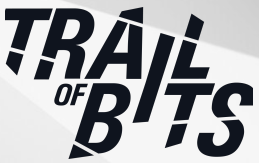

Otim Smart Wallet

Security Assessment

March 25, 2025

Prepared for:
Dave Carroll and Julian Rachman
Otim

Prepared by: **Quan Nguyen and Omar Inuwa**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Blockchain & Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Quan Nguyen, Consultant
quan.nguyen@trailofbits.com

Omar Inuwa, Consultant
omar.inuwa@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
February 27, 2025	Pre-project kickoff call
March 10, 2025	Status update meeting #1
March 13, 2025	Delivery of report draft
March 13, 2025	Report readout meeting
March 25, 2025	Delivery of final comprehensive report

Executive Summary

Engagement Overview

Otim engaged Trail of Bits to review the security of the Otim smart wallet project. Otim leverages EIP-7702 to transform externally owned accounts (EOAs) into smart wallets. It utilizes a delegate contract, allowing EOAs to delegate calls to it. The delegate contract, in turn, supports whitelisted action contracts, enabling controlled execution of predefined actions.

A team of two consultants conducted the review from March 3 to March 13, 2025, for a total of three engineer-weeks of effort. Our testing efforts focused on Otim's delegate, core, infrastructure action, and library smart contracts. With full access to source code and documentation, we performed static and dynamic testing of Otim's codebase, using automated and manual processes.

Observations and Impact

The Otim smart wallet contracts are well structured, modular, and easy to follow. During the review, we identified no vulnerabilities that could put user funds at risk. The delegate contract securely verifies signatures, and we found no reentrancy issues. However, the protocol has issues related to token integration. Specifically, the implementation lacks ERC-721 and ERC-1155 callbacks, which hinders proper NFT handling (TOB-OTIM-1). Additionally, it does not properly handle ERC-20 tokens like USDT that do not return a Boolean on transfer (TOB-OTIM-2). While these issues do not jeopardize user funds, they result in a lack of functionality within the smart wallet.

Furthermore, although user funds are secure, we found several vulnerabilities that could cause protocol owners to lose funds. For instance, the `Treasury` contract lacks a withdrawal function for ERC-20 tokens, so any such tokens sent to the contract would be locked within it (TOB-OTIM-4). Additionally, the Otim executors are susceptible to gas griefing; attackers could front run transactions to cause executions to revert, wasting gas (TOB-OTIM-8, TOB-OTIM-9). Gas calculation–related issues could also cause users to pay more when the network is congested (TOB-OTIM-6) and cause users to be undercharged when the executor pays for the priority fee (TOB-OTIM-5). These issues could lead to inconsistent transaction execution, potentially resulting in failed transactions or delays, further impacting the user experience.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Otim take the following steps:

- **Remediate the findings disclosed in this report.** Specifically, address the front-running and token-integration issues to ensure compatibility with a broad

range of token standards and seamless composability within the wider DeFi ecosystem. This includes implementing proper handling for ERC-721 and ERC-1155 callbacks, supporting ERC-20 tokens that do not return a Boolean on transfer, and adding a withdrawal function to the `Treasury` contract. Additionally, introduce safeguards to protect Otim executors from gas griefing attacks, minimizing the risk of front-running and wasted gas on reverted transactions.

- **Add more testing.** Expand the test suite to account for diverse token types, including those with unique behaviors such as double-entry bookkeeping, lack of Boolean return values on transfers, and non-reverting failure states. Account for such token types in the test suite when creating new action contracts. Testing should also cover NFTs, ERC-1155 tokens, and any other standards intended for integration to improve the protocol's robustness and interoperability.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	6
Low	0
Informational	2
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	2
Denial of Service	3
Timing	3
Undefined Behavior	1

Project Goals

The engagement was scoped to provide a security assessment of the Otim smart wallet implementation. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can an attacker manipulate the instruction storage data of other users to alter information without requiring signatures?
- Does the system correctly calculate fees?
- Is it possible to front run transactions and negatively affect the system?
- Does the system handle non-ERC-20 standard tokens correctly?
- Do the smart contracts conform to the targeted standards, such as EIP-7702?
- Are there any reentrancy risks with ERC-777 transfer hooks or other contract calls?
- Are signatures correctly verified?
- Do the smart contracts properly implement the intended design features?
- Does the codebase conform to industry best practices?

