QuillAudits

AUDIT REPORT

July 2025

For

PAYRAM

Table of Content

Table of Content

Executive Summary

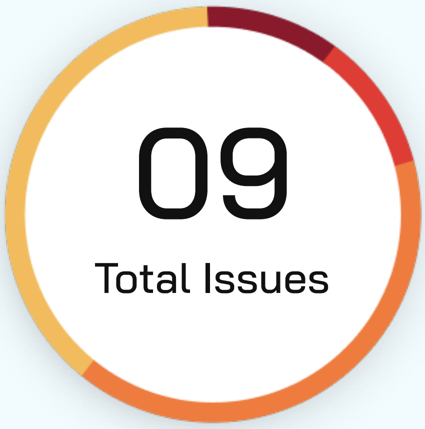
Project Name	Payram
Protocol Type	Payment gateway
Project URL	https://payram.com/
Overview	The Payram Smart Contract project consists of three primary smart contracts: GlobalFeeCollector, FundSweeper, and FactoryContract. This project provides a robust framework for deploying and managing smart contracts with specific functionalities for fee collection and fund sweeping. • **GlobalFeeCollector**: A universal contract deployed once for global fee collection. • **FundSweeper:** A contract deployed as an implementation and cloned for specific use cases. • **FactoryContract:** A factory contract used to deploy clones of the FundSweeper contract with specific initialization parameters.
Audit Scope	The scope of this Audit was to analyze the Payram Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/PayRam/payram-ethereum-contracts
Contracts in Scope	contracts/FundSweeperV2.sol contracts/GlobalFeeCollector.sol contracts/FundSweeperEIP7702.sol contracts/DepositWallet.sol contracts/FactoryContract.sol contracts/GlobalFeeCollectorEIP7702.sol contracts/IGlobalFeeCollector.sol contracts/FundSweeper.sol contracts/FactoryContractEIP7702.sol contracts/interfaces/IDepositWallet.sol contracts/IFundSweeper.sol
Branch	Main
Commit Hash	23ae9d275b0241a5f10373a4f301416e68faa99d
Language	Solidity

Blockchain	Ethereum
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	2nd July 2025 - 15th July 2025
Updated Code Received	16th July 2025 & 22nd July 2025
Review 2	16th July 2025 - 22nd July 2025
Fixed In	2e1d2451918b956e76338e3685d1abafd027fca3

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



🟥	Critical	11.1 (10%)
🟥	High	11.1 (10%)
🟧	Medium	44.4 (40%)
🟨	Low	33.3 (40%)
🟪	Informational	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	1	1	4	3	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Attacker Can Front–Run Deployment to Redirect Funds via Unauthorized Fund Collector	Critical	Resolved
2	Anyone Can Trigger Fund Movements and Fee Logic in Absence of Assigned Operator	High	Resolved
3	Fee Transfer Failure Can Reduce Future Fee Payouts to Fee Collector	Medium	Resolved
4	Contract Can Permanently Lock Ether, Affecting Admin Recoverability	Medium	Resolved
5	Operator Can Permanently Override Global Fee Rate for Specific Address	Medium	Resolved
6	Use a safe transfer helper library for ERC20 transfers	Medium	Resolved
7	Fee logic can unintentionally overcharge accounts intended to be zero–fee	Low	Resolved
8	Failed transfer can mislead off–chain accounting systems tracking GlobalFeeCollector	Low	Resolved

Issue No.	Issue Title	Severity	Status
9	Floating and Outdated Pragma	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.

Critical Severity Issues

Attacker Can Front—Run Deployment to Redirect Funds via Unauthorized Fund Collector

Resolved

Path

contracts/FactoryContractEIP7702.sol

Function Name

createFundSweeperContract()

Description

The FundSweeper contract is deployed via CREATE2 through a factory (FactoryContractEIP7702), using deterministic address computation derived from the proxy implementation, init parameters (master, globalFeeCollector), and a user-supplied salt. This allows the deposit wallet address to be computed off-chain prior to deployment.

However, the critical vulnerability lies in how fundCollector is initialized. The factory deploys the proxy contract with initialization parameters excluding fundCollector, and then immediately calls setInitialFundCollector(fundCollector) on the proxy. This second call is not part of the constructor, and therefore its logic does not influence the deterministic address.

