Ambire Security Audit

Report Version 0.1

February 22, 2024

Conducted by the **Hunter Security** team:
George Hunter, Lead Security Researcher

Table of Contents

1 About Hunter Security

Hunter Security is a duo team of independent smart contract security researchers. Having conducted over 50 security reviews and reported tens of live smart contract security vulnerabilities, our team always strives to deliver top-quality security services to DeFi protocols. For security review inquiries, you can reach out to us on Telegram or Twitter at *@georgehntr*.

2 Disclaimer

Audits are a time-, resource-, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

3 Risk classification

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

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - involves a small loss of funds or affects a core functionality of the protocol.
- **Low** - encompasses any unexpected behavior that is non-critical.

3.2 Likelihood

- **High** - a direct attack vector; the cost is relatively low compared to the potential loss of funds.
- **Medium** - only a conditionally incentivized attack vector, with a moderate likelihood.
- **Low** - involves too many or unlikely assumptions; offers little to no incentive.

3.3 Actions required by severity level

- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

The Hunter Security team was engaged by Ambire to review the stkWALLET smart contract during the period from February 21, 2024, to January 22, 2024.

Overview

Project Name	Ambire stkWALLET
Repository	https://github.com/AmbireTech/ambire-common
Commit hash	c544c6a4f10fd7546421b65b35e92caef8660d24
Resolution	-
Methods	Manual review

Timeline

-	February 21, 2024	Audit kick-off
v0.1	February 22, 2024	Preliminary report
v1.0	-	Mitigation review

Scope

contracts/stkWALLET.sol

Issues Found

High risk	0
Medium risk	0
Low risk	1
Informational	2

5 Consultants

George Hunter - a proficient and reputable independent smart contract security researcher with over 50 solo and team security engagements contributing to the security of numerous smart contract protocols in the past 2 years. Previously held roles include Lead Smart Contract Auditor at Paladin Blockchain Security and Smart Contract Engineer at Nexo. He has audited smart contracts for clients such as LayerZero, Euler, TraderJoe, Maverick, Ambire, and other leading protocols. George's extensive experience in traditional audits and meticulous attention to detail contribute to Hunter Security's reviews, ensuring comprehensive coverage and preventing vulnerabilities from slipping through.

6 System overview

The stkWALLET contract is a wrapper of the existing xWALLET token which itself represents a staking pool of WALLET tokens. The purpose of the wrapper is to improve UX by allowing users to pass the amount of tokens they would like to transfer denominated in WALLET instead of xWALLET shares. It supports all standard ERC20 methods. Minting and burning of stkWALLET tokens happens through the `wrap`/`wrapAll` and `unwrap` public functions.

6.1 Codebase maturity

Code complexity - *Excellent*
The contract is minimalistic and straightforward. There are no nested calls nor inherited contracts.

Security - *Excellent*
The scope consists of one simple ~50 nSLOC contract that has been thoroughly reviewed, and no security vulnerabilities were identified.

Decentralization - *Excellent*
There are no privileged entities.

Testing suite - *N/A*
Not provided.

Documentation - *Good*
The contract's code is self-explanatory.

Best practices - *Excellent*
The contract adheres to all best practices.

6.2 Privileged actors

- None.

6.3 Threat model

What in the stkWALLET contract has value in the market?

The minted stkWALLET tokens as well as the held (wrapped) xWALLET shares of the users which are exchangeable for WALLET tokens.

What are the worst-case scenarios for the stkWALLET contract?

- Stealing the wrapped xWALLET tokens.
- Manipulating accounting of users' balances.
- Incorrect implementation of the ERC20 standard causing integration issues.
- Being unable to unwrap tokens.

What are the main entry points and non-permissioned functions?