Project Targets

The engagement involved reviewing and testing the following target.

otim-protocol

Repository	github.com/otimlabs/otim-protocol
Version	`a683e0fa9cd59043a2b90e0946324d2bc421ce9f`
Type	Solidity
Platform	EVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Delegate contract:** We assessed the `OtimDelegate` contract, which serves as the core delegate contract for the Otim protocol, leveraging EIP-7702 to turn EOAs into smart wallets. Our review focused on validating the robustness of the signature validation mechanisms to prevent replay attacks. Additionally, we examined the contract's capability to handle a wide range of token types, including native ETH, ERC-20 tokens, and NFTs.

- **Action contracts:** We evaluated the security and functionality of the action contracts, which the `OtimDelegate` contract whitelists and delegates call to. Our analysis covered specific actions such as ERC-20 transfers and automatic refueling of balances when holdings drop below a threshold. We checked the gas calculations and payments, assessed the flexibility of the system in supporting a broad range of ERC-20 tokens, and looked for potential vulnerabilities, including susceptibility to DDoS attacks.

- **Fee calculations and `Treasury` contract:** We reviewed the `FeeTokenRegistry` and `OtimFee` contracts, which handle fee calculations and payments. We analyzed the fee structure for different transaction types to confirm that fees are correctly deducted. Additionally, we reviewed the `Treasury` contract to ensure proper collection and management of fees.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- While we evaluated the use of the EIP-7702 delegate contract (smart wallet) as part of the Otim protocol, we did not fully assess the implications of using the wallet as a general-purpose wallet for interacting with arbitrary contracts. Specific use cases will require ad hoc integration reviews.

- The Otim team indicated that additional action contracts are currently under development. These contracts are outside the scope of this audit and will require integration reviews upon completion.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The arithmetic in the contracts is simple and easy to understand. The codebase uses Solidity 0.8.26, which provides built-in overflow and underflow protection for all arithmetic operations.	**Strong**
Auditing	The `OtimDelegate` contract actively emits events whenever an instruction is executed or deactivated, ensuring real-time tracking of state changes. Similarly, the `ActionManager` contract emits events for every critical operation, such as adding or removing actions, providing transparency and traceability for key actions within the system.	**Satisfactory**
Authentication / Access Controls	The `ActionManager` contract uses a two-tier control system: `DEFAULT_ADMIN_ROLE` for routine tasks and `KILL_SWITCH_ROLE` for emergencies. Actions require EIP-712 cryptographic verification to ensure authorization. The `fromDelegateCodeOnly` modifier in `InstructionStorage` restricts the ability to make storage modifications to the `OtimDelegate` contract.	**Satisfactory**
Complexity Management	Core functions have clear separation of concerns, with each handling a specific responsibility. The `OtimDelegate` contract manages instruction execution and deactivation, `ActionManager` controls action availability, and specialized contracts (`Transfer`, `Refuel`) handle specific tasks. NatSpec documentation is provided for each function, explaining its purpose, parameters, and security considerations, ensuring the code is accessible and maintainable.	**Satisfactory**
Decentralization	The system utilizes a centralized control model with	**Moderate**

