permanent`, inside the `LockDetails` struct, along with setting the `endTime` to 0. By using this approach, it can be misinterpreted as a lock that does not exist or is empty. When a lock is claimed, the `LockDetails` for that `tokenId` is set to `LockDetails(0, 0, 0, false)`. We can see how this can be easily confused with a lock that is permanent if sorted by `endTime`. The described ambiguous logic is enforced by the EIP-5725 `view` functions like `vestingPeriod` which looks only for start time and end time. Furthermore, `vestedPayoutAtTime` checks that timestamp is greater than `endTime` (which in case of a `permanent` is 0) then it returns the full vested amount which in the `permanent` case is false.
Recommendation	Consider using `type(uint48).max` to mark a permanent lock. This is by the maximum value of 281474976710655, which is high enough to not collide with any real future timestamp.
Resolution	✓ RESOLVED The `_endTime` returns `type(uint48).max`.

Issue #06	Expired locks can be reactivated
Severity	● LOW SEVERITY
Description	We identified two cases where expired locks can be made active which, in theory, breaks the lifetime flow of lock. `increaseUnlockTime()` checks if modified lock is expired only if the new state of the lock is not permanent — this means that an expired lock can be transformed into a permanent lock. The `merge()` function requires that `toLock` is not expired. However there is no such requirement for `fromLock`. There is no immediate risk, but the team should consider if the expiry of `fromLock` should be checked as well.
Recommendation	Consider checking if the lock is expired transforming it into a permanent lock with `increaseUnlockTime` function. Consider checking if `fromLock` is expired as well in the merge function.
Resolution	✓ RESOLVED

Issue #07	`claimablePayout` will fail with arithmetic error
Severity	● LOW SEVERITY

Description	The `claimablePayout` `view` function from the ERC5725 implementation is used to determine how much locked value can be claimed for a specific token id at the current timestamp.
	
	This function will fail with an arithmetic error instead of returning 0 if the amount has been claimed already or if the previously claimed amount is greater than the current claimed amount.
	
	`function claimablePayout(`
	`        uint256 tokenId`
	`    ) public view override(IERC5725Upgradeable)`
	`validToken(tokenId) returns (uint256 payout) {`
	`    return vestedPayout(tokenId) - _payoutClaimed[tokenId];`
	`}`
	
	The revert will occur because `vestedPayout` returns 0 (no value left to be vested) or the value is lower than `_payoutClaimed`.
	
	When claiming a lock's value, it should be specified that the token is not burnt but the locked value is set to 0.

Recommendation	Consider burning the token once a lock is claimed, and if a user wishes to relock, then it should be done with a new token.
Resolution	✓ RESOLVED

Issue #08	The upgradeable version of `ReentrancyGuard` should be used
Severity	INFORMATIONAL
Description	Since the voting escrow contract is supposed to be upgradeable, the upgradeable versions of the OpenZeppelin contracts it uses should be used (as explicitly described in their docs). In particular, `ReentrancyGuard` is used instead of `ReentrancyGuardUpgradeable`. This has already been noted in the developer's comments but it should be implemented. There is a difference between the two where the regular version uses the constructor to set the initial state to `NOT_ENTERED`, while the latter uses an initializer function. There is no direct risk, but it is recommended to be consistent and use the contracts that were built with upgradeability in mind.
Recommendation	Consider using `ReentrancyGuardUpgradeable`.
Resolution	✓ RESOLVED

Issue #09	Lack of validation in the constructor
Severity	● INFORMATIONAL
Description	The constructor of the contract is used to set the `_token` and `artProxy` variables. However, there is no validation in both places for `address(0)` which can lead to improper configuration of the protocol. There should also be `address(0)` checks in: - `createLockFor()` - `createDelegatedLockFor()` Within `_createLock()`, check that `duration` is not 0 and return early.
Recommendation	Consider adding a check that ensures the parameters above cannot be 0.
Resolution	● PARTIALLY RESOLVED 1. `createLockFor` and `createDelegatedLockFor` already have `address(0)` validation in `_mint()` 2. `_createLock` already has a validation for 0 duration 3. Added validation for `mainToken`, but left out `artProxy` due to size limitations