Because the salt, master, and globalFeeCollector are fixed, any actor can deterministically compute and deploy the proxy contract at the same address before the intended deployer, so long as they use the exact same parameters for the ERC1967Proxy constructor. By doing this, a malicious attacker can front-run the legitimate deployer's createFundSweeperContract() call and successfully deploy a contract with the same logic and correct ownership (master), but with a malicious fundCollector.

Due to the missing access restriction in setInitialFundCollector, the attacker is able to set themselves as the fund collector before the original deployer's transaction executes. This gives the attacker control over where funds are sent. Critically, the contract allows anyone to invoke batchTransferTokens() and batchTransferEtherUsingDepositWallet() if the OPERATOR_ROLE has not yet been granted, due to the permissive check in operatorAllowed().

If the attacker calls these batch functions before the merchant takes administrative action (e.g., assigning roles or resetting the collector), they can drain all deposited tokens and ETH into the malicious fund collector address.

Impact

An attacker can front-run the contract deployment, deploy the contract with a malicious fund collector, and then immediately sweep all assets transferred into the deposit address before legitimate initialization completes.

Recommendation

Move all critical initialization logic into the constructor, i.e., pass the fundCollector parameter as part of the initialize() function invoked via the proxy constructor. This makes it part of the initCodeHash, which in turn changes the deployed address if the attacker attempts to modify any values.

High Severity Issues

Anyone Can Trigger Fund Movements and Fee Logic in Absence of Assigned Operator

Resolved

Path
contracts/FundSweeper.sol

Function
batchTransferTokens

Description
The FundSweeper contract employs a custom access control modifier operatorAllowed to restrict access to sensitive functions such as batchTransferTokens() and batchTransferEtherUsingDepositWallet(). This modifier is intended to enforce that only addresses holding the OPERATOR_ROLE can invoke these functions. However, its implementation fails to guarantee access control when no addresses have been granted the OPERATOR_ROLE.

Specifically, the operatorAllowed modifier first checks if the OPERATOR_ROLE has any members by evaluating getRoleMemberCount(OPERATOR_ROLE) > 0. Only if this count is non-zero does the modifier proceed to check that msg.sender has the appropriate role. If no members exist for the role, the modifier falls through without enforcing any access control, effectively making the guarded functions publicly callable.

Impact
If no operator has been set via the grantRole(OPERATOR_ROLE, ...) function, any external account can exploit this gap to trigger fund movement logic that was meant to be protected. This could allow unauthorized users to initiate transfers of ERC-20 tokens or Ether from the FundSweeper contract or from associated IDepositWallet smart wallets to arbitrary whitelisted fund collectors.

Recommendation
The operatorAllowed modifier should be rewritten to always enforce that the caller has the OPERATOR_ROLE, regardless of how many members are currently assigned and OPERATOR_ROLE should be set during the constructor itself.

QuillAudits's Response
Instead of removing OPERATOR_ROLE give OPERATOR_ROLE in the constructor and add that modifier batchTransferTokens() and batchTransferEtherUsingDepositWallet(). That would be more preferable.

Medium Severity Issues

Fee Transfer Failure Can Reduce Future Fee Payouts to Fee Collector

Resolved

Path

contracts/FundSweeper.sol

Function

distributeFunds()

Description

The FundSweeper contract's distributeFunds function attempts to send collected fees to a fee collector using a low-level .call() to the token contract. However, it does not maintain any accounting state or retry mechanism if the fee transfer fails. As a result, if the fee transfer fails during one invocation of batchTransferTokens, the fee amount remains locked in the contract. On the next invocation of batchTransferTokens, the fee for the new balance is calculated over the entire token balance in the contract, including any residual tokens left from the previous failed transfer. This logic leads to underpayment of the fee collector and overpayment to the fund collector.

To illustrate this, consider the following example:

- Initial token balance in the contract: 0.
- On the first call to batchTransferTokens, 100 tokens are collected.
- A 5% fee is computed, equating to 5 tokens. The remaining 95 tokens are transferred to the fund collector successfully.
- However, the transfer of 5 tokens to the fee collector fails (e.g., due to a revert from a fee collector contract).
- Now the contract holds 5 tokens (unclaimed fee).
- On a second call to batchTransferTokens, another 100 tokens are collected.
- The balance in the contract is now 105 tokens. The fee (5%) is calculated as 5.25 tokens, and the fund collector receives 99.75 tokens.
- This time, if the fee transfer succeeds, the fee collector only receives 5.25 tokens in total, even though they were owed 10 (5 from the first failed transfer and 5 from the second).