	admin-managed actions and fees. Key administrative functions, such as action registration and fee management, are governed by privileged roles. The `Treasury` contract centralizes fee collection, and a kill switch mechanism enables admins to halt all actions in emergencies. While admins control the registration of actions, users retain full control over which actions they choose to use.	
Documentation	The codebase includes comprehensive NatSpec documentation, providing detailed explanations of complex logic. Core contracts and actions are thoroughly described, and their implementations align consistently with the specifications.	**Satisfactory**
Low-Level Manipulation	The codebase strategically uses assembly for gas optimization in storage operations, with each complex assembly implementation paired with a high-level reference implementation. Differential testing validates that low-level code matches its high-level counterpart, while thorough documentation explains all assembly operations, ensuring both efficiency and maintainability.	**Strong**
Testing and Verification	The codebase is thoroughly tested, ensuring robust vulnerability detection. Differential testing is employed to verify that low-level code aligns with its high-level counterpart. However, the test suite fails to cover a variety of tokens, leading to issues such as locked funds (TOB-OTIM-4) and the inability to handle certain tokens (TOB-OTIM-1, TOB-OTIM-2).	**Moderate**
Transaction Ordering	The codebase contains significant vulnerabilities related to transaction ordering. It lacks protection against gas griefing attacks involving front-running (TOB-OTIM-8, TOB-OTIM-9). Users cannot specify maximum gas prices for fee calculations, potentially exposing them to unexpectedly high costs during network congestion (TOB-OTIM-6).	**Weak**

Summary of Findings

The table below summarizes the findings of the review, including details on type and severity.

ID	Title	Type	Severity
1	Lack of NFT callbacks	Denial of Service	**Medium**
2	Protocol does not handle tokens that do not return a Boolean	Denial of Service	**Medium**
3	Lack of validation of latestRoundData return value	Data Validation	**Medium**
4	ERC-20 tokens cannot be withdrawn from Treasury contract	Denial of Service	**High**
5	Gas price calculation in OtimFee contract does not include priority fee	Undefined Behavior	**Informational**
6	Lack of maximum gas price protection in OtimFee contract	Timing	**Medium**
7	Edge case when balance is at threshold	Data Validation	**Informational**
8	Gas griefing vulnerability in native token transfer	Timing	**Medium**
9	Delegate contract can be replaced to carry out gas griefing attack	Timing	**Medium**

Detailed Findings

1. Lack of NFT callbacks	
Severity: **Medium**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-OTIM-1
Target: `OtimDelegate.sol`	

Description
In the current implementation, when an EOA upgrades to a smart wallet, it will no longer be able to receive NFTs due to a lack of token callbacks in the current implementation.

Per the ERC-721 and ERC-1155 standards, when an NFT is transferred using `safeTransfer`, the method checks if the receiving address is an EOA or a smart contract. For the transfer to succeed, the receiver must be an EOA or a smart contract that implements `IERC721Receiver.onERC721Received`. The EIP-7702 standard transforms an EOA into a smart contract, so `IERC721Receiver.onERC721Received` must be implemented in the smart wallet implementation; otherwise, the user will not be able to continue to receive NFTs.

Exploit Scenario
A user attempts to purchase an NFT from a marketplace, but their transaction will always revert due to a lack of token callbacks in their smart wallet implementation.

Recommendations
Short term, add support for ERC-721 and ERC-1155 in the delegate contract; add support directly, or make a receiver contract that the delegate contract will inherit from.

Long term, identify other standards the Otim protocol is planned to be compatible with and test how they behave with Otim smart wallets.

2. Protocol does not handle tokens that do not return a Boolean	
Severity: **Medium**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-OTIM-2
Target: `actions/TransferERC20Action.sol`, `actions/RefuelERC20Action.sol`	

Description

The `TransferERC20Action` and `RefuelERC20Action` contracts use a strict return-value check for token transfers; however, many popular tokens like USDT do not return any value on a successful transfer, so the contracts will wrongly treat these transfers as failures.

The lack of a return value is interpreted as false in Solidity. So when a transfer is made in the `TransferERC20Action` or `RefuelERC20Action` contract using a token like USDT, even if the transfer is successful, the call will revert due to this logic:

```
 86    // transfer the refuel amount to the target
 87    bool success = IERC20(refuelERC20.token).transfer(refuelERC20.target,
refuelAmount);
 88
 89    // if the transfer fails, revert
 90    if (!success) {
 91        revert TokenTransferFailed();
 92    }
```

Figure 2.1: If there is no return value, success defaults to false in `TransferERC20Action.sol` and `RefuelERC20Action.sol`.

