CONTRACT WOLF

Security Assessment

Titan Locker V2

Verified on 07/18/2026

SUMMARY

Project	CHAIN	METHODOLOGY
Titan Locker V2	Robinhood Chain	Manual & Automatic Analysis

FILES	DELIVERY	TYPE
Repository	07/18/2026	Standard Audit

4	0	1	2	0	1	4
Total Findings	Critical	Major	Medium	Minor	Informational	Resolved

🟥 0 Critical	An exposure that can affect the contract functions in several events that can risk and disrupt the contract
🟧 1 Major	An opening & exposure to manipulate the contract in an unwanted manner
🟧 2 Medium	An opening that could affect the outcome in executing the contract in a specific situation
⬜ 0 Minor	An opening but doesn't have an impact on the functionality of the contract
🟦 1 Informational	An opening that consists information but will not risk or affect the contract
🟩 4 Resolved	ContractWolf's findings has been acknowledged & resolved by the project

STATUS ✔️**AUDIT PASSED**

TABLE OF CONTENTS | Titan Locker V2

DISCLAIMER | Titan Locker V2

ContractWolf audits and reports should not be considered as a form of project's "Advertisement" and does not cover any interaction and assessment from "Project Contract" to "External Contracts" such as PancakeSwap, UniSwap, SushiSwap or similar.

ContractWolf does not provide any warranty on its released report and should not be used as a decision to invest into audited projects.

ContractWolf provides a transparent report to all its "Clients" and to its "Clients Participants" and will not claim any guarantee of bug-free code within its **SMART CONTRACT**.

ContractWolf's presence is to analyze, audit and assess the Client's Smart Contract to find any underlying risk and to eliminate any logic and flow errors within its code.

Each company or project should be liable to its security flaws and functionalities.

SCOPE OF WORK | Titan Locker V2

Titan Locker V2 team has agreed and provided us with the files that need to be tested (*Github, BSCscan, Etherscan, Local files etc*). The scope of audit is the main contract.

The goal of this engagement is to identify if there is a possibility of security flaws in the implementation of smart contract and its systems.

ContractWolf will be focusing on contract issues and functionalities along with the project claims from smart contract to their website, whitepaper, repository which has been provided by **Titan Locker V2**.

AUDITING APPROACH | Titan Locker V2

Every line of code along with its functionalities will undergo manual review to check for security issues, quality of logic and contract scope of inheritance. The manual review will be done by our team that will document any issues that they discovered.

METHODOLOGY

The auditing process follows a routine series of steps :

1. Code review that includes the following :
   - Review of the specifications, sources and instructions provided to ContractWolf to make sure we understand the size, scope and functionality of the smart contract.
   - Manual review of code. Our team will have a process of reading the code line-by-line with the intention of identifying potential vulnerabilities, underlying and hidden security flaws.

2. Testing and automated analysis that includes :
   - Testing the smart contract function with common test cases and scenarios to ensure that it returns the expected results.

3. Best practices and ethical review. The team will review the contract with the aim to improve efficiency, effectiveness, clarifications, maintainability, security and control within the smart contract.

4. Recommendations to help the project take steps to eliminate or minimize threats and secure the smart contract.

TOKEN DETAILS | Titan Locker V2

-

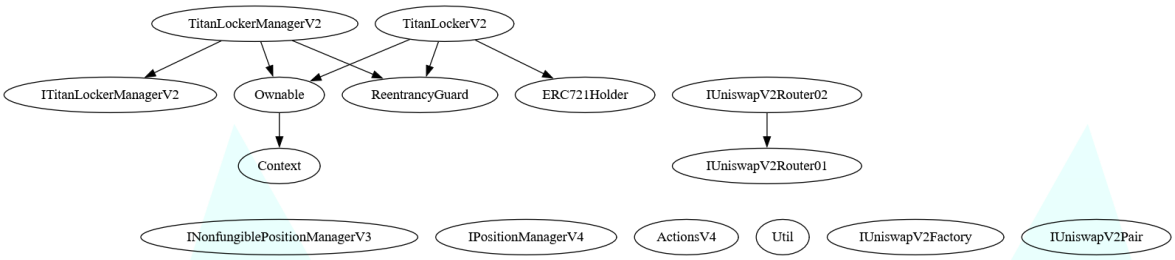
Token Name	Symbol	Decimal	Total Supply	Chain
--	--	--	--	Robinhood Chain