Issue #10	Splits with zero or small weights should be prohibited
Severity	● INFORMATIONAL
Description	When a user wishes to split its lock into multiple locks, an array of weights is provided. These weights represent the percentage of the total value of the locked tokens each subsequent token will receive. A check is missing that should verify if a weight is 0 or if the yielded value for a specific weight is 0 to avoid creating lock tokens with 0 value locked. Furthermore, an extra comment should be added to this function to inform users that weights should be denominated into a certain value, e.g. X out of 10_000 or out of `1e18` so that it can be easily integrated by third parties.
Recommendation	Consider checking if the yielded value after the weight at a certain position `i` if applied is 0, if so then revert. A user must choose the right weight to avoid yielding 0 value locks.
Resolution	● ACKNOWLEDGED The team stated that `_createLock` will revert for zero amount locks.

Issue #11	Inefficient `LockMismatch` check when merging
Severity	● INFORMATIONAL
Description	The `merge` function implements a check to ensure that a non-permanent lock cannot be merged into a permanent lock and vice versa: `if (oldLockedFrom.isPermanent == true && oldLockedFrom.isPermanent != oldLockedTo.isPermanent) revert PermanentLockMismatch();` The check is inefficient because its first condition requires that `oldLockedFrom` is permanent in order to conduct the mismatch check. If `oldLockedTo` is permanent, the mismatch validation will not be conducted and this will allow an unexpected merge.
Recommendation	Consider updating the check so that it assumes all cases of mismatched `PermanentLock`: `if (oldLockedFrom.isPermanent != oldLockedTo.isPermanent) revert PermanentLockMismatch()`
Resolution	● ACKNOWLEDGED The team stated: "This check is intended. It prevents permanent locks from being unlocked through a merge, and for non-permanent locks to be able to be merged into permanent locks."

Issue #12	Tokens resulting from a split can not have a designated delegate
Severity	● INFORMATIONAL
Description	When a user splits their lock into multiple locks for every new lock, the delegate is set as an owner.
Recommendation	Consider adding an array of delegates that matches the array of weights which in theory matches the amount of tokens that will result in a split.
Resolution	● ACKNOWLEDGED The team stated: "Intended and noted in the comments."

Issue #13	`getPastTotalSupply` can look for future total supply
Severity	INFORMATIONAL
Description	`getPastTotalSupply` is used to get the total supply at a certain point in time. There is no check if the point in time transmitted as a parameter is in the future.
Recommendation	Consider adding a check that the point in time parameter is not in the future.
Resolution	ACKNOWLEDGED The team stated: "Leaving as is. If the timestamp is in the future, it will return the latest value."

Issue #14	No setter function for `artProxy`
Severity	INFORMATIONAL
Description	The `artProxy` contract address is set in the constructor, but there is no function to update it if necessary. `artProxy` is upgradeable so there might not be a reason to change it, but in case such a need might arise in future, it would be better to have an owner-restricted setter function. Otherwise, if it should be set only once, declare `artProxy` as immutable.
Recommendation	Consider implementing a setter function or declare `artProxy` as immutable.
Resolution	ACKNOWLEDGED The team stated: "No owner for `VEv2` contract. `ArtProxy` can't be immutable since there is an initializer. Added a validation check in the constructor."

Issue #15	Event not emitted properly
Severity	INFORMATIONAL
Description	The `_updateLock` function emits the `SupplyUpdated()` event. The parameters of this event are incorrectly emitted. The first parameter should be the supply before but it currently emits the updated supply. The second parameter should be the updated supply and it emits the updated supply but by performing the addition that was done already at the beginning of the function. This event is also emitted at every execution. It should only be emitted if the supply was actually changed.