Exploit Scenario

Transfers involving non-returning tokens (e.g., USDT) cause the `TransferERC20Action` or `RefuelERC20Action` contract to revert, so these tokens are essentially unusable within the protocol. The protocol's incompatibility with high-volume tokens severely limits user adoption. Overall, the protocol is disadvantaged by missing out on significant liquidity and user base.

Recommendations

Short term, use libraries that implement the `safeTransfer` function such as OpenZeppelin's library, which handle the edge case of tokens not returning a Boolean.

Long term, go through the token integration checklist, and ensure that all edge cases are covered.

3. Lack of validation of latestRoundData return value	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-OTIM-3
Target: `infrastructure/FeeTokenRegistry.sol`	

Description

The `weiToToken` function in the `FeeTokenRegistry` contract uses the `latestRoundData` function without validating whether the returned data is stale, potentially leading to the usage of outdated prices.

Specifically, the function does not check the value of `updatedAt`, creating a vulnerability to stale prices, which might jeopardize gas payments. According to Chainlink's documentation, a timestamp of `0` means the round is not complete and should not be used. Without this check, the `weiToToken` function could receive incorrect price data. Furthermore, the function does not check for an invalid `roundId`, which is `0`.

```
 72    // slither-disable-next-line unused-return
 73    (, int256 latestPrice,,,) =
AggregatorV3Interface(data.priceFeed).latestRoundData();
 74
 75    // if the latest price is zero or negative, revert
 76    if (latestPrice <= 0) {
 77        revert InvalidPrice();
 78    }
```

Figure 3.1: Lack of checks when receiving price data in `FeeTokenRegistry.sol`

Exploit Scenario

A user has to unduly pay more gas for a transaction because the price feed returns a stale price above the current market value. Alternatively, a user unduly pays less money to the transaction executor because the price feed returns a stale price below the current market value.

Recommendations

Short term, add checks to `weiToToken` to ensure that `updatedAt != 0`, that the feed data is still within an acceptable threshold (i.e., not stale), and that the current `roundId` is not `0`.

Long term, implement for all third-party price feeds fallback mechanisms, robust monitoring, and graceful handling of failures and returned stale data.

4. ERC-20 tokens cannot be withdrawn from Treasury contract

Severity: **High**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-OTIM-4
Target: `infrastructure/Treasury.sol`	

Description

The `Treasury` contract currently lacks functionality for withdrawing ERC-20 tokens; any such tokens sent to the treasury would be permanently locked in the contract.

The Otim protocol allows users to pay fees with either native ETH or any ERC-20 token registered by Otim. These fees are directed to the `Treasury` contract.

```
18    function withdraw(address target, uint256 value) external onlyOwner {
19        // check the target is not the zero address
20        if (target == address(0)) {
21            revert InvalidTarget();
22        }
23
24        // check the contract has enough balance to withdraw
25        if (value > address(this).balance) {
26            revert InsufficientBalance();
27        }
28
29        // withdraw the funds
30        (bool success, bytes memory result) = target.call{value: value}("");
31
32        // check if the withdrawal was successful
33        if (!success) {
34            revert WithdrawalFailed(result);
35        }
36    }
```

Figure 4.1: The `withdraw` function in `Treasury.sol`

However, the `Treasury` contract currently supports only ETH withdrawals via the `withdraw` function; it lacks a function for withdrawing ERC-20 tokens.

Exploit Scenario

The Otim protocol accepts USDC as a fee token; these fees accumulate in the `Treasury` contract over time. When the protocol team attempts to access these funds, they discover that there is no mechanism to withdraw ERC-20 tokens from the treasury. The USDC becomes permanently locked in the contract, resulting in financial losses for the protocol.

Recommendations

Short term, add a function to withdraw ERC-20 tokens from the `Treasury` contract.

