Smart Contract Audit Report

Date of Audit: Mar 12, 2025
Version: v1.1
Audited by: Shahaf Antwarg
Client: ETHGas
Repository: https://github.com/ethgas-developer/ethgas-contracts-core-for-audit
Contracts Reviewed: EthgasPool.sol, EthgasStaking.sol

Report Properties

Client	ETHGas
Version	v1.1
Contracts	EthgasPool, EthgasStaking
Author	Shahaf Antwarg
Auditors	Shahaf Antwarg
Classification	Confidential

Version Info

Version	Commit	Date	Author	Description
v1.1	d32f681	Mar 12 2025	Shahaf Antwarg	Final Release
v1.0	9a2f073	Feb 6 2025	Shahaf Antwarg	Release Candidate

Deployments

Contract	Chain	Address	Date
EthgasPool	Ethereum	0x818EF032D736B1a2ECC8556Fc1bC65aEBD8482c5	Mar 18, 2025

Table of Contents

1. Introduction

Following our request to review the source code of ETHGas's smart contracts, this report outlines our systematic approach to evaluating potential security issues, semantic inconsistencies between the design and implementation, and recommendations for improvements in both security and performance. Our review focused on the core contracts that handle staking, reward claims, and deposits. Overall, while the contracts exhibit many best practices, we identified several issues - ranging from incorrect comparison of times and reentrancy risks to redundant checks and minor code quality problems that require remediation before production deployment.

This audit focuses on the **EthgasPool** (responsible for pooling of tokens/ETH and applying daily withdrawal caps) and **EthgasStaking** (managing stake lockups and withdrawals). The ETHGas team requested a review of these contracts to detect security vulnerabilities, logical errors, and inefficiencies.

1.1 About ETHGas

ETHGas is a marketplace to source and trade blockspace commitments, alongside the Base Fee itself. It operates as a hybrid exchange - a centralized venue, where the central limit orderbook (CLOB) matches buyers with sellers, alongside a non-custodial smart contract where the collateral is held to backstop validator commitments. ETHGas is a neutral party offering end-to-end privacy.

1.2 About node.security

node.security Audits is a dedicated team specializing in smart contract security and blockchain risk management. Our team leverages a combination of automated tools and manual review to evaluate contracts thoroughly. We adhere to industry best practices and continuously update our methodologies based on emerging threats and vulnerabilities. For any queries or additional information, we can be reached via Telegram.

1.3 Disclaimer

This audit represents an independent security review of the smart contract(s) provided and is not a substitute for the complete functional testing that should be performed before any software release. Although every effort has been made to identify vulnerabilities, no audit can guarantee that all potential issues have been discovered. We strongly recommend additional independent audits and a public bug bounty program to further strengthen the security posture. This report is intended solely for security evaluation purposes and should not be interpreted as investment advice.

2. Report Structure

This audit report is organized to facilitate clear and efficient navigation from the most critical findings to less significant issues. Each issue is carefully documented with its severity rating and status, ensuring that readers can quickly assess the overall security posture and prioritize remediation efforts.

2.1 Issue tags

Issues are tagged as:

- **Resolved**: The issue has been fixed.
- **Unresolved**: The issue remains open.
- **Verified**: The functionality has been reviewed and confirmed with the client.

2.2 Severity levels

Severity levels are defined as follows:

- **Critical**: Issues that may lead to direct loss of funds or severe misallocation.
- **High**: Issues that significantly disrupt contract operation or pose a high risk of exploitation.
- **Medium**: Issues that affect the operation in non-catastrophic ways.
- **Low**: Minor issues that have minimal impact.
- **Info**: Observations that do not impact functionality but provide guidance for best practices.

