





AUDIT REPORT

August 2025


For

f[x]dx

Table of Content

Executive Summary	04
Number of Security Issues per Severity	06
Summary of Issues	07
Checked Vulnerabilities	09
Techniques and Methods	11
Types of Severity	13
Types of Issues	14
Severity Matrix	15
 Medium Severity Issues	16
1. Access Control Bypass in Timelock Contract – Proposal Spam and Manipulation	16
2. ETH Amount Mismatch in Router Deposit Function	18
3. Missing Binding of Parameters for Fund Recovery	20
4. Token Dust Triggers DoS for Token Removal Functionality	22
 Low Severity Issues	24
5. Missing Pause Modifier	24
6. ERC-20 Deposit Logic Incompatible with Fee-on-Transfer Tokens	26
7. Missing Timelock Enforcement on Sensitive System Actions – Contradicting Documentation vs Actual Implementation	28
8. Upgradeable Vault Missing Storage Gap (___gap)	30
9. CEI Pattern Violation	31



 Informational Issues	33
10. Inconsistent and Gas Inefficient String Messages in Require Statements	33
11. Unused Import in Timelock Contract	36
Functional Tests	37
Automated Tests	38
Threat Model	39
Closing Summary & Disclaimer	41



Executive Summary

Project Name FXDX

Protocol Type Vault

Project URL <https://portal.avituslabs.xyz/>

Overview Portal Wallet Contracts is a multi-contract vault system for managing deposits and withdrawals of ETH and ERC-20 tokens.

Core Capabilities:

- Multi-token deposits (ETH and ERC-20)
- Timelock-protected governance for critical operations (role changes, fund recovery, router updates)
- Upgradeable vault architecture using the UUPS proxy pattern
- Emergency controls with pause/unpause functionality and fund recovery mechanisms
- Separation across components

Architecture Components:

- Router: User-facing entry point for deposits and withdrawals
- Vault: Secure fund storage and accounting
- RoleManager: Access control and role administration
- Timelock: Enforces governance delays for sensitive actions

Audit Scope The scope of this Audit was to analyze Portal Wallet Smart Contracts for quality, security, and correctness.

Source Code link <https://github.com/Avitus-Labs/portal-wallet-contracts/tree/main>

Branch main

Contracts in Scope

1. contracts/router/WalletRouter.sol
2. contracts/vault/Vault.sol
3. contracts/interface/IVault.sol
4. contracts/access/RoleManager.sol
5. contracts/timelock/Timelock.sol
6. contracts/interface/ITimelock.sol
7. contracts/interface/IRoleManager.sol



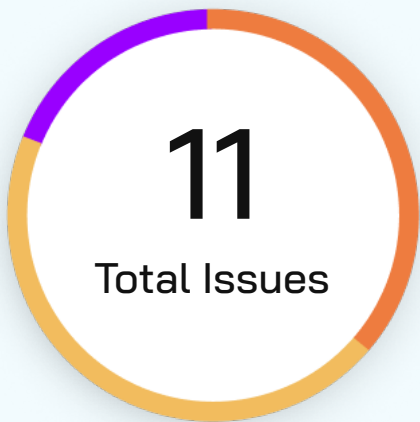
Commit Hash	cc265d685e0e6ac5be52987c39ca00ec7e95f890
Language	Solidity
Blockchain	Avitus
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	6th August 2025 to 18th August 2025
Updated Code Received	20th August 2025
Review 2	20th August 2025
Fixed In	0bf76efa69dd0c17663fba703a99c2e816f16a5f

Verify the Authenticity of Report on QuillAudits Leaderboard:

<https://www.quillaudits.com/leaderboard>



Number of Issues per Severity



Critical	0 (0%)
High	0 (0%)
Medium	4 (36.0%)
Low	5 (45.0%)
Informational	2 (19.0%)

		Severity				
		Critical	High	Medium	Low	Informational
Issues	Open	0	0	0	0	0
	Acknowledged	0	0	0	0	0
	Partially Resolved	0	0	0	0	0
	Resolved	0	0	4	5	2



Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Access Control Bypass in Timelock Contract – Proposal Spam and Manipulation	Medium	Resolved
2	ETH Amount Mismatch in Router Deposit Function	Medium	Resolved
3	Missing Binding of Parameters for Fund Recovery	Medium	Resolved
4	Token Dust Triggers DoS for Token Removal Functionality	Medium	Resolved
5	Missing Pause Modifier	Low	Resolved
6	ERC-20 Deposit Logic Incompatible with Fee-on-Transfer Tokens	Low	Resolved
7	Missing Timelock Enforcement on Sensitive System Actions – Contradicting Documentation vs Actual Implementation	Low	Resolved
8	Upgradeable Vault Missing Storage Gap (___gap)	Low	Resolved
9	CEI Pattern Violation	Low	Resolved



Issue No.	Issue Title	Severity	Status
10	Inconsistent and Gas Inefficient String Messages in Require Statements	Informational	Resolved
11	Unused Import in Timelock Contract	Informational	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		
		High	Medium	Low
Likelihood	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

Access Control Bypass in Timelock Contract – Proposal Spam and Manipulation

Resolved

Path

contracts/timelock/Timelock.sol

Function Name

```
proposeSetWalletRouter(address _walletAddress, bytes32 key, address caller)

executeSetWalletRouter(bytes32 actionId, address _walletRouter, address caller)

proposeRecoverFunds(address token, address recipient, uint256 amount, bytes32
key, address caller)

executeRecoverFunds(bytes32 actionId, address token, address caller)
```

Description

The Timelock contract contains an access control vulnerability where the `onlyVaultAdmin` modifier uses a user-controlled `caller` parameter instead of `msg.sender` to verify permissions.

```
/// @notice Restricts function access to accounts with the VAULT_ADMIN_ROLE.
/// @param caller The address to check.
/// @dev Reverts if the caller does not have the VAULT_ADMIN_ROLE.
modifier onlyVaultAdmin(address caller) {
    require(roleManager.hasRole(roleManager.VAULT_ADMIN_ROLE(), caller),
"Caller lacks VAULT_ADMIN_ROLE");
    _;
}
```

While the actual execution of sensitive operations is properly protected in the Vault contract, this vulnerability allows attackers to spam and manipulate proposals in the Timelock contract. But the most impactful consequence is that this vulnerability gives anyone the ability to delete legitimate proposals.

Impact : MEDIUM

- Attackers can flood the system with malicious proposals, consuming gas and interfering with governance.
- Attackers can delete legitimate admin proposals.
- Event logs get polluted with spam.
- No direct fund theft, but significant governance disruption.

Likelihood : MEDIUM

- Admin addresses are public and easily discoverable.
- Exploitation requires minimal effort and no special privileges.
- Cheap to execute, especially on low-gas chains.
- No direct monetary benefit for the attacker



Recommendation

It is recommended to remove the **caller** parameter and always use **msg.sender** in access control checks.

Specific Fixed In Commit

0bf76efa69dd0c17663fba703a99c2e816f16a5f

QuillAudits' Response

A new **onlyVault** modifier was added:

```
/// @notice Restricts function access to the Vault contract.
/// @dev Reverts if the caller is not the Vault contract.
modifier onlyVault() {
    if (msg.sender != vault) revert Error.OnlyVault();
    _;
}
```