Long term, develop test cases specifically for token recovery scenarios. In addition, review how funds flow between users and components and document these flows comprehensively to improve system security.

5. Gas price calculation in OtimFee contract does not include priority fee

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-OTIM-5
Target: `actions/fee-models/OtimFee.sol`	

Description
The `OtimFee` contract incorrectly uses `block.basefee` instead of `tx.gasprice` for calculating gas costs, potentially leading to incorrect fee calculations.

The `chargeFee` function calculates the gas cost using `block.basefee`, which represents the base fee per gas in the current block under EIP-1559.

```
38    function chargeFee(uint256 gasUsed, Fee memory fee) public override {
39        // fee.value == 0 is a magic value signifying a sponsored Instruction
40        if (fee.value == 0) {
41            return;
42        }
43
44        // calculate the cost of the gas used in wei
45        uint256 weiGasCost = (gasUsed + gasConstant) * block.basefee;
```

Figure 5.1: Excerpt of the `chargeFee` function in `OtimFee.sol`

This value does not account for the priority fee (tip) included in transactions, so the calculation will not reflect the actual gas cost if the priority fee is paid. The actual cost paid by callers is determined by `tx.gasprice`, which includes both the base fee and priority fee.

Exploit Scenario
During a period of network congestion, an Otim executor submits a transaction with a high priority fee to ensure quick execution. While the Otim executor pays 200 gwei total (the 150 gwei base fee plus the 50 gwei priority fee), the `OtimFee` contract calculates costs using only the 150 gwei base fee. For a transaction using 100,000 gas, this results in the protocol undercharging by 0.005 ETH (50 gwei multiplied by 100,000).

As this pattern repeats across thousands of transactions, especially during high-activity periods when priority fees are substantial, the protocol accumulates significant financial losses that could have been captured as revenue.

Recommendations
Short term, replace `block.basefee` with `tx.gasprice` in the gas cost calculation.

Long term, implement comprehensive testing for various gas price scenarios to ensure the fee calculation remains accurate across different network conditions.

6. Lack of maximum gas price protection in OtimFee contract

Severity: **Medium**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-OTIM-6
Target: `actions/fee-models/OtimFee.sol`	

Description

The `OtimFee` contract does not allow users to specify a maximum gas price, exposing them to potentially unlimited fee costs during network congestion.

When calculating fees, the contract directly uses the current base gas price (`block.basefee`) without any upper bound protection.

```
38    function chargeFee(uint256 gasUsed, Fee memory fee) public override {
39        // fee.value == 0 is a magic value signifying a sponsored Instruction
40        if (fee.value == 0) {
41            return;
42        }
43
44        // calculate the cost of the gas used in wei
45        uint256 weiGasCost = (gasUsed + gasConstant) * block.basefee;
```

Figure 6.1: Excerpt of the `chargeFee` function in `OtimFee.sol`

During periods of high network congestion, gas prices can spike dramatically, potentially resulting in unexpectedly high fees for users. Without the ability to specify a maximum gas price they are willing to pay, users have no protection against these spikes and may end up paying significantly more than anticipated.

Exploit Scenario

Alice signs an instruction expecting to pay a reasonable fee based on recent gas prices. However, moments later, a popular NFT drop causes gas prices to spike tenfold. Without a maximum gas price limit, Alice's instruction processes at the elevated rate, charging her $50 in fees instead of the expected $5. This creates a poor user experience and discourages Alice and other users affected by this spike from interacting with the protocol during volatile gas market conditions.

Recommendations

Short term, modify the `Fee` struct and `chargeFee` function to include a maximum gas price parameter.

Long term, implement comprehensive testing for various gas price scenarios to ensure the fee calculation remains accurate across different network conditions.