Recommendation	Consider emitting the right parameters:
	`emit SupplyUpdated(supplyBefore, supply);`
	
	Additionally, consider moving the event inside the check that validates if new value was sent:
	`if (_increasedValue != 0 && depositType != DepositType.SPLIT_TYPE) {`
	`    [...]`
	`    emit SupplyUpdated()`
	`}`

Resolution	✓ RESOLVED

Issue #16	Typographical issues
Severity	● INFORMATIONAL
Description	Consider renaming `VotingEscrowV2Upgradeable()` to `__VotingEscrowV2Upgradeable__init()` to make it clear that it is an initializer function. — Consider removing the underscore from the public state variable `_token`. The convention is to use underscores only for private/internal variables/functions. — Consider moving the `_lockDetails` and `totalNftsMinted` variables to the top instead of placing them in the middle of the contract. It creates a clear separation of concerns. — Consider renaming `globalCheckpoint()` to `updateGlobalCheckpoint()` to clearly signal it is a state changing function. — Consider renaming the `NoLockFound()` error within `increaseAmount()` to `ERC721NonexistentToken()`. — Within `tokenURI`, a `uint` type is used instead of `uint256`. — `totalNftsMinted` being initialized to 0 in the definition is redundant and can be removed. — Both `msg.sender` and `_msgSender()` are used throughout the contract. Consider sticking to one convention.

	Line 524
	`// reset supply, _deposit_for increase it.`
	
	This is a typographical error as there is no `_deposit_for` but `_updateLock`.
	
	—
	
	All functions are marked with this dev comment:
	`@dev See {IERC5725}`
	
	It requires the reader to manually find the interface file and read the actual description of the function and lose context along the way.
	
	Consider changing it to
	`@inheritdoc IERC5725`
	
	This instructs the Solidity compiler to automatically get and display the description from the interface upon hovering on the function.
	
	—
	
	There are several incorrect or missing NatSpec comments. Reference: https://drive.google.com/file/d/1h_D9l6YI-oHIcXWqHWmYCYmH2jOQwa6A/view?usp=sharing

Recommendation	Consider fixing the typographical issues.
Resolution	✓ RESOLVED

Issue #17	Gas optimizations
Severity	● INFORMATIONAL
Description	The constant variable `decimals` is not used anywhere in code. Consider removing it if not needed. — `delegateBySig()` and `DELEGATION_TYPEHASH` are not used. Consider removing them to reduce contract size and deployment costs. — Within `checkAuthorized`, check and revert early if `_tokenId` is 0 to prevent unnecessary ownership checks. The same applies to `tokenURI()`. — Within `increaseUnlockTime()`, move the following check to the beginning to return early and prevent doing unnecessary checks: `if (oldLocked.endTime <= block.timestamp) revert LockExpired();` — Consider removing the `validToken` modifier in `_claim()` since the second modifier `checkAuthorized` already does the check. — Consider removing the `token()` function which is redundant since the underlying `_token` is already public and has a getter. — Within `split()`, calculate and revert early if `duration < CLOCK_UNIT` because it will revert anyway in `_createLock()` and waste gas.

	The `LockDetails` struct can be optimized by defining end and start time as `uint48`:
	`struct LockDetails {`
	`    uint256 amount; /// @dev amount of tokens locked`
	`    uint48 startTime; /// @dev when locking started`
	`    uint48 endTime; /// @dev when locking ends`
	`    bool isPermanent; /// @dev if its a permanent lock`
	`}`
	
	—
	
	<u>In split</u>
	`uint256 duration = locked.isPermanent ? 0 : locked.endTime > currentTime ? locked.endTime - currentTime : 0`
	
	This can be optimized to `uint256 duration = locked.isPermanent ? 0 : locked.endTime - currentTime;` because it is already known that `locked.endTime > currentTime`.
	