All the affected functions were protected with the new **onlyVault** modifier



ETH Amount Mismatch in Router Deposit Function

Resolved

Path

contracts/router/WalletRouter.sol

Function Name

`deposit(address token, uint256 amount)`

Description

The `deposit()` function in the `WalletRouter` contract contains a vulnerability where it does not validate that `msg.value` equals the `amount` parameter for ETH deposits.

```
    /// @notice Deposits tokens or ETH into the vault.
    /// @param token The token address (address(0) for ETH).
    /// @param amount The amount to deposit.
    /// @dev Reverts if paused, amount is zero, vault is not set, or token is
    not supported. Emits Deposit event.
    function deposit(address token, uint256 amount) external payable
    nonReentrant whenNotPaused {
        require(amount > 0, "Invalid amount");
        require(address(vault) != address(0), "Vault not set");
        require(vault.isSupportedToken(token), "Token not supported");
        if (token == address(0)) {
            payable(address(vault)).sendValue(amount); // Send ETH to Vault
            vault.handleDeposit(msg.sender, token, amount); // Record deposit
        } else {
            uint256 balanceBefore = IERC20(token).balanceOf(address(vault));
            IERC20(token).safeTransferFrom(msg.sender, address(vault), amount);
            uint256 balanceAfter = IERC20(token).balanceOf(address(vault));
            uint256 receivedAmount = balanceAfter - balanceBefore;
            require(receivedAmount >= amount, "Token transfer failed");
            vault.handleDeposit(msg.sender, token, receivedAmount);
        }
        emit Deposit(msg.sender, token, amount, block.timestamp);
    }
```

This implementation allows a user to specify any `msg.value` and pass any number in the `amount` parameter, causing the Vault to register the wrong amount and possibly for the Router to use its own ETH balance to make up the difference.

Moreover - event emits the requested amount (not actual `msg.value`).

Impact : MEDIUM

- Vault records inflated/deflated deposit values.
- Router's ETH balance can be emptied quickly.

Likelihood : MEDIUM

- Easy to find via review/testing.
- No privileges required, only a public function call.
- Strong attacker incentive due to direct, instant profit.
- Requires previous ETH balance which is straightforward easy in the current business logic



Recommendation

It is recommended to add validation to ensure `msg.value == amount` for ETH deposits and use `msg.value` when transferring ETH and recording deposit amounts.

Suggested corrected implementation

```
if (token == address(0)) {
    require(msg.value == amount, "ETH amount mismatch");
    payable(address(vault)).sendValue(msg.value);
    vault.handleDeposit(msg.sender, token, msg.value);
}
```

Specific Fixed In Commit

0bf76efa69dd0c17663fba703a99c2e816f16a5f

QuillAudits' Response

The `deposit` function was modified to ensure `msg.value = amount`

```
/// @notice Deposits tokens or ETH into the vault.
/// @param token The token address (address(0) for ETH).
/// @param amount The amount to deposit.
/// @dev Reverts if paused, amount is zero, vault is not set, or token is
not supported. Emits Deposit event.
function deposit(address token, uint256 amount) external payable
nonReentrant whenNotPaused {
    if (amount == 0) revert Error.InvalidAmount();
    if (address(vault) == address(0)) revert Error.VaultNotSet();
    if (!vault.isSupportedToken(token)) revert Error.TokenNotSupported();

    if (token == address(0)) {
        if (msg.value != amount) revert Error.ETHAmountMismatch(); <= New
check
        payable(address(vault)).sendValue(amount); // Send ETH to Vault
        vault.handleDeposit(msg.sender, token, amount); // Record deposit
        emit Deposit(msg.sender, token, msg.value, block.timestamp);
    } else {
        ../
    }
}
```



Missing Binding of Parameters for Fund Recovery

Resolved

Path

contracts/timelock/Timelock.sol

Function Name

`proposeRecoverFunds(address token, address recipient, uint256 amount, bytes32 key, address caller)`

`executeRecoverFunds(bytes32 actionId, address token, address caller)`

Description

`proposeRecoverFunds` only stores `key`, `target` (set to `token`), and `executableAfter` in `PendingVault`. It does not store `recipient` or `amount`. Consequently, `executeRecoverFunds` cannot verify that the (`recipient`, `amount`) provided at execution time matches what was originally proposed. Even if you later wire execution to move funds, you cannot enforce that the same recipient and amount were time-locked.

```
/// @notice Proposes recovering funds from the Vault.
/// @param token The token address to recover.
/// @param recipient The address to receive the recovered funds.
/// @param amount The amount to recover.
/// @param key The action identifier.
/// @param caller The address proposing the action.
/// @dev Only callable by VAULT_ADMIN_ROLE. Reverts if action is already
proposed. Emits RecoverFundsPropose event.
function proposeRecoverFunds(address token, address recipient, uint256
amount, bytes32 key, address caller) external onlyVaultAdmin(caller) {
    bytes32 actionId = keccak256(abi.encode(key, token, recipient, amount,
block.timestamp));
    require(pendingVault[actionId].executableAfter == 0, "Action already
proposed");
    pendingVault[actionId] = PendingVault({
        key: key,
        target: token,
        executableAfter: block.timestamp + MIN_TIMELOCK_DELAY
    });
    emit RecoverFundsPropose(actionId, token, recipient, amount,
block.timestamp + MIN_TIMELOCK_DELAY);
}
```

Impact : MEDIUM

- Governance observers see a proposal for (A,100) but execution applies (B,1_000_000).
- Off-chain monitoring cannot rely on event → execution parity.