SOURCE

Source	*Sent Via local-files*

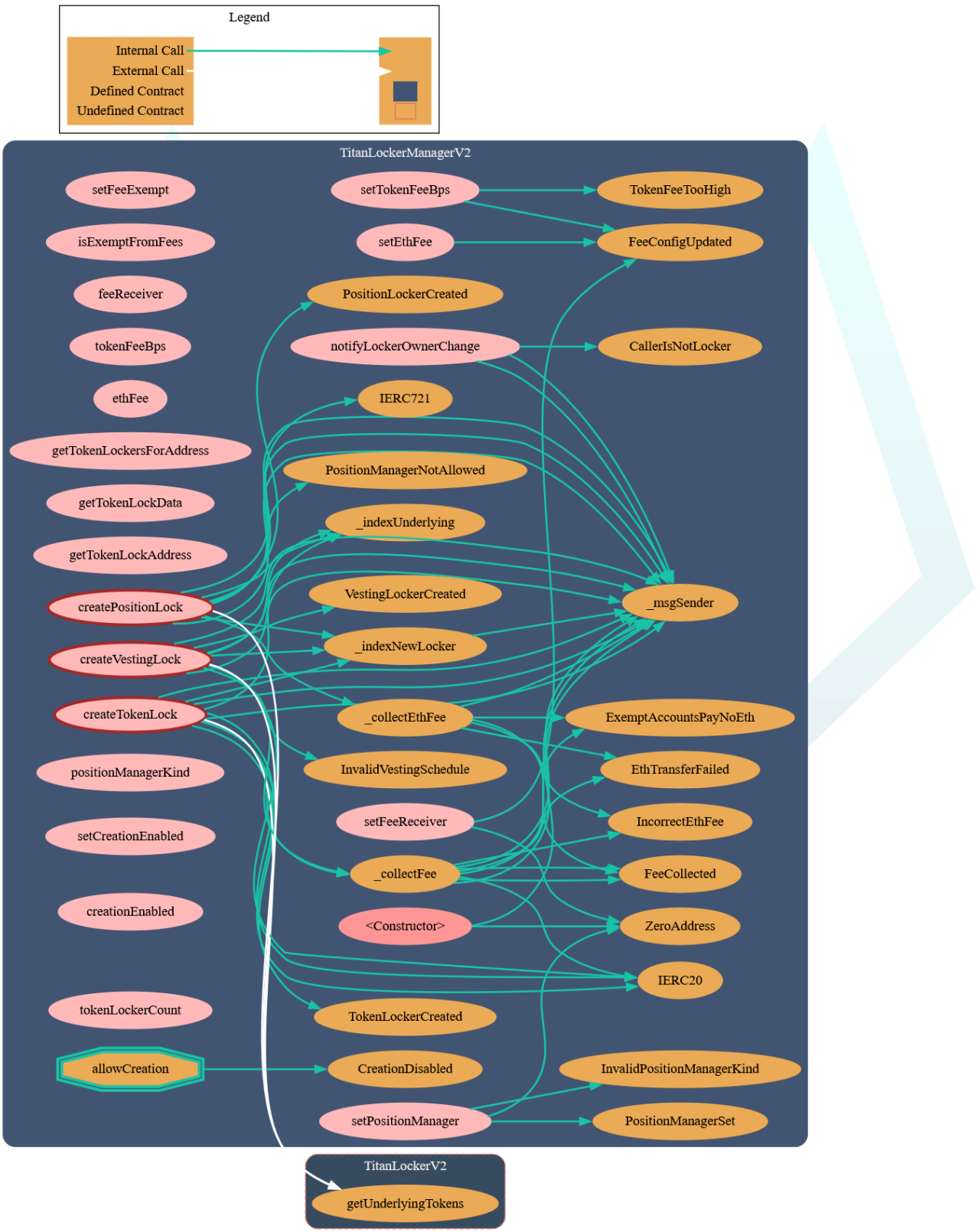
INHERITANCE GRAPH | Titan Locker V2

Inheritance Graph of Contract Functions



CALL GRAPH | Titan Locker V2

Call Graph of Contract Functions



FINDINGS | Titan Locker V2

4	0	1	2	0	1	4
Total Findings	Critical	Major	Medium	Minor	Informational	Resolved

This report has been prepared to state the issues and vulnerabilities for Titan Locker V2 through this audit. The goal of this report findings is to identify specifically and fix any underlying issues and errors

ID	Title	File & Line #	Severity	Status
SWC-114	Transaction Order Dependence	TitanLockerManagerV2.sol	Major	Resolved
Logic Error	Bps rounding lets dust locks pay zero fee	TitanLockerManagerV2.sol	Medium	Resolved
Logic Error	collectFees() misreports amounts for UNIV4 locks	TitanLockerV2.sol	Medium	Resolved
Logic Flaw	Missing Validation Allows Zero-Amount Locks	TitanLockerManagerV2.sol	Informational	Resolved

SWC ATTACKS | Titan Locker V2

Smart Contract Weakness Classification and Test Cases

ID	Description	Status
SWC-100	Function Default Visibility	Passed
SWC-101	Integer Overflow and Underflow	Passed
SWC-102	Outdated Compiler Version	Passed
SWC-103	Floating Pragma	Passed
SWC-104	Unchecked Call Return Value	Passed
SWC-105	Unprotected Ether Withdrawal	Passed
SWC-106	Unprotected SELF DESTRUCT Instruction	Passed
SWC-107	Reentrancy	Passed