This cumulative underpayment arises because the contract does not separate failed fees from distributable balances or track untransferred fees in persistent storage. There are multiple instances in the contract where low-level .call operations are used, but failures are not properly handled. All such calls should explicitly revert on failure to prevent silent errors and ensure correct contract behavior and state updates.

QuillAudits's Response

All low level call should be reverted if they fail. Still in collectFunds() the low level call does not get reverted if failed.

Contract Can Permanently Lock Ether, Affecting Admin Recoverability

Resolved

Path

contracts/FactoryContract.sol

Function

receive()

Description

The FactoryContract contains a receive() function that enables it to accept native Ether deposits. However, it lacks any mechanism to withdraw or recover these funds once received. This is a critical oversight in contract design, especially for contracts that are not meant to hold user or operational balances in the form of native tokens. There is no function exposed that allows the contract owner or any authorized entity to transfer Ether out of the contract, and there is no automated refund or forwarding logic in place.

As a result, any Ether sent to the contract, whether intentionally (e.g., by a user mistaking the contract for a payable endpoint) or unintentionally (e.g., via self-destructs or malicious transfers), becomes permanently locked

Operator Can Permanently Override Global Fee Rate for Specific Address

Resolved

Path

src/GlobalFeeCollectorEIP7702.sol

Function

revokeFeePercentageForAddress()

Description

The vulnerability exists in the logic used to “revoke” a previously set fee percentage for a specific account in the fee management system, likely within a contract such as GlobalFeeCollector. The function revokeFeePercentageForAddress(address account) is intended to restore the account to using the global default fee rate. However, its current implementation merely resets the account’s fee to the current global rate by assigning feePercentageMap[account] = feePercentage; rather than deleting or nullifying the mapping entry.

The key issue arises due to this assignment taking a snapshot of the global fee rate at the time of revocation, rather than re-establishing a dynamic link to the global rate. As a result, if the global fee rate is updated in the future, the revoked account continues to use the stale snapshot value, which deviates from the intended behavior of “reverting back to the global default.” The fix should be to set the mapping back to 0, letting getFeeDetails… fallthrough to the latest global figure.

Use a safe transfer helper library for ERC20 transfers

Resolved

Path

contracts/GlobalFeeCollector.sol

Function

transfer()

Description

The GlobalFeeCollector.transfer(IERC20 token) function assumes that every ERC–20 token it handles conforms strictly to the OpenZeppelin–style interface and returns a Boolean flag on transfer. In practice a significant subset of production tokens, either return void (no data) or revert on failure while returning no value. When such a token is passed to transfer(IERC20 token), the high–level Solidity call expects a bool but receives zero bytes, causing ABI decoding to revert. The revert bubbles up, cancelling the entire transaction and leaving the collected fees stranded in the GlobalFeeCollector contract.

QuilAudits's Response

Convert the below call in collectFunds() also to safeTransferFrom()

```
// Call the transferFrom function on the token contract
                (success, data) = address(token).call(
                    abi.encodeWithSelector(
                        token.transferFrom.selector,
                        from,
                        address(this),
                        transferAmount
                    )
                );
```

Low Severity Issues

Fee logic can unintentionally overcharge accounts intended to be zero–fee

Resolved

Path

contracts/GlobalFeeCollector.sol

Function

getFeeDetailsForSender()

Description

GlobalFeeCollector allows administrators to assign account–specific fee rates via setFeePercentageForAddress(address,uint256). Internally, the mapping feePercentageMap stores the override, while getFeeDetailsForSender() interprets a zero value in that mapping as “no override, fall back to the global feePercentage.” This sentinel conflates two distinct meanings: “unset” and “explicitly zero.” When an administrator attempts to set a true zero–basis–point fee (to exempt a VIP address from fees) the assignment correctly records 0, but getFeeDetailsForSender() immediately treats that value as absent and returns the global rate instead. Therefore, it is impossible to configure a fee–free account; every address will be charged at least the globally configured percentage. Fix should be to treat “unset” with an auxiliary boolean or sentinel value (e.g. type(uint256).max) so that 0 remains a legal override.