Likelihood : MEDIUM

- Parameter omission is systematic; every recovery proposal is affected.
- Easy to exploit but only by any entity with execution privileges



Recommendation

It is recommended to store all parameters for each action type and validate them on execution. Use typed structs per action, for example:

```
struct PendingRecoverFunds {  
    address token;  
    address recipient;  
    uint256 amount;  
    uint256 executableAfter;  
}
```

Specific Fixed In Commit

0bf76efa69dd0c17663fba703a99c2e816f16a5f

QuillAudits' Response

PendingRecoverFunds struct was updated to include the binding parameters **amount** and **recipient**:

```
/// @notice Struct to store pending fund recovery actions.  
/// @param key Unique identifier for the action.  
/// @param token Address of the token to recover.  
/// @param recipient Address to receive the recovered funds.  
/// @param amount Amount of tokens to recover.  
/// @param executableAfter Timestamp after which the action can be executed.  
struct PendingRecoverFunds {  
    bytes32 key;  
    address token;  
    address recipient;  
    uint256 amount;  
    uint256 executableAfter;  
}
```



Token Dust Triggers DoS for Token Removal Functionality

Resolved

Path

contracts/vault/Vault.sol

Function Name

```
removeSupportedToken(address token)
receive()
```

Description

Anyone can send ETH/tokens directly to the Vault, increasing on-chain balances without increasing `totalDeposits`. `removeSupportedToken` requires both tracked deposits and on-chain balances to be zero—single “dust” can permanently block removal.

```
/// @notice Executes the removal of a supported token after timelock
validation.
/// @param token The token address to remove.
/// @dev Only callable by VAULT_ADMIN_ROLE. Emits TokenSupportRemoved event.
function removeSupportedToken(address token) external onlyVaultAdmin {
    require(supportedTokens[token], "Token not supported");
    require(totalDeposits[token] == 0, "Cannot remove token with deposits");
    if (token == address(0)) {
        require(address(this).balance == 0, "Vault has ETH balance");
    } else {
        require(IERC20(token).balanceOf(address(this)) == 0, "Vault has
token balance");
    }
    supportedTokens[token] = false;
    emit TokenSupportRemoved(token);
}
```

Impact : MEDIUM

- DoS on token removal; accounting divergence between tracked and actual balances.

Likelihood : HIGH

- Anyone can dust the Vault.

Recommendation

Add a timelocked `sweepDust(token, to, amount)` or `reconcileExtraBalance(token)` to align accounting; documents that direct sends are unsupported. Consider gating removal on tracked deposits only, if acceptable.

Specific Fixed In Commit

0bf76efa69dd0c17663fba703a99c2e816f16a5f



QuillAudits' Response

4 new functions were added to implement and segment dust removal into proposal and execution logic, the functions account for possibly deposited tokens and implement checks accordingly:

```
function sweepDust(address token, address to, uint256 amount, bytes32
actionId) external onlyVaultAdmin {
    if (amount == 0) revert Error.InvalidAmount();
    if (!timelock.executeSweepDust(actionId, token, to, amount)) revert
Error.ExecutionFailed();

    if (token == address(0)) {
        uint256 unaccountedETH = address(this).balance -
totalDeposits[address(0)];
        if (amount > unaccountedETH) revert
Error.CannotSweepTokenWithDeposit();
        payable(to).sendValue(amount);
    } else {
        if (!isContract(token)) revert Error.TokenIsNotAContract();
        uint256 unaccounted =
IERC20Upgradeable(token).balanceOf(address(this)) - totalDeposits[token];
        if (amount > unaccounted) revert
Error.CannotSweepTokenWithDeposit();
        IERC20Upgradeable(token).safeTransfer(to, amount);
    }
    emit DustSwept(token, to, amount);
}
```