2.3 Vulnerability Categories

Our audit followed a structured checklist covering a wide range of potential issues, from basic coding errors to advanced DeFi attack vectors. This approach ensures that every aspect of the code is thoroughly examined. The categories we evaluated include:

Basic Coding Bugs:
- Constructor and initializer mismatches
- Overflows, underflows, and unchecked arithmetic
- Short address or parameter attacks
- Uninitialized storage pointers

Semantic Consistency Checks:
- Consistency between the code and the design documentation
- Clarity of comments and correct parameter naming
- Detection of misleading or outdated code comments

Access Control & Authorization:
- Verification of role-based restrictions
- Proper management of privileged operations to prevent unauthorized access

Reentrancy & External Calls:
- Identification of reentrancy vulnerabilities
- Ensuring adherence to the checks-effects-interactions pattern

Business Logic & Financial Flow:
- Correct implementation of deposit, withdrawal, staking, and reward distribution
- Enforcement of lock periods, daily withdrawal caps, and minimum deposit amounts
- Ensuring consistency between internal state and actual token balances

Advanced DeFi Attack Vectors:
- Front-running, flash loan exploitation, and MEV risks
- Oracle manipulation and price feed vulnerabilities

Resource Management & Gas Optimization:
- Efficiency of loops and unbounded iterations
- Removal of redundant operations
- Appropriate use of compiler optimizations and internal function visibility

External Integration & Compatibility:
- Interaction with ERC20 tokens and potential issues with deflationary or rebasing tokens
- Proper handling of external contract calls and integration with critical services

Coding Standards & Documentation:
- Consistent naming conventions and parameter usage
- Correct and clear code comments without typos
- Adherence to best practices in coding style and organization

This comprehensive checklist enabled us to systematically identify and address vulnerabilities, ensuring a detailed and thorough audit of the contracts.

3. Scope of Work

Contracts Audited:

- **EthgasPool:** Receives deposits, enforces daily caps, includes administrative server-transfer features.
- **EthgasStaking:** Allows users to stake tokens, with forced lock periods and the ability to withdraw after expiry.

Focus Areas:

1. **Security Vulnerabilities** (reentrancy, access control flaws, logic inconsistencies).
2. **Lock & Withdrawal Logic** (especially `minLockPeriod` usage).
3. **Callback Vectors** (malicious tokens triggering reentrancy).
4. **Daily Cap Enforcement** & role-based restrictions.
5. **Gas Efficiency** & code maintainability.

4. Methodology

1. **Manual Code Review**: Line-by-line inspection of each contract, focusing on fund flows (deposit, withdraw), lock enforcement, and external calls.
2. **Automated Analysis**: Tools such as Slither, Mythril for static analysis and vulnerability detection (reentrancy patterns, uninitialized storage, etc.).
3. **Threat Modeling**: Considering ways an attacker could bypass daily caps, lock periods, or manipulate callbacks.
4. **Testing & Simulation**: Deployed in a test environment, tried scenarios for normal usage and malicious attempts.
5. **Reporting**: Consolidation of all findings by severity, with recommended fixes and references to the code.

5. Findings

Our audit identified several issues ranging from **high** to **low** severity. Of particular note are two **medium** severity issues and one **high** severity incorrect time comparison vulnerability in `EthgasStaking`. The contracts exhibit an otherwise strong design that aligns with recognized best practices (using role-based access control, SafeERC20, etc.). By fixing the highlighted issues, ETHGas can confidently deploy these contracts in production.

Below is a summary of the issues found. Each includes a **severity rating** (Critical, High, Medium, Low, Informational), title, potential exploit scenarios, and our **recommended remediation**.

ID	Severity	Title	Category	Status
H-01	**High**	Incorrect Lock Period Comparison	Business Logic, Time and State	Resolved
M-01	**Medium**	Reentrancy Risk in Deposit and Withdrawal Functions	Reentrancy & External Calls	Resolved
M-02	**Medium**	Uncontrolled ETH Acceptance via receive() Function	Security Features, Business Logic	Resolved
L-01	**Low**	Ambiguous Additional Lock Time Handling	Business Logic, Time and State	Verified
L-02	**Low**	Unrestricted Access to convertEthToWeth Function	Access Control & External Calls	Resolved
L-03	**Low**	Redundant Balance Checks	Basic Coding Bugs	Verified
L-04	**Low**	Error Typo	Code Quality	Resolved
L-05	**Low**	Daily Withdrawal Cap Logic	Business Logic & Financial Flow	Verified
L-06	**Low**	Missing Event Emission for Token Support Initialization	Semantic Consistency	Resolved
I-01	**Info**	Library Function Visibility	Resource Management	Verified
I-02	**Info**	Duplicate Deposit Logic	Code Quality	Verified
I-03	**Info**	Redundant Role Modifier Parameter Usage	Code Quality	Resolved
I-04	**Info**	Redundant Contract Casting in Role Modifiers	Code Quality	Resolved