SWC-108	State Variable Default Visibility	Passed
SWC-109	Uninitialized Storage Pointer	Passed
SWC-110	Assert Violation	Passed
SWC-111	Use of Deprecated Solidity Functions	Passed
SWC-112	Delegate call to Untrusted Callee	Passed
SWC-113	DoS with Failed Call	Passed
SWC-114	Transaction Order Dependence	Passed
SWC-115	Authorization through tx.origin	Passed
SWC-116	Block values as a proxy for time	Passed
SWC-117	Signature Malleability	Passed
SWC-118	Incorrect Constructor Name	Passed
SWC-119	Shadowing State Variables	Passed
SWC-120	Weak Sources of Randomness from Chain Attributes	Passed
SWC-121	Missing Protection against Signature Replay Attacks	Passed
SWC-122	Lack of Proper Signature Verification	Passed

ID	Description	Status
SWC-123	Requirement Violation	Passed
SWC-124	Write to Arbitrary Storage Location	Passed
SWC-125	Incorrect Inheritance Order	Passed
SWC-126	Insufficient Gas Griefing	Passed
SWC-127	Arbitrary Jump with Function Type Variable	Passed
SWC-128	DoS With Block Gas Limit	Passed
SWC-129	Typographical Error	Passed
SWC-130	Right-To-Left-Override control character(U+202E)	Passed
SWC-131	Presence of unused variables	Passed
SWC-132	Unexpected Ether balance	Passed
SWC-133	Hash Collisions With Multiple Variable Arguments	Passed
SWC-134	Message call with hardcoded gas amount	Passed
SWC-135	Code With No Effects	Passed
SWC-136	Unencrypted Private Data On-Chain	Passed