Low Severity Issues

Missing Pause Modifier

Resolved

Path

contracts/router/WalletRouter.sol

Function Name

`setVault(address _vault)`

Description

The `setVault()` function in the `WalletRouter` contract lacks the **whenNotPaused** modifier, allowing the vault address to be changed even when the system is paused. This bypasses the intended pause mechanism that is followed all throughout the codebase and could allow critical system changes during emergencies.

```
/// @notice Executes the setting of a new vault address after timelock
validation.
/// @param _vault The new vault address to set.
/// @dev Only callable by ROUTER_ADMIN_ROLE. Emits VaultSet event on
success.
function setVault(address _vault) external onlyRouterAdmin {
    require(_vault != address(0), "Invalid vault address");
    vault = IVault(_vault);
    emit VaultSet(_vault);
}
```

Impact : MEDIUM

- Enables system reconfiguration during emergency pauses.
- Malicious admin could change vault during a security incident.

Likelihood : LOW

- Only **ROUTER_ADMIN_ROLE** can exploit.
- Occurs only when paused.
- Readily visible via code review.

Recommendation

It is recommended to add the **whenNotPaused** modifier to `setVault` to enforce pause state compliance.



Suggested corrected implementation

```
/// @notice Executes the setting of a new vault address after timelock
validation.
/// @param _vault The new vault address to set.
/// @dev Only callable by ROUTER_ADMIN_ROLE. Emits VaultSet event on
success.
function setVault(address _vault) external onlyRouterAdmin whenNotPaused {
require(_vault != address(0), "Invalid vault address");
    vault = IVault(_vault);
    emit VaultSet(_vault);
}
```

Specific Fixed In Commit

0bf76efa69dd0c17663fba703a99c2e816f16a5f

QuillAudits' Response

The `whenNotPaused` modifier was added to the `setVault` function.



ERC-20 Deposit Logic Incompatible with Fee-on-Transfer Tokens

Resolved

Path

contracts/router/WalletRouter.sol

Function Name

deposit(address token, uint256 amount)

Description

Router computes `receivedAmount = balanceAfter - balanceBefore`, then requires `receivedAmount >= amount`, which fails for fee-on-transfer tokens.

```
/// @notice Deposits tokens or ETH into the vault.
/// @param token The token address (address(0) for ETH).
/// @param amount The amount to deposit.
/// @dev Reverts if paused, amount is zero, vault is not set, or token is
not supported. Emits Deposit event.
function deposit(address token, uint256 amount) external payable
nonReentrant whenNotPaused {
    require(amount > 0, "Invalid amount");
    require(address(vault) != address(0), "Vault not set");
    require(vault.isSupportedToken(token), "Token not supported");

    if (token == address(0)) {
        payable(address(vault)).sendValue(amount); // Send ETH to Vault
        vault.handleDeposit(msg.sender, token, amount); // Record deposit
    } else {
        uint256 balanceBefore = IERC20(token).balanceOf(address(vault));
        IERC20(token).safeTransferFrom(msg.sender, address(vault), amount);
        uint256 balanceAfter = IERC20(token).balanceOf(address(vault));
        uint256 receivedAmount = balanceAfter - balanceBefore;
        require(receivedAmount >= amount, "Token transfer failed");
        vault.handleDeposit(msg.sender, token, receivedAmount);
    }
    emit Deposit(msg.sender, token, amount, block.timestamp);
}
```

Impact : MEDIUM

- DoS for deflationary tokens; deposits always revert.

Likelihood : LOW

- Many tokens apply transfer fees.

