QuillAudits

AUDIT REPORT

November 2025

For

YAMATO

UNIVERSAL
TRADING
INFRASTRUCTURE

Table of Content

Executive Summary

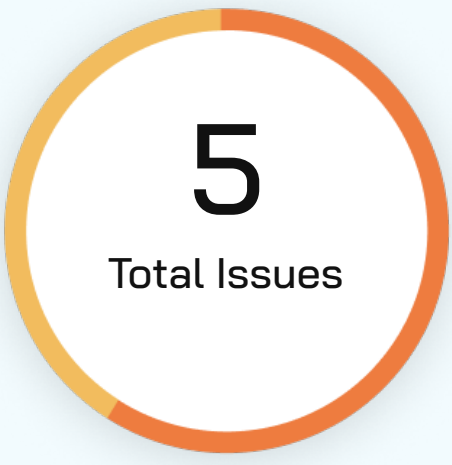
Project Name	Yamata
Project Type	Safe Timelock
Project URL	https://www.yamata.io/
Overview	This is a Gnosis Safe timelock module that enforces a delay period before executing transactions. Safe owners (excluding a designated "platform" address) can queue transactions with a specified execution time (ETA) that must be at least timelockDelay seconds in the future. Once queued, transactions can only be executed after their ETA has passed. The module provides functionality to queue, cancel, and execute transactions, with the contract owner able to update both the timelock delay and platform address. This design adds a security layer by preventing immediate transaction execution, allowing time for review and potential cancellation of malicious operations.
Audit Scope	The scope of this Audit was to analyze the Yamata Smart Contracts for quality, security, and correctness.
Source Code Link	https://github.com/MarronLab/safe-module-contracts/tree/main
Branch	main
Commit Hash	2de43196eed021f2105f6cc85310f3c1794b9869
Contracts in Scope	contracts/SafeTimelockModule.sol
Language	Solidity
Blockchain	EVM
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	4th November 2025 - 12th November 2025
Updated Code Received	18th November 2025
Review 2	20th November 2025

Fixed In	https://github.com/MarronLab/safe-module-contracts/tree/248d55199bd7899b40cca61ea2f864e72b200d0c

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	0 (0.0%)
High	0 (0.0%)
Medium	3 (60.0%)
Low	2 (40.0%)
Informational	0 (0.0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	0	**3**	**2**	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Lack of Transaction Uniqueness in queueTransaction Allows Collisions Between Identical Requests	Medium	Resolved
2	Missing Grace Period Allows Permanent Expiration of Queued Transactions	Medium	Resolved
3	Timelock Delay Can Be Set to Zero, Effectively Disabling Delay Protection	Medium	Resolved
4	Missing Two-Step Ownership Transfer Increases Risk of Accidental or Malicious Ownership Loss	Low	Resolved
5	Floating and Outdated Solidity Pragma May Lead to Inconsistent Compilation	Low	Resolved

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Computations Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC's conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls

- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall
- Upgradeable safety
- Using throw
- Using inline assembly
- Style guide violation
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Medium Severity Issues

Lack of Transaction Uniqueness in queueTransaction Allows Collisions Between Identical Requests

Resolved

Path

SafeTimelockModule.sol

Path

`queueTransaction()`

Description

The queueTransaction function derives its transaction identifier (txHash) solely from the tuple of parameters (safe, to, value, data, operation). Because no unique differentiator such as nonce, or timestamp is included in the hash computation, two queued transactions with identical inputs will produce the same txHash.

As a result, the contract cannot distinguish between them — if one transaction is canceled or executed, the other becomes permanently unexecutable or uncancelable, since both share the same storage slot in the queued/executed/canceled mappings.

Impact

Safe owners are unable to queue multiple identical transactions.

Likelihood

Medium

Recommendation

Introduce a unique identifier per queued transaction by including a per-Safenonce incremented on each new queue.

Missing Grace Period Allows Permanent Expiration of Queued Transactions

Resolved

Path

SafeTimelockModule.sol

Path

`executeTransaction()`

Description

The contract enforces that a queued transaction can only be executed once the current block timestamp exceeds the specified eta:

```
require(getBlockTimestamp() >= _eta, "ETA not reached");
```

However, there is no grace period defined after the eta, meaning if the transaction is not executed exactly after _eta, it can remain executable indefinitely or never be executed depending on future conditions.

In typical timelock implementations, a grace period is included to allow transactions to be executed within a fixed time window (e.g., 14 days after eta). After that window expires, the transaction becomes invalid to prevent stale or forgotten operations from being executed far in the future.

Impact

Queued transactions may remain valid indefinitely.

Likelihood

Medium

Recommendation

Introduce a grace period window within which queued transactions can be executed.