7. Edge case when balance is at threshold

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-OTIM-7
Target: `actions/RefuelERC20Action.sol`, `actions/RefuelAction.sol`	

Description

In the `RefuelERC20Action` contract, when a user's balance reaches a predefined threshold, a transaction is made on their behalf to refuel their balance. In the current implementation, the function that refuels a user's account is intended to revert only if their balance exceeds the threshold. However, because the condition uses the `>=` operator instead of the `>` operator when validating whether the balance is above the threshold, it also reverts when the balance exactly equals the threshold. For example, if a user's threshold is `0`, refueling is blocked because the balance check fails when the balance is `0`, and the balance can never fall below `0`.

```
63    // if the balance is above the threshold, revert
64    if (balance >= refuel.threshold) {
65        revert BalanceOverThreshold();
66    }
```

Figure 7.1: Inaccurate condition in `RefuelERC20Action.sol` and `actions/RefuelAction.sol`

Recommendations

Short term, disallow `threshold = 0` explicitly to prevent this edge case that would block refueling.

Long term, assess threshold logic to ensure it aligns with the intended user experience.

8. Gas griefing vulnerability in native token transfer

Severity: **Medium**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-OTIM-8
Target: `actions/RefuelAction.sol`, `actions/TransferAction.sol`	

Description

The `RefuelAction` and `TransferAction` contracts are vulnerable to gas griefing attacks due to the way they handle native token transfers to the target address, which could lead to financial losses for Otim executors.

When native tokens are transferred to target addresses, these contracts use low-level calls without gas limitations, and fees are charged only upon successful execution.

```
 80    (bool success, bytes memory result) = refuel.target.call{value:
refuelAmount}("");
 81    // slither-disable-end reentrancy-events
 82    // slither-disable-end missing-zero-check
 83    // slither-disable-end arbitrary-send-eth
 84
 85    // if the transfer fails, revert
 86    if (!success) {
 87        revert NativeTransferFailed(result);
 88    }
 89
 90    // charge the fee
 91    chargeFee(startGas - gasleft(), refuel.fee);
```

Figure 8.1: Excerpt of the `execute` function in `RefuelAction.sol`

Since the fee is charged only after a successful call, a malicious target can front run transfer transactions with malicious ones that consume gas and revert, preventing the executor from receiving compensation for their expended resources.

Exploit Scenario

Alice signs an instruction to transfer 1 ETH to Bob's wallet (an EOA) every week. Bob monitors the mempool to see when the Otim executor broadcasts a transaction to execute Alice's instruction. He front runs this transaction with an EIP-7702 transaction that transforms his EOA into a contract by pointing to a malicious delegate contract. This contract implements a `receive` function that executes an unbounded loop consuming all available gas before reverting. When the Otim executor's transaction executes immediately after, it attempts to transfer ETH to what is now Bob's contract. The transaction reverts after consuming significant gas, but before reaching the fee charging logic. As a result, the

executor bears the full gas cost of the failed transaction without receiving any compensation, resulting in financial loss.

Recommendations

Short term, implement a gas limit for external calls to prevent complex executions in recipient contracts and protect against gas griefing attacks.

Long term, redesign the fee mechanism to charge users regardless of transfer success, potentially by implementing a two-phase execution process where fees are charged before attempting risky external calls, or by implementing a prepayment system for gas costs.

9. Delegate contract can be replaced to carry out gas griefing attack	
Severity: **Medium**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-OTIM-9
Target: `actions/OtimDelegate.sol`	

Description

The `executeInstruction` function in the `OtimDelegate` contract is vulnerable to a gas griefing attack through delegate contract replacement. Malicious users could front run legitimate executor transactions with a transaction that replaces the delegate contract, causing the executor's transaction to revert and the executor to waste gas without receiving compensation.

In the current implementation, Otim executors pay gas fees up front when calling `executeInstruction`, expecting fee payment from the user's balance after successful execution.

However, a malicious user can monitor the mempool for pending `executeInstruction` transactions and front run them with a transaction that replaces the delegate contract implementation. When the original transaction executes, it will fail or behave unexpectedly, causing the executor to spend gas without receiving the expected fee payment.