Recommendation

It is recommended to make a decision and document if fee-on-transfer tokens will be supported, then pick a policy: (A) don't support such tokens (require equality), or (B) support them—accept `receivedAmount > 0`, account and emit the **actual** received amount consistently.



```
    /// @notice Executes the setting of a new vault address after timelock
validation.
    /// @param _vault The new vault address to set.
    /// @dev Only callable by ROUTER_ADMIN_ROLE. Emits VaultSet event on
success.
    function setVault(address _vault) external onlyRouterAdmin whenNotPaused {
require(_vault != address(0), "Invalid vault address");
    vault = IVault(_vault);
    emit VaultSet(_vault);
    }
```

Specific Fixed In Commit

0bf76efa69dd0c17663fba703a99c2e816f16a5f

QuillAudits' Response

The check that blocked deposit of fee-on-transfer tokens was removed and the actual received amount is admitted in the Deposit() event:

```
    function deposit(address token, uint256 amount) external payable
nonReentrant whenNotPaused {
    ../
        emit Deposit(msg.sender, token, receivedAmount, block.timestamp);
    }
}
```



Missing Timelock Enforcement on Sensitive System Actions – Contradicting Documentation vs Actual Implementation

Resolved

Path

contracts/wallet/WalletRouter.sol, contracts/vault/Vault.sol

Function Name

`setVault(address _vault)`

`addSupportedToken(address token, bytes32 actionId)`

`removeSupportedToken(address token, bytes32 actionId)`

Description

The architecture specifies a **Timelock** contract intended to enforce delayed execution for sensitive governance actions. However, several high-impact system functions bypass the timelock entirely, allowing an admin to execute them instantly, despite the NatSpec comments stating they can only execute after timelock validation.

Design expectation

All governance actions (vault migration, supported token changes) should be proposed and executed only after `MIN_TIMELOCK_DELAY`.

Actual behavior

`WalletRouter.setVault` is callable directly by `ROUTER_ADMIN_ROLE` with no timelock.

```
/// @notice Executes the setting of a new vault address after timelock
validation.
/// @param _vault The new vault address to set.
/// @dev Only callable by ROUTER_ADMIN_ROLE. Emits VaultSet event on
success.
function setVault(address _vault) external onlyRouterAdmin {
    require(_vault != address(0), "Invalid vault address");
    vault = IVault(_vault);
    emit VaultSet(_vault);
}
```

Token list changes in Vault does not require timelock.

```
/// @notice Executes the addition of a supported token after timelock
validation.
/// @param token The token address to add.
/// @dev Only callable by VAULT_ADMIN_ROLE. Emits TokenSupportAdded event.
function addSupportedToken(address token) external onlyVaultAdmin {
    require(!supportedTokens[token], "Token already supported");
    if (token != address(0)) {
        require(isContract(token), "Token is not a contract");
    }
    supportedTokens[token] = true;
    emit TokenSupportAdded(token);
}
```



```
/// @notice Executes the removal of a supported token after timelock
validation.
/// @param token The token address to remove.
/// @dev Only callable by VAULT_ADMIN_ROLE. Emits TokenSupportRemoved event.
function removeSupportedToken(address token) external onlyVaultAdmin {
    require(supportedTokens[token], "Token not supported");
    require(totalDeposits[token] == 0, "Cannot remove token with deposits");
    if (token == address(0)) {
        require(address(this).balance == 0, "Vault has ETH balance");
    } else {
        require(IERC20(token).balanceOf(address(this)) == 0, "Vault has
token balance");
    }
    supportedTokens[token] = false;
    emit TokenSupportRemoved(token);
}
```

Impact : MEDIUM

- Partially undermines the "delayed execution" guarantees expected from a timelock design.
- Users have no on-chain delay to react to governance changes.
- Documentation mismatch

Likelihood : LOW

- Restricted Access
- No complex exploitation steps – direct function call possible immediately.

Recommendation

Make the implementation adhere to the documentation or adjust the documentation to reflect the real flow.

Specific Fixed In Commit

0bf76efa69dd0c17663fba703a99c2e816f16a5f

QuillAudits' Response

The documentation was changed to reflect the implementation without timelock validation.



Upgradeable Vault Missing Storage Gap (__gap)

Resolved

Path

contracts/vault/Vaults.sol

Description

The Vault contract is upgradeable via UUPS (UUPSUpgradeable) but lacks the standard storage gap (`uint256[50] private __gap;`) at the end of its storage. In upgradeable contracts, adding new state variables in future versions can shift storage and overwrite existing slots, causing storage collisions, corrupted accounting, and potentially unrecoverable funds. OpenZeppelin recommends reserving a gap to allow safe variable additions without breaking layout.