	—
	
	Within split, the following conversion is obsolete: `supply -= uint256(int256(locked.amount));`
	
	—
	
	Within merge, the `PermanentLockMismatch` can be simplified if `(oldLockedFrom.isPermanent && !oldLockedTo.isPermanent) revert PermanentLockMismatch();`

Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	✓ RESOLVED

2.2 ERC5725Upgradeable

`ERC5725Upgradeable` is an abstract contract that implements EIP-5725 for transferable vesting NFTs. The EIP-5725 represents a framework to release tokens vested over a specific period of time.

2.2.1 Issues & Recommendations

Issue #18	Contract will fail to initialize after deployment
Severity	● MEDIUM SEVERITY
Description	The initialize function `__ERC5725_init()` uses the `initializer` modifier instead of `onlyInitializing`. As a result, calling `VotingEscrowV2Upgradeable.VotingEscrowV2Upgradeable()` initializing function will fail because it will block the execution of the internally called `__ERC5725_init()`. When contracts with initializers are inherited, their initializer functions should use the `onlyInitializing` modifier so that the inheriting contract can safely call them

Recommendation	Consider modifying the initializer function like the following:
	`function __ERC5725_init(....) internal onlyInitializing  {`
	`    [...]`
	`}`

Resolution	✓ RESOLVED

Issue #19	Insufficient validation
Severity	● LOW SEVERITY
Description	Within `setClaimApproval()`, there should be a check that the operator and `tokenId` are not the 0 values. The same goes for `setClaimApprovalForAll()` and `isApprovedClaimOrOwner()`.
Recommendation	Consider implementing the above recommendations
Resolution	● ACKNOWLEDGED

Issue #20	Typographical issues and gas optimizations
Severity	● INFORMATIONAL

Description	Within `_setClaimApproval()`, consider using the internal `_ownerOf()` instead of the external `ownerOf()` to reduce gas usage.
	
	—
	
	Within `_beforeTokenTransfer()`, the `from` variable is checked on each iteration. Since it is the same parameter, it can be checked only once and reused: `fromNotZero = from != address(0)`.
	
	—
	
	Missing NatSpec on the `init` function:
	`/**`
	`     * @notice Initializes the contract with the given name and symbol.`
	`     * @param name_ The name of the token.`
	`     * @param symbol_ The symbol of the token.`
	`     */`
	
	—
	
	Missing `__ERC165_init` call in the `init` function.

Recommendation	Consider implementing the above recommendations.
Resolution	✓ RESOLVED

2.3 Checkpoints

`Checkpoints` is a library that defines structures and functions for checkpointing values. It is useful for recording the history of values that change over time, such as a voting power.

2.3.1 Issues & Recommendations

Issue #21	Array size can grow too big
Severity	● LOW SEVERITY
Description	The `Checkpoint` library uses a `Trace` struct with an array for storing checkpoints. Each new checkpoint is pushed to the array. It is possible that the array can grow too big which might lead to DOS when executing `binarySearch` in `lower/upperLookup()` functions. Checkpoints are created on almost any action in the voting escrow contract, thus the possibility of this should be considered.
Recommendation	We recommend that the team monitor the array and consider implementing safeguards that prevent the array from growing too big. If appropriate, a function can be added to allow the array to be emptied, or to transfer the recent checkpoints into a new array and free space for the new checkpoints to prevent DOS.
Resolution	● ACKNOWLEDGED The team stated: "Rationale: The values below help support the acknowledgment of the issue. `_globalCheckpoints` will be the largest array as it holds checkpoints for every `VotingEscrow` action. The most compute intensive operation will come from `EscrowDelegateCheckpoints._getAdjustedCheckpoint()`, which performs an `upperLookupRecent()` on `store_._globalCheckpoints`. `O(log2(#checkpoints))` represents the number of iterations required for the binary search algorithm. As of 2024-09-05, there has been 210k transactions on the `VotingEscrowV2Upgradeable` contract which has been deployed for about 6 months."