Failed transfer can mislead off–chain accounting systems tracking GlobalFeeCollector

Resolved

Path

contracts/GlobalFeeCollector.sol

Function

transfer()

Description

The GlobalFeeCollector contract attempts to forward all ERC–20 balances in the transfer(IERC20 token) function and emits two events to signal the outcome. After calling token.transfer(feeCollector, balance), the code checks the returned Boolean. If the transfer fails, it emits FundTransferFailed; regardless of success or failure it then unconditionally emits FundTransferred. The analogous Ether–withdrawal routine transferEther() follows the same pattern. Because the success event is emitted even on failure, any indexing service, analytics pipeline, or automated reconciliation bot that relies on the FundTransferred event will record a successful payout that never occurred. The on–chain state remains unchanged while off–chain dashboards and financial reports diverge from reality.

Floating and Outdated Pragma

Resolved

Path

contracts/DepositWallet.sol

Function

pragma

Description

Locking the pragma prevents the contract from being compiled with outdated Solidity versions that might contain security flaws.
In this case the pragma is left open as >= 0.8.24, meaning the code could be compiled with any later compiler release, which re–introduces the very risk the lock is meant to avoid.
Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the 0.8.29 pragma version

Functional Tests

Some of the tests performed are mentioned below:

- ✓ initialize should deploy only once (subsequent call reverts)
- ✓ initialize should store fundSweeperImplementation correctly
- ✓ initialize should store globalFeeCollectorContract correctly
- ✓ initialize should grant DEFAULT_ADMIN_ROLE to deployer
- ✓ initialize should revert with InvalidFundSweeperImplementationAddress for zero address
- ✓ initialize should revert with InvalidFundSweeperImplementationAddress for non–contract address
- ✓ initialize should revert with InvalidGlobalFeeCollectorContractAddress for zero address
- ✓ initialize should revert with InvalidGlobalFeeCollectorContractAddress for non–contract address
- ✓ Only admin should call setFundSweeperImplementation
- ✓ setFundSweeperImplementation should update implementation address
- ✓ setFundSweeperImplementation should emit FundSweeperImplementationChanged event
- ✓ setFundSweeperImplementation should revert with InvalidFundSweeperImplementationAddress for zero address
- ✓ setFundSweeperImplementation should revert with InvalidFundSweeperImplementationAddress for EOA address
- ✓ Only current admin should call changeAdmin
- ✓ changeAdmin should transfer DEFAULT_ADMIN_ROLE to newAdmin
- ✓ changeAdmin should emit AdminChanged event
- ✓ changeAdmin should revert with InvalidAdminAddress for zero address
- ✓ createFundSweeperContract should deploy ERC1967Proxy deterministically with salt
- ✓ createFundSweeperContract should initialize proxy with owner, fundCollector, globalFeeCollector
- ✓ createFundSweeperContract should emit ContractDeployed event
- ✓ createFundSweeperContract should revert with InvalidFundCollectorAddress for zero fundCollector

- ✓ createFundSweeperContract should revert with InvalidSalt for zero salt
- ✓ createFundSweeperContract should revert with InvalidOwner for zero owner
- ✓ createFundSweeperContract should revert with InvalidFundCollectorSet when whitelist check fails
- ✓ createFundSweeperContract should revert with InvalidGlobalFeeCollectorSet when GFC mismatch
- ✓ createFundSweeperContract should revert on duplicate salt collision
- ✓ multipleDeployContract should deploy contracts for all salts provided
- ✓ multipleDeployContract should emit DepositWalletDeployed for each deployment
- ✓ multipleDeployContract should revert with InvalidSalt when salts array empty
- ✓ upgradeFundSweeperContract should succeed when caller has admin role on proxy
- ✓ upgradeFundSweeperContract should update proxy implementation to latest fundSweeperImplementation
- ✓ upgradeFundSweeperContract should emit ContractUpgraded event
- ✓ addFundCollector should add new address to whitelist and emit FundCollectorAdded
- ✓ addFundCollector should revert with FundCollectorAlreadyAdded on duplicate add
- ✓ batchTransferTokens should revert with FromParameterLengthMismatch on length mismatch
- ✓ batchTransferTokens should revert with InvalidFundCollectorAddress when fundCollector not whitelisted
- ✓ batchTransferTokens should query IGlobalFeeCollector and cap feePercentage at MAX_FEE_PERCENTAGE (500)
- ✓ distributeFunds should transfer amountAfterFee to fundCollector and emit FundTransferred
- ✓ distributeFunds should transfer capped fee to feeCollector and emit FeeTransferred
- ✓ distributeFunds should emit FeeTransferredFailed when fee transfer reverts
- ✓ _authorizeUpgrade should allow upgrade only to UPGRADE_ADMINS_ROLE holders
- ✓ changeFeeCollector should update feeCollector and emit FeeCollectorChanged
- ✓ changeFeeCollector should revert with InvalidFeeCollectorAddress for zero address
- ✓ transfer (ERC–20) should send full balance to feeCollector and emit FundTransferred
- ✓ transfer should emit FundTransferFailed if token.transfer returns false
- ✓ transfer should work even for tokens that return no/false boolean (mock to fail)