CW ASSESSMENT | Titan Locker V2

ContractWolf Vulnerability and Security Tests

ID	Name	Description	Status
CW-001	Multiple Version	Presence of multiple compiler version across all contracts	✔
CW-002	Incorrect Access Control	Additional checks for critical logic and flow	✔
CW-003	Payable Contract	A function to withdraw ether should exist otherwise the ether will be trapped	✔
CW-004	Custom Modifier	major recheck for custom modifier logic	✔
CW-005	Divide Before Multiply	Performing multiplication before division is generally better to avoid loss of precision	✔
CW-006	Multiple Calls	Functions with multiple internal calls	✔
CW-007	Deprecated Keywords	Use of deprecated functions/operators such as block.blockhash() for blockhash(), msg.gas for gasleft(), throw for revert(), sha3() for keccak256(), callcode() for delegatecall(), suicide() for selfdestruct(), constant for view or var for actual type name should be avoided to prevent unintended errors with newer compiler versions	✔
CW-008	Unused Contract	Presence of an unused, unimported or uncalled contract	✔
CW-009	Assembly Usage	Use of EVM assembly is error-prone and should be avoided or double-checked for correctness	✔
CW-010	Similar Variable Names	Variables with similar names could be confused for each other and therefore should be avoided	✔
CW-011	Commented Code	Removal of commented/unused code lines	✔
CW-012	SafeMath Override	SafeMath is no longer needed starting with Solidity v0.8+. The compiler now has built-in overflow checking.	✔

FIXES & RECOMMENDATION

SWC-114 | Transaction Order Dependence

`_tokenFeeBps` is owner-settable up to `BPS_DENOMINATOR(100%)` via `setTokenFeeBps`. createTokenLock/createVestingLock take no parameter capping the fee the caller is willing to accept.

If the owner (or a compromised owner key) calls `setTokenFeeBps(10000)` while a user's `createTokenLock` transaction is pending in the mempool, the user's transaction will still execute but `amountToLock` becomes 0, silently confiscating their entire deposit as "fee" with no revert to warn them. The ETH-fee path is self-protecting (`IncorrectEthFee` reverts on mismatch), but the token-fee path has no equivalent guard.

```
uint256 tokenFeeAmount = (amount_ * _tokenFeeBps) / BPS_DENOMINATOR; amountToLock =
amount_ - tokenFeeAmount;
```

Recommendation

let the caller specify the maximum fee (in bps) they're willing to pay, and revert if the live fee exceeds it. The same pattern as `amountOutMin` on a DEX swap:

```
function createTokenLock(
    address tokenAddress_,
    uint256 amount_,
    uint40 unlockTime_,
    uint16 maxTokenFeeBps_          // NEW
) external payable override allowCreation nonReentrant {
    uint40 id = _tokenLockerCount++;
    uint256 amountToLock = _collectFee(id, tokenAddress_, amount_, maxTokenFeeBps_);
    //rest of the code
}

function _collectFee(
    uint40 id_,
    address tokenAddress_,
    uint256 amount_,
    uint16 maxTokenFeeBps_          // NEW
) private returns (uint256 amountToLock) {
    //rest of the code
```

Logic Error | collectFees() misreports amounts for UNIV4 locks

For UNIV4 locks, the named return variables `amount0`/`amount1` are only assigned in the `UNIV3` branch. The `else` (UNIV4) branch never assigns them, so they keep their default zero value. Fees are still transferred correctly to the owner via the `TAKE_PAIR` action (no fund loss), but:

- The function's return value is always `(0, 0)` for UNIV4 locks, breaking any caller/integration that relies on it.
- The `FeesCollected(amount0, amount1)` event always reports `(0, 0)` for UNIV4 locks, which will silently corrupt any off-chain indexer, dashboard, or accounting system built on this event.

```
function collectFees() external onlyOwner onlyPosition nonReentrant returns (uint256
amount0, uint256 amount1) {
    address recipient = _owner();

    if (_kind == ITitanLockerManagerV2.LockKind.UNIV3) {
        (amount0, amount1) = INonfungiblePositionManagerV3(_asset).collect(...);
    } else {
        // UNIV4 branch -- amount0/amount1 are NEVER assigned here
        IPositionManagerV4(_asset).modifyLiquidities(abi.encode(actions, params),
block.timestamp);
    }

    emit FeesCollected(amount0, amount1);
}
```

Recommendation

capture the actual amounts taken. If `IPositionManagerV4` doesn't return them directly from `modifyLiquidities`, read the balance delta (Logic Sample):

```
} else {
    address recipient_ = recipient;
    (PoolKeyV4 memory key, ) =
IPositionManagerV4(_asset).getPoolAndPositionInfo(_tokenId);

    uint256 bal0Before = IERC20(key.currency0).balanceOf(recipient_);
    uint256 bal1Before = IERC20(key.currency1).balanceOf(recipient_);

    bytes memory actions = abi.encodePacked(ActionsV4.DECREASE_LIQUIDITY,
ActionsV4.TAKE_PAIR);
    bytes[] memory params = new bytes[](2);
    params[0] = abi.encode(_tokenId, uint256(0), uint128(0), uint128(0), bytes(""));
```

```
    params[1] = abi.encode(key.currency0, key.currency1, recipient_);
    IPositionManagerV4(_asset).modifyLiquidities(abi.encode(actions, params),
block.timestamp);

    amount0 = IERC20(key.currency0).balanceOf(recipient_) - bal0Before;
    amount1 = IERC20(key.currency1).balanceOf(recipient_) - bal1Before;
}
```

Logic Error | Bps rounding lets dust locks pay zero fee

Integer division rounds down. At the default 500 bps (5%), any `amount_` such that `amount_ * 500 < 10000` (i.e. `amount_ < 20`) pays zero fee. Combined with no minimum-amount check on `createTokenLock`/`createVestingLock`, this lets an attacker mint themselves a cheap token and spam near-zero-value locks essentially fee-free — which directly feeds finding #2's array-growth DoS vector at negligible cost.

```
uint256 tokenFeeAmount = (amount_ * _tokenFeeBps) / BPS_DENOMINATOR;
```

Recommendation

either enforce a minimum `amount_` (e.g. require `amount_ >= someFloor` or that `tokenFeeAmount > 0` whenever a fee is expected), or round the fee up instead of down (`(amount_ * _tokenFeeBps + BPS_DENOMINATOR - 1) / BPS_DENOMINATOR`) so a nonzero fee is always taken on the token-fee path.

Logic Flaw | Missing Validation Allows Zero-Amount Locks

`createTokenLock` and `createVestingLock` do not check that `amount_ > 0`. A user can create a lock with zero tokens and still pay the creation fee (ETH or token). This results in an empty locker that serves no purpose, wastes gas, and clutters the on-chain index.

`createTokenLock` and `createVestingLock` – no `require(amount_ > 0)`.

Recommendation
Add an explicit check at the beginning of both functions

```
function createTokenLock( ... ) external payable ... {
    require(amount_ > 0, "Amount must be > 0");
    // ... rest of the function
}

function createVestingLock( ... ) external payable ... {
    require(amount_ > 0, "Amount must be > 0");
    // ... rest of the function
}
```

AUDIT COMMENTS | Titan Locker V2

Smart Contract audit comment for a non-technical perspective

🟢 Owner cannot modify existing locks
🟢 Contract cannot be paused
🟢 Owner can renounce and transfer ownership
🟢 Owner retain custody through isolated locker contracts
🟢 Owner controls fee configuration
🟢 Owner can disable creation of new lockers
🟢Owner cannot arbitrarily withdraw locked funds

CONTRACTWOLF

Blockchain Security - Smart Contract Audits