Exploit Scenario

A legitimate executor submits a transaction to execute an instruction. An attacker monitors the mempool and identifies this transaction. The attacker front runs the transaction and replaces the delegate contract. The executor's transaction executes against the modified delegate contract. The transaction fails or executes incorrectly, causing the executor to waste gas. The executor receives no reimbursement for their spent gas.

Recommendations

Short term, add a check to verify that the designator hash matches the `OtimDelegate` contract address externally before the `executeInstruction` function is called.

Long term, consider a design that does not require executors to pay gas costs up front.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeded industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category does not apply to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, implementing them can enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

FeeTokenRegistry

1. **Adjust the `weiToToken` function's revert reason on unregistered fee tokens for better debugging.** In the `weiToToken` function, if the fee token is not registered, the function returns the `invalidFeeToken` value to indicate the reason for reverting. However, it seems that the revert reason should be the `tokenNotRegistered` value instead.

OtimDelegate

1. **Implement support for EIP-1271.** With Otim transforming EOAs into smart contracts, it could impact signature verification in protocols that rely on EIP-1271. While EOAs can still sign messages as usual, some protocols verify signatures by checking the contract's code size and calling `target.isValidSignature()`. If Otim's wallet does not implement EIP-1271, these protocols may encounter verification issues.

D. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On March 24, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the Otim team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, Otim has resolved all nine issues disclosed in this report. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Lack of NFT callbacks	Medium	Resolved
2	Protocol does not handle tokens that do not return a Boolean	Medium	Resolved
3	Lack of validation of latestRoundData return value	Medium	Resolved
4	ERC-20 tokens cannot be withdrawn from Treasury contract	High	Resolved
5	Gas price calculation in OtimFee contract does not include priority fee	Informational	Resolved
6	Lack of maximum gas price protection in OtimFee contract	Medium	Resolved
7	Edge case when balance is at threshold	Informational	Resolved
8	Gas griefing vulnerability in native token transfer	Medium	Resolved
9	Delegate contract can be replaced to carry out gas griefing attack	Medium	Resolved

Detailed Fix Review Results

TOB-OTIM-1: Lack of NFT callbacks
Resolved in commit `3e86cf0f5d`. The team implemented a `Receiver` contract to properly handle incoming native ETH, ERC-721, and ERC-1155 tokens.

TOB-OTIM-2: Protocol does not handle tokens that do not return a Boolean
Resolved in commit `05b9bd3e43`. Token transfers now use OpenZeppelin's `safeTransfer` functionality, which handles tokens that do not return Booleans.

TOB-OTIM-3: Lack of validation of latestRoundData return value
Resolved in commit `64a30d4f`. The team implemented checks that validate the data returned by `latestRoundData` to ensure it is not stale.

TOB-OTIM-4: ERC-20 tokens cannot be withdrawn from Treasury contract
Resolved in commit `623fb69a49`. The team added a `withdrawERC20` function to enable ERC-20 token withdrawal from the `Treasury` contract.

TOB-OTIM-5: Gas price calculation in OtimFee contract does not include priority fee
Resolved in commit `94019f425c1c`. The team replaced `block.basefee` with `tx.gasprice` for a more accurate gas cost calculation.

TOB-OTIM-6: Lack of maximum gas price protection in OtimFee contract
Resolved in commit `94019f425c1c`. The team implemented the `maxBaseFeePerGas` and `maxPriorityFeePerGas` parameters for better gas fee control.

TOB-OTIM-7: Edge case when balance is at threshold
Resolved in commit `57a575a5bb`. The team implemented a change that allows users to set `threshold` to `0`, and their instruction will activate if the target's balance equals zero.

TOB-OTIM-8: Gas griefing vulnerability in native token transfer
Resolved in commit `52e278fc0`. The team implemented a `gasLimit` field to restrict gas consumption during external calls. Failed calls will deactivate the instruction while still charging users for gas costs.

TOB-OTIM-9: Delegate contract can be replaced to carry out gas griefing attack
Resolved in commit `82bd66c36`. The team implemented a `Gateway` contract to verify that the target code hash matches the expected "delegation designator" hash.

E. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries and government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on X and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.