H-01: Incorrect Lock Period Comparison

Severity: High
Category: Business Logic / Time and State

Description:
In the EthgasStaking contract's deposit functions, the new lock period is computed as:
`newLockPeriod = block.timestamp + _additionalLockPeriod + lockBlkRemaining`
and is then compared against `minLockBlkTime[_token]` using:
`if (newLockPeriod < minLockBlkTime[_token] && _additionalLockPeriod != 0) revert InvalidLockingPeriod();`

This comparison is flawed because `newLockPeriod` is an **absolute timestamp** (a point in time in the future), whereas `minLockBlkTime[_token]` represents a **duration** (the minimum lock period, typically in seconds). Comparing an absolute time with a duration is semantically incorrect. Instead, the contract should compare the intended lock duration (i.e. `_additionalLockPeriod + lockBlkRemaining`) with `minLockBlkTime[_token]`.

Reproduction Steps:

1. Deploy EthgasStaking with a specific minimum lock period (e.g., 100 seconds) for a token.
2. Perform a deposit where `block.timestamp` is, say, 1000, and the existing lock time yields a remaining lock period of 20 seconds.
3. Use an `_additionalLockPeriod` of 50 seconds.
4. The contract computes `newLockPeriod = 1000 + 50 + 20 = 1070`.
5. The condition checks if `1070 < 100` (if `minLockBlkTime[_token]` were 100 seconds), which is always false.
6. Instead, the intended check should ensure that the lock duration (`50 + 20 = 70` seconds) is not less than the minimum required lock duration (100 seconds).

Recommendation:

Refactor the check to compare durations rather than an absolute timestamp. For example, change the condition to:

```
uint256 newLockPeriod = _additionalLockPeriod + lockBlkRemaining;
if (newLockPeriod < minLockBlkTime[_token] && _additionalLockPeriod != 0) revert InvalidLockingPeriod();
unlockBlkTime[_token][_for] = block.timestamp + newLockPeriod;
```

- Update and expand test cases to ensure that deposits always enforce a lock period of at least `minLockBlkTime[_token]`.

M-01: Reentrancy Risk in Deposit and Withdrawal Functions

Severity: Medium
Category: Reentrancy & External Calls / Advanced DeFi Attack Vectors

Description:
Both the deposit and withdrawal functions in EthgasStaking (and similarly in EthgasPool) update internal state before performing token transfers using SafeERC20. Although this ordering follows the checks-effects-interactions pattern, it does not prevent reentrant calls, especially when tokens with callback functionality are involved. A token implementing a tokensReceived-like hook could trigger reentrant calls during these operations, potentially allowing an attacker to manipulate state (e.g., increment balances on deposit or deplete funds on withdrawal).

Reproduction Steps:

1. Deploy the target contract (e.g., EthgasStaking) using a token that implements a callback capable of re-entering the contract.
2. Call the deposit function with this token and observe whether the callback can re-enter the function, causing multiple balance increments.
3. Similarly, trigger the withdrawal function to check if reentrant calls can result in unexpected state modifications (e.g., multiple withdrawals).

Recommendation:

- Add the `nonReentrant` modifier to both deposit and withdrawal functions to prevent reentrant calls.
- Ensure that critical state changes are fully applied before external token transfers are made.

M-02: Uncontrolled ETH Acceptance via receive() Function

Severity: Medium
Category: Security Features / Business Logic

Description:
The contract includes a default `receive() external payable {}` function that accepts ETH directly. Although the **deposit function controls ETH deposits** by **requiring WETH to be explicitly added** to the supported tokens array, the fallback function remains active and accepts any ETH transfers sent directly to the contract. This creates a potential security concern because it **bypasses the controlled deposit logic**. Unsolicited ETH transfers may lead to unexpected balances that are not processed (i.e., not wrapped into WETH) or accounted for within the contract's internal state. An attacker could exploit this behavior to trigger unforeseen side effects or manipulate contract operations.

Reproduction Steps:

1. Deploy the contract with the current `receive()` function in place.
2. Send ETH directly to the contract address (without invoking the deposit function).
3. Observe that the contract accepts the ETH, increasing its balance, even though WETH was not explicitly added to the supportedToken map.