Impact : MEDIUM

Without a gap, future upgrades risk storage collisions that can corrupt balances (e.g., `totalDeposits`, `supportedTokens`), break governance state, and make upgrades brittle—even if the new logic is correct.

Likelihood : MEDIUM

The contract is intended to be upgraded; future versions are likely to introduce new variables. With no gap in place, the very next upgrade risks breaking storage.

Recommendation

It is recommended to add a storage gap at the end of the contract to permit safe future upgrades:

```
uint256[50] private __gap;
```

Additional notes: keep variable order stable across versions, follow OpenZeppelin's upgradeable patterns, and include proxy upgrade tests that validate storage integrity across versions.

Specific Fixed In Commit

0bf76efa69dd0c17663fba703a99c2e816f16a5f

QuillAudits' Response

The `__gap` private variable is an array of reserved storage slots placed at the end of an upgradeable contract. It ensures that future variables can be added without shifting the positions of existing storage slots, which would otherwise corrupt state during upgrades.



CEI Pattern Violation

Resolved

Path

contracts/vault/Vault.sol, contracts/access/RoleManager.sol

Function Name

`handleWithdrawal(address recipient, address token, uint256 amount)``executeRoleAction(bytes32 actionId)``acceptAdminTransfer()`

Description

Transfers occur before decrementing `totalDeposits` and the function isn't `nonReentrant`. Although callable only by the Router, defense-in-depth and CEI are standard for asset transfers.

1. Vault.sol - handleWithdrawal()

```
function handleWithdrawal(address recipient, address token, uint256 amount)
external onlyWalletRouter onlySupportedToken(token) whenNotPaused {
    require(totalDeposits[token] >= amount, "Insufficient tracked deposits");
    if (token == address(0)) {
        require(address(this).balance >= amount, "Insufficient Vault ETH
balance");
        payable(recipient).sendValue(amount); // ✗ INTERACTION before EFFECTS
    } else {
        require(IERC20(token).balanceOf(address(this)) >= amount, "Insufficient
Vault token balance");
        IERC20(token).safeTransfer(recipient, amount); // ✗ INTERACTION before
EFFECTS
    }
    totalDeposits[token] -= amount; // ✗ EFFECTS after INTERACTIONS
    emit WithdrawalProcessed(recipient, token, amount);
}
```

2. RoleManager.sol - executeRoleAction()

```
function executeRoleAction(bytes32 actionId) external
onlyRole(DEFAULT_ADMIN_ROLE) {
    PendingRoleAction memory action = pendingRoleActions[actionId];
    require(action.executableAfter != 0, "Action not proposed");
    require(block.timestamp >= action.executableAfter, "Timelock not expired");

    if (action.isGrant) {
        _grantRole(action.role, action.account); // ✗ INTERACTION before
EFFECTS
    } else {
        _revokeRole(action.role, action.account); // ✗ INTERACTION before
EFFECTS
    }
    emit RoleChangeExecuted(actionId, action.role, action.account,
action.isGrant);
    delete pendingRoleActions[actionId]; // ✗ EFFECTS after INTERACTIONS
}
```



3. RoleManager.sol - acceptAdminTransfer()

```
function acceptAdminTransfer() external {
    PendingAdminTransfer memory transfer = pendingAdminTransfers[msg.sender];
    require(hasRole(DEFAULT_ADMIN_ROLE, transfer.oldAdmin), "Old admin lacks
DEFAULT_ADMIN_ROLE");
    require(transfer.executableAfter != 0, "No transfer proposed");
    require(block.timestamp >= transfer.executableAfter, "Timelock not
expired");
    _grantRole(DEFAULT_ADMIN_ROLE, msg.sender); // ✗ INTERACTION before EFFECTS
    _revokeRole(DEFAULT_ADMIN_ROLE, transfer.oldAdmin); // ✗ INTERACTION before
EFFECTS
    emit AdminTransferAccepted(transfer.oldAdmin, msg.sender);
    delete pendingAdminTransfers[msg.sender]; // ✗ EFFECTS after INTERACTIONS
}
```