Issue #22	Gas optimizations
Severity	● INFORMATIONAL
Description	Within `upperLookup()` and `upperLookupRecent()`, consider caching `_unsafeAccess(self._checkpoints, pos - 1)` instead of calculating it every time.
Recommendation	Consider implementing the recommendations above.
Resolution	● ACKNOWLEDGED

2.4 EscrowDelegateCheckpoints

`EscrowDelegateCheckpoints` is a library designed to manage and record checkpoints for an escrow system. It tracks changes in voting power and token locks over time, allowing for efficient lookups of historical data. The key features include:

- Checkpoint Management: Records changes in token locking and voting power at specific timestamps.

- Slope and Bias Tracking: Manages the rate of change (slope) and the total amount (bias) of locked tokens, which affect voting power.

- Global and Delegate Checkpoints: Separately tracks global checkpoints and delegate-specific checkpoints.

- Efficient Lookups: Provides functions to efficiently retrieve historical values at specific timestamps

2.4.1 Issues & Recommendations

Issue #23	**`checkpoint()` can potentially calculate permanent value of an escrow incorrectly**
Severity	INFORMATIONAL

Description	Within `checkpoint()`, delegatee values of a `delegateeAddress` is calculated by subtracting the old value and then adding the new values.
	`if (delegateTs != 0) {`
	`    _checkpointDelegatee(store_, delegateeAddress, uOldPoint, uOldEndTime, false);`
	`    _checkpointDelegatee(store_, delegateeAddress, uNewPoint, uNewEndTime, true);`
	`}`
	
	Within `_checkpointDelegatee`, the `permanent` value of a point is calculated as follows:
	`lastPoint.permanent = escrowPoint.permanent < lastPoint.permanent ? lastPoint.permanent - escrowPoint.permanent : int128(0);`
	
	If the current point is less `permanent` than the last point, then it will simply be set to 0. Due to the above calculation, for example, when we want to increase permanent value of a delegatee address in `checkpoint()` and the values are:
	
	`uOldAmount = 7`
	`uNewAmount = 10`
	
	To increase `permanent` by 10-7=3 and when the current `permanent` value is 5, the expected value should be 5+3=8. However, due to the order in which calculations are done, which is 5 - 7 = -2, this gets set to 0 and then 10 is added to it, resulting in 10. Thus, the value of `permanent` is over-calculated by 2. However, we do not see a reasonable path where `delegateeAddress.permanent < uOldPoint.permanent` in normal use cases, hence we rate this issue as informational.

Recommendation	Consider switching the order in which `_checkpointDelegatee()` is called. The increase calculation can be done before the decrease calculation.
Resolution	✓ RESOLVED

Issue #24	Typographical issues and gas optimizations
Severity	● INFORMATIONAL

Description	Consider changing `MAX_TIME` from `2 * 365 * 86400` to `2 * 365 days` for brevity.
	
	—
	
	Within `_getAdjustedCheckpoint`, `clockTime` variable is redundant as `timestamp` can be used.
	
	—
	
	Within `delegate`, the check `if (oldDelegatee != delegatee && oldDelegatee != address(0)) {` can be simplified to `if (oldDelegatee != address(0)) {` because it is already known that `oldDelegatee != delegatee`.
	
	—
	
	Throughout the contract, the `testTime` is misspelled in the comments as `tesTime`.
	
	—
	
	Line 130
	`if ((uNewEndTime > uOldEndTime))`
	
	The extra parentheses can be deleted.
	
	Line 131
	`newDslope -= uNewPoint.slope;`
	
	This can be deleted and `globalSlopeChanges` can be updated directly.
	
	—
	
	There are several incorrect or missing NatSpec comments. Reference: https://drive.google.com/file/d/1DkxcFFaMnjFI6U9yhsla7kz4HHuv9D8S/view?usp=sharing

Recommendation	Consider implementing the recommended changes.
Resolution	● PARTIALLY RESOLVED

2.5 EscrowDelegateStorage

`EscrowDelegateStorage` is the storage contract used to store the checkpoint system.

2.5.1 Issues & Recommendations

No issues found.

PALADIN

BLOCKCHAIN SECURITY