Recommendation:

- **If direct ETH transfers are unintended:**
  - Modify or remove the `receive()` function so that any ETH sent directly reverts.
- **If direct ETH transfers should be handled:**
  - Implement logic within the `receive()` function to check whether WETH is a supported token.
- Ensure comprehensive testing of the fallback function to verify that ETH is either rejected or processed as intended, preventing discrepancies in internal fund accounting.

L-01: Ambiguous Additional Lock Time Handling

Severity: Low
Category: Business Logic / Time and State

Description:
The EthgasStaking contract currently supports depositing additional funds with an `_additionalLockPeriod` value of 0 to allow for more flexible staking. The contract ends up allowing only two cases: either an additional lock time of 0 or a value that is strictly greater than the minimum required lock period. This ambiguity means that if additional staking with 0 lock time is intended to provide flexibility, the contract should instead allow **any** `_additionalLockPeriod` value without imposing an unexpected constraint. Given that H-01 highlights a flaw in comparing an absolute future timestamp with a duration, we recommend revisiting this condition. Even if the timestamp-versus-duration issue (H-01) is fixed, it is still unexpected behavior to allow an additional staking deposit with either 0 or only the minimum lock time. This ambiguity could lead to inconsistent staking periods and potentially enable premature withdrawals.

Reproduction Steps:

1. Deploy EthgasStaking with a defined minimum lock period (`minLockBlkTime`) for a supported token.
2. Make an initial deposit with an `_additionalLockPeriod` that meets or exceeds the minimum, establishing a valid lock.
3. Before the lock expires, deposit additional tokens using various `_additionalLockPeriod` values, including 0 and values below the minimum.
4. Observe that the contract only accepts 0 or values greater than the minimum, leading to ambiguous behavior.

Recommendation:

- Revisit the conditional logic that restricts the `_additionalLockPeriod` to either 0 or values greater than the minimum.
- Redesign the mechanism to either:
    - Enforce that any additional deposit always extends the lock duration so that the overall lock is at least `minLockBlkTime[_token]`, or
    - Allow any `_additionalLockPeriod` value without implicitly enforcing a minimum, making the behavior explicit and predictable.
- Update test cases to cover a range of `_additionalLockPeriod` inputs and ensure the resulting lock duration meets the intended protocol behavior.

L-02: Unrestricted Access to convertEthToWeth Function

Severity: Low
Category: Access Control & External Calls / Business Logic

Description:
The EthgasPool contract includes a `convertEthToWeth()` function, which converts the contract's ETH balance into WETH by calling `IWETH(weth).deposit{value: address(this).balance}()`. However, this function lacks access restrictions, meaning that any external account can trigger the conversion at any time. While the conversion itself does not redirect funds to an unauthorized party, it may interfere with the contract's intended fund management process and cause unexpected behavior in the overall system.

Reproduction Steps:

1. Deploy the EthgasPool contract with the current implementation.
2. Send ETH directly to the contract (via the `receive()` function).
3. Call `convertEthToWeth()` from any account (even one that is not privileged).
4. Observe that the entire ETH balance is converted to WETH, regardless of whether this operation was intended by the protocol design.

Potential Impact:

- Unauthorized triggering of ETH-to-WETH conversion could disrupt planned deposit workflows or accounting, especially if specific ordering or timing is critical.
- Although the function does not transfer funds to the caller, the lack of restrictions may expose the contract to unintended interactions.

Recommendation:

- Restrict access to `convertEthToWeth()` so that only an authorized role (e.g., ADMIN_ROLE or TREASURER_ROLE) can invoke it.
- Alternatively, document and clarify that this function is intended to be public and ensure that its behavior is consistent with the overall design of the fund management process.
- Consider adding a modifier to the function to enforce proper access control if unintended usage is a concern.

L-03: Redundant Balance Checks

Severity: Low
Category: Basic Coding Bugs

Description:
Some deposit functions include explicit balance checks before calling SafeERC20's `safeTransferFrom`. Since SafeERC20 automatically reverts on insufficient balance or allowance, these checks are redundant and add unnecessary code complexity.

Reproduction Steps:

1. Review the deposit function and identify the explicit balance checks.
2. Verify that `safeTransferFrom` reverts under the same conditions.

Recommendation:

- Remove redundant balance checks to simplify the code and reduce gas consumption.