- `stkWALLET.wrap()` - deposits caller's specified amount of xWALLET shares and mints equivalent amount of stkWALLET tokens.
- `stkWALLET.wrapAll()` - same as `stkWALLET.wrap()` but the deposit amount is the whole caller's balance of xWALLET shares.
- `stkWALLET.unwrap()` - burns the specified amount of stkWALLET tokens and transfers the corresponding amount of xWALLET shares back to the caller.
- `stkWALLET.transfer()` - standard ERC20 `transfer` function. Instead of calculating the actual amount of stkWALLET tokens to transfer by simply using the specified `amount` argument, it divides it by the retrieved `xWallet.shareValue()` ratio.
- `stkWALLET.transferFrom()` - standard ERC20 `transferFrom` method. Implements the same accounting mechanism as described above.
- `stkWALLET.approve()` - standard ERC20 `approve` method. Implements the same accounting mechanism as described above.

6.4 Observations

Several interesting design decisions were observed during the review of the stkWALLET smart contract, some of which are listed below:

- The contract implements the ERC20 standard without inheriting from a major library implementation like OpenZeppelin's.
- `stkWALLET.totalSupply()` will also include tokens that have been sent directly to the contract rather than only the once deposited via the `wrap()` function.
- The contract determines how many shares to actually transfer by using the `xWallet.shareValue()` ratio and dividing the passed user amount by it. This may cause rounding issues if the `stkWALLET` is used for accounting in other protocols.

6.5 Useful resources

The following resources were used during the audit:

- *xWALLET contract's code*

7 Findings

7.1 Low

7.1.1 Sanity checks

Severity: *Low*

Context: stkWALLET.sol

Description: There are several places in the contract where sanity checks could be added to improve input validation and adhere to best practices implemented by other ERC20 token implementations:

1. Consider adding a zero-address check in `transfer` and `transferFrom` to prevent users from accidentally burning their shares. (implemented by OZ's ERC20)
2. Consider checking that the `from` address in `transferFrom` is not the zero address. Otherwise, user will be able to transfer 0 shares from the zero address which may be misinterpreted by off-chain scripts that rely on that Transfer events will be emitted with address(0) as the `from` address only upon mints. (implemented by OZ's implementations)
3. Consider implementing a check that prevents the caller of `approve` to give allowance to the zero address. (implemented by OZ's ERC20)

Recommendation: Consider implementing the aforementioned zero-value checks.

Resolution: Pending.

7.2 Informational

7.2.1 transferFrom will decrease the allowance even if set to max

Severity: *Informational*

Context: stkWALLET.sol#L56

Description: A general practice in ERC20 contract implementations is that `transferFrom` does not decrease the allowance of the caller if it's set to type(uint256).max.

However, `stkWALLET.transferFrom` decreases it regardless of it's value which may break general assumtions:

```
function transferFrom(address from, address to, uint amount) external returns (bool
    success) {
  ...
  allowed[from][msg.sender] = allowed[from][msg.sender] - amount;
  ...
}
```

Here's an example from OpenZeppelin's ERC20 implementation:

```
function _spendAllowance(address owner, address spender, uint256 value) internal
    virtual {
    uint256 currentAllowance = allowance(owner, spender);
    if (currentAllowance != type(uint256).max) {
        ...
    }
}
```

Recommendation: Consider implementing the above max allowance check.

Resolution: Pending.

7.2.2 Typographical mistakes and code style suggestions

Severity: *Informational*

Context: stkWALLET.sol

Description: The contract contains one or more typographical issue(s). In an effort to keep the report size reasonable, we enumerate these below:

1. Use uint256 instead of uint.
2. Make the `xWallet` variable `immutable` in order to save gas every time it's read (1x cold `SLOAD` in almost every function and several warm `SLOAD`s).
3. Add custom errors or revert messages in require-checks as well as implementing custom require checks for underflow scenarios in `transfer` and `transferFrom`.
4. Consider what the intended order in `transferFrom` is - whether the `allowed` or `shares` mapping should be updated first. Currently, the `shares[from]` is updated first, but if a custom error is added the proper way would be to first revert due to insufficient allowance rather than balance.
5. Move the constructor to the top of the contract before any function declarations.
6. A typo made on line 77 - `NTE` should be `NOTE`.

Recommendation: Consider implementing the aforementioned suggestions.

Resolution: Pending.