Threat Model

Contract	Function	Threats
FactoryContract	initialize	• Malicious or buggy _fundSweeperImplementation / _globalFeeCollectorContract set at deployment • Fee collector address can be hostile → siphons future fees
	setFundSweeperImplementation	• Rogue admin upgrading to back–doored implementation • Implementation set to EOA / address(0) can brick every future deploy/upgrade • Front–running race between multiple admins to choose different impl
	createFundSweeperContract	• Salt front–running: attacker pre–deploys at deterministic address to block deployment (address squatting) • Owner parameter points to hostile contract that self–upgrades proxy • FundCollector whitelist check can be bypassed if implementation contains bug (external dependency) • Reentrancy during constructor of custom FundSweeper impl (although nonReentrant, but indirect calls still possible) • DoS by sending empty bytecode or excessive gas usage

Contract	Function	Threats
	deployContract	• Anyone can deploy arbitrary bytecode via factory → brand hijack / deceptive wallets • Bytecode with self–destruct can later free address for attacker reuse • Salt collision griefing (deploy zero–bytecode first, then victim tx reverts)
	multipleDeployContract	• Batch loop gas–bomb can consume block gas limit → DoS to callers • Partial completion leaves unpredictable state (some wallets deployed, others not)
	upgradeFundSweeperContract	• Proxy admin check relies on proxy's hasRole; mis–configured FundSweeper may grant role too broadly • If current implementation already upgraded, call reverts → potential upgrade lockout
FundSweeper	initialize	• Incorrect _globalFeeCollectorContract or _fundCollector can divert funds. • Missing initializer protection allows re-initialization (takeover). • Setting _admin to address(0) can brick the contract.

Contract	Function	Threats
	addFundCollector / removeFundCollector	• Admin can whitelist malicious fund collectors. • Front-running could prevent valid additions/removals. • Removal of critical fund collector could halt fund flow.
	grantRoleGeneralized / revokeRoleGeneralized	• Risk of granting admin-level roles to wrong accounts. • Granting to zero address may cause undefined behavior. • Revoking active roles without replacement can break access.
	batchTransferTokens	• Possible reentrancy via malicious token contracts. • Fee collector could return high fee before cap logic is applied. • Array length mismatch causes revert (DoS vector). • Partial success may lead to inconsistent fund state.
	batchTransferEtherUsingDepositWallet	• External wallet contract may revert, consume gas, or block execution. • .call{value:} is vulnerable to reentrancy if fallback is unprotected. • Funds received during execution may not be swept.

Contract	Function	Threats
GlobalFeeCollector	initialize	• Missing initializer protection allows re-initialization (takeover). • Setting _feeCollector to address(0) results in permanent fund loss. • High _feePercentage (up to 100%) may enable full fund drain.
	transfer	• Transfers tokens without checking return value of ERC-20 transfer. • Non-standard tokens may not return boolean, causing false positives. • Emits success event regardless of actual success.
	transferEther	• Ether .call{value:} is vulnerable to fallback reentrancy. • Fails silently if feeCollector is address(0). • Event emission on failure may cause false reporting.
	receive	• ETH sent without context may be stuck unless manually transferred. • Attackers can dust the contract for log spamming.

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.

Closing Summary

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

Issues of Critical, High, Medium and Low severity were found. At the end the PayRam team resolved all the issues mentioned

Disclaimer

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.

QuillAudits

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

Follow Our Journey

AUDIT REPORT

July 2025

For

PAYRAM

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com