L-04: Error Typo

Severity: Low
Category: Code Quality / Semantic Consistency

Description:
There is a typographical error in the error message ("**MinimalStakingAmounttNotMet**" contains an extra "t"). Although this does not affect functionality, it reduces code clarity and professionalism.

Reproduction Steps:

1. Trigger an error condition that emits the problematic message.
2. Observe the typo in the emitted error message.

Recommendation:

- Correct typographical errors to ensure consistent, professional messaging in the codebase.

L-05: Daily Withdrawal Cap Logic

Severity: Low
Category: Business Logic & Financial Flow

Description:
The daily withdrawal cap mechanism resets the withdrawal counter based on a 24-hour inactivity window. As a result, if withdrawals occur frequently, the cap may not reset as expected, leading to a cumulative withdrawal amount that exceeds the intended limit over a given day.

Reproduction Steps:

1. Execute multiple withdrawals within intervals shorter than 24 hours.
2. Observe that the `currentDailyWithdrawalAmount` does not reset until 24 hours of inactivity occur.
3. Compare the total withdrawn against the expected daily cap.

Recommendation:

- Document the sliding window behavior clearly.
- If a strict daily cap is desired, modify the logic to reset the counter at a fixed time (e.g., midnight UTC).

L-06: Missing Event Emission for Token Support Initialization

Severity: Low
Category: Semantic Consistency / Code Quality

Description:
In the EthgasStaking constructor, the supported tokens are initialized by updating the `supportedToken` mapping for each token in the `_tokensAllowed` array. However, no events are emitted during this initialization. In contrast, the `setStakable` function emits an event when a token's stakability is changed. The lack of event emissions during deployment means that off-chain monitoring or auditing tools have no record of which tokens were initially allowed, potentially reducing transparency and traceability in the contract's initialization process.

Reproduction Steps:

1. Deploy the EthgasStaking contract using the constructor with a list of allowed tokens.
2. Observe that no events (such as a `TokenStakabilityChanged` event) are emitted during the constructor execution, even though tokens are being marked as supported.
3. Compare this behavior to the `setStakable` function, which does emit an event when token support is updated.

Recommendation:

- Modify the constructor to emit an event for each token added to the `supportedToken` mapping. Alternatively, consider calling the `setStakable` function during initialization to log each token's support status.
- This improvement will enhance transparency and facilitate easier off-chain tracking of the contract's initial state.

I-01: Library Function Visibility

Severity: Info
Category: Resource Management / Gas Optimization

Description:
In the TransferFundHelper library, some functions are declared as `external` despite being used solely within the EthgasPool contract. This choice increases gas costs due to the overhead of external calls and unnecessarily exposes the functions.

Reproduction Steps:

1. Analyze the usage of TransferFundHelper functions within EthgasPool.
2. Compare gas costs for external versus internal calls.

Recommendation:

- Change the visibility of these functions to `internal` if they are not intended to be called by other contracts.
- This will reduce gas usage and improve encapsulation.
- Check for DepositHelper as well.

I-02: Duplicate Deposit Logic

Severity: Info
Category: Code Quality / Maintainability

Description:
In the EthgasStaking contract, similar logic is duplicated across multiple deposit functions (`depositFor` and `depositETHFor`). Both functions perform nearly identical steps - validating inputs, updating balances, calculating lock periods, and emitting deposit events - before executing token-specific transfer operations. This duplication increases maintenance overhead and the risk of inconsistencies if one part is updated while the other is not.

Reproduction Steps:

1. Review the source code for `depositFor` and `depositETHFor`.
2. Notice that both functions execute similar input validations, state updates (e.g., updating `balance` and `unlockBlkTime`), and event emissions.
3. Compare these sections to identify duplicated logic that could be consolidated.