Timelock Delay Can Be Set to Zero, Effectively Disabling Delay Protection

Resolved

Path

SafeTimelockModule.sol

Path

`setTimelockDelay()`

Description

The setTimelockDelay function allows the contract owner to arbitrarily update the timelock delay without any validation:

```
function setTimelockDelay(uint256 _newDelay) external onlyOwner {
    timelockDelay = _newDelay;
    emit NewDelay(_newDelay);
}
```

Because no minimum delay check is enforced, the owner can set _newDelay to 0, effectively removing the timelock mechanism. This undermines the core purpose of the contract — enforcing a mandatory waiting period before executing queued transactions.

Impact

Setting delay to zero allows immediate transaction execution, bypassing the timelock's intended safeguard.

Likelihood

Medium

Recommendation

Enforce a minimum timelock delay to ensure that the delay cannot be completely disabled.

Low Severity Issues

Missing Two-Step Ownership Transfer Increases Risk of Accidental or Malicious Ownership Loss

Resolved

Path

SafeTimelockModule.sol

Description

`constructor()`

Description

The contract inherits from OpenZeppelin's Ownable and uses the standard single-step ownership transfer mechanism.
This means when the owner calls transferOwnership(newOwner), ownership is immediately transferred without requiring the new owner to accept it.

If an incorrect or non-receiving address (e.g., zero address, contract without owner handling) is passed, ownership can be irreversibly lost.
Furthermore, without a confirmation step, there is no safety check that the intended recipient actually controls the new address.

Floating and Outdated Solidity Pragma May Lead to Inconsistent Compilation

Resolved

Path

SafeTimelockModule.sol

Description

`pragma()`

Description

The contract specifies a floating and outdated Solidity compiler version.This allows compilation with any compiler version ⩾ 0.8.10 and < 0.9.0, which can result in inconsistent behavior or security vulnerabilities if a newer compiler introduces breaking changes, subtle semantic shifts, or unverified optimizations. Additionally, version 0.8.10 is outdated — later compiler versions include important bug fixes and security improvements (e.g., in ABI encoding, optimizer behavior, and EVM opcode handling).

Functional Tests

Some of the tests performed are mentioned below:

- Constructor should initialize platform and timelockDelay correctly and emit NewDelay and PlatformSet events
- Constructor should revert if _platform is the zero address
- onlyOwner modifier should restrict setTimelockDelay and setPlatform functions to the contract owner
- setTimelockDelay should update the delay value and emit NewDelay event
- setPlatform should update the platform address and emit PlatformSet event
- onlyNonPlatformOwner should allow calls only from Safe owners and reject the platform address
- onlyNonPlatformOwner should revert if caller is not a Safe owner
- queueTransaction should revert if _eta is earlier than current timestamp + timelockDelay
- queueTransaction should successfully store transaction hash and emit QueueTransaction event
- cancelTransaction should set queued transaction status to false and emit CancelTransaction event
- executeTransaction should revert if transaction is not queued
- executeTransaction should revert if block.timestamp < _eta
- executeTransaction should call IGnosisSafe.execTransactionFromModule and emit ExecuteTransaction event on success
- executeTransaction should revert if Safe transaction execution fails
- Multiple transactions with same parameters should produce identical txHash (confirm lack of uniqueness)
- Transaction re-execution should fail once queuedTransactions[txHash] is set to false after execution
- Delay validation — reducing timelockDelay to zero should allow instant queuing (confirm potential bypass)
- Timestamp check — confirm that queued transactions remain valid indefinitely (no grace period)

- ✔ **Getter functions (getPlatform, getTimelockDelay) should return correct stored values**
- ✔ **Event emission — verify that all events (NewDelay, PlatformSet, QueueTransaction, CancelTransaction, ExecuteTransaction) are emitted with accurate parameters**

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Threat Model

Contract	Function	Threats
SafeTimelockModule	constructor	Misconfiguration of _platform or _initialDelay may break governance flow; Zero address platform could disable execution control; Centralization risk due to single owner managing timelock configuration
	setTimelockDelay	Delay can be set to zero, disabling timelock; No upper bound could freeze governance if set excessively high; Missing event validation may cause off-chain desynchronization
	setPlatform	Setting platform to malicious or incorrect address could block Safe owners from interaction; Lack of validation enables accidental misconfiguration or lockout
	queueTransaction	Lack of transaction uniqueness — identical parameters produce same hash, leading to overwrites or DoS; ETA can be manipulated for premature execution; Reentrancy not expected but trust in Safe required; Incorrect _safe input could reference unauthorized Safe
	cancelTransaction	Absence of uniqueness allows canceling unrelated queued transactions with same parameters; Safe owner impersonation if getOwners spoofed or stale

Contract	Function	Threats
	executeTransaction	Missing grace period allows indefinite execution of old transactions; Reentrancy possible through execTransactionFromModule callback; Stale or malicious queued tx could execute after long delay; No replay protection — re-queuing with same params possible

Closing Summary

In this report, we have considered the security of Yamata. We performed our audit according to the procedure described above.

Issues of Medium and Low severity were found.Yamata team resolved all the issues mentioned in this report

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

November 2025

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com