Recommendation:

- Refactor the common deposit logic into a single internal function (e.g., `_deposit`) that handles validation, state updates, and event emission.
- Modify `depositFor` and `depositETHFor` to call this helper function, followed by their token-specific transfer operations (using `safeTransferFrom` for tokens and wrapping ETH for WETH).
- This approach will reduce code duplication, simplify future updates, and ensure consistency across deposit functions.

```
function _deposit(address _token, address _for, uint256 _amount, uint256
_additionalLockPeriod) internal {
        // validation
        // update balance
        emit Deposit(++eventId, _for, _token, _amount, _additionalLockPeriod);
    }

    function depositFor(address _token, address _for, uint256 _amount, uint256
_additionalLockPeriod) whenNotPaused external {
        _deposit(_token, _for, _amount, _additionalLockPeriod);
        IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
    }

    function depositETHFor(address _for, uint256 _additionalLockPeriod)
whenNotPaused payable external {
        _deposit(WETH_ADDRESS, _for, msg.value, _additionalLockPeriod);
        IWETH(WETH_ADDRESS).deposit{value:msg.value}();
    }
```

I-03: Redundant Role Modifier Parameter Usage

Severity: Info
Category: Code Quality / Best Practices

Description:
Currently, functions in the contracts use role modifiers that require passing `msg.sender` explicitly (e.g., `onlyAdminRole(msg.sender)`). Since Solidity modifiers have direct access to `msg.sender`, this extra parameter is redundant and can be simplified. This change will improve code clarity and reduce potential errors, while also aligning with best practices for Solidity development.

Example of Current Implementation:

```
modifier onlyAdminRole(address _account) {
   IACLManager(aclManager).checkAdminRole(_account);
   _;
}

function setStakable(address _token, bool _canStake) external onlyAdminRole(msg.sender) {
   // function logic...
}
```

Recommended Change:
Define the modifier without an explicit parameter, and reference `msg.sender` internally. Then update the function signature accordingly.

Example of Recommended Implementation:

```
modifier onlyAdmin() {
   IACLManager(aclManager).checkAdminRole(msg.sender);
   _;
}

function setStakable(address _token, bool _canStake) external onlyAdmin {
   // function logic...
}
```

Recommendation:

- Refactor role modifiers (e.g., `onlyAdminRole`, `onlyTimelockRole`, etc.) to remove the redundant `msg.sender` parameter.
- Update all function declarations to use the new modifier names (e.g., `onlyAdmin`) without passing any arguments.
- This refactoring will streamline the code and improve readability without impacting security.

I-04: Redundant Contract Casting in Role Modifiers

Severity: Info
Category: Code Quality / Best Practices

Description:
In the current implementation, role modifiers invoke role-checking functions using explicit casting (e.g., `IACLManager(aclManager).checkAdminRole(_account)`). Given that `aclManager` is already declared with the type `IACLManager`, the explicit cast is redundant. The code can be simplified by directly calling `aclManager.checkAdminRole(_account)`.

Recommended Change:
Remove the explicit cast, and, as discussed in a related issue, also refactor the modifier to avoid passing `msg.sender` explicitly. The modifier can directly reference `msg.sender` without casting, as shown below:

```
modifier onlyAdmin() {
   aclManager.checkAdminRole(msg.sender);
   _;
}

function setStakable(address _token, bool _canStake) external onlyAdmin {
   // function logic...
}
```

Recommendation:

- Remove redundant casts by calling `aclManager.checkAdminRole(...)` directly.
- Combine this change with refactoring the role modifiers to eliminate the need to pass `msg.sender` as a parameter.
- This improves code clarity and consistency without affecting functionality or security.

6. Final Recommendations

Summary

1. **Fix the Lock Period Comparison**: Refactor the check to compare durations rather than an absolute timestamp.
2. **Add Reentrancy Guards**: Protect flows potential reentry.
3. **Align Daily Caps with Expected Behavior**: Decide if a sliding or strict daily approach is intended, then update logic or docs accordingly.
4. **Minor Validation**: code comments, minor naming corrections, etc.

Post-Remediation Testing

We recommend thoroughly **testing** the updated code to confirm that changes are effective and do not introduce new bugs:

- Confirm you cannot add additional lock time lower than the minimum defined.
- Attempt malicious reentry with a custom callback token during deposit.
- Simulate repeated withdrawals near the daily cap boundary to ensure the logic meets expectations.

7. Conclusion

The EthgasPool and EthgasStaking contracts demonstrate an overall secure design, leveraging role-based access control, SafeERC20, and best practices for external calls.

Once issues are addressed, the contracts will be well-positioned for secure deployment. We commend the ETHGas team for proactively seeking this audit and recommend a post-fix verification step to ensure completeness. Should further questions arise, or if the contracts evolve significantly in the future, we advise a subsequent audit to maintain robust security standards over time.