CEI Pattern requires:

- 1 Checks - Validate conditions
- 2 Effects - Update state variables
- 3 Interactions - Call external contracts/transfers

These functions violate CEI by:

- Performing external calls (sendValue, safeTransfer, _grantRole, _revokeRole) before updating state
- Updating state variables after external interactions

This creates reentrancy vulnerabilities and state inconsistencies

Impact : LOW

While these CEI violations might not cause classic reentrancy right now - they can cause accounting inconsistencies or unexpected behavior, especially if future integrations expand call paths.

Likelihood : MEDIUM

The current caller is constrained, but patterns evolve and tokens may have hooks.

Recommendation

It is recommended to apply CEI patterns to all the functions listed above (for example decrement before transfer) and follow CEI in all future contributions to the codebase.

Specific Fixed In Commit

Obf76efa69dd0c17663fba703a99c2e816f16a5f

QuillAudits' Response

All 3 affected functions were refactored to follow the CEI pattern.



Informational Issues

Inconsistent and Gas Inefficient String Messages in Require Statements

Resolved

Path

all

Function Name

all

Description

Similar validation failures produce different revert messages across contracts and uses string messages throughout the codebase, which is not optimal from a gas efficiency perspective. This inconsistency causes poor UX, complicates debugging and integrations that parse errors, and increases support burden. Standardizing errors (and preferably migrating to custom errors) yields clearer APIs and lower gas.

Here are all the unique require string messages throughout all contracts in the contracts folder:

```
"Action already proposed"
"Action not proposed"
"Caller lacks VAULT_ADMIN_ROLE"
"Cannot remove token with deposits"
"Delay must be at least 1 day"
"Implementation is not a contract"
"Insufficient Vault ETH balance"
"Insufficient Vault token balance"
"Insufficient tracked deposits"
"Invalid WalletRouter"
"Invalid account"
"Invalid admin address"
"Invalid amount"
"Invalid implementation address"
"Invalid key"
"Invalid recipient"
"Invalid RoleManager"
"Invalid Timelock"
"Invalid vault address"
"No transfer proposed"
"Not WalletRouter"
"Not operator"
"Not router admin"
"Old admin lacks DEFAULT_ADMIN_ROLE"
"Only Router Admin can call this function"
"Recover funds not executed"
"Set WalletRouter not executed"
"Timelock not expired"
"Token already supported"
"Token is not a contract"
"Token mismatch"
```



```
"Token not supported"  
"Token transfer failed"  
"Vault has ETH balance"  
"Vault has token balance"  
"Vault not set"  
"WalletRouter is not a contract"  
"WalletRouter mismatch"
```

Recommendation

It is recommended to convert all string messages to custom errors for gas savings and programmatic handling.

Like so:

```
error ActionAlreadyProposed();  
error ActionNotProposed();  
error CallerLacksVaultAdminRole();  
error CannotRemoveTokenWithDeposits();  
error DelayMustBeAtLeastOneDay();  
error ImplementationIsNotAContract();  
error InsufficientVaultETHBalance();  
error InsufficientVaultTokenBalance();  
error InsufficientTrackedDeposits();  
error InvalidWalletRouter();  
error InvalidAccount();  
error InvalidAdminAddress();  
error InvalidAmount();  
error InvalidImplementationAddress();  
error InvalidKey();  
error InvalidRecipient();  
error InvalidRoleManager();  
error InvalidTimelock();  
error InvalidVaultAddress();  
error NoTransferProposed();  
error NotWalletRouter();  
error NotOperator();  
error NotRouterAdmin();  
error OldAdminLacksDefaultAdminRole();  
error RecoverFundsNotExecuted();  
error SetWalletRouterNotExecuted();  
error TimelockNotExpired();  
error TokenAlreadySupported();  
error TokenIsNotAContract();  
error TokenMismatch();  
error TokenNotSupported();  
error TokenTransferFailed();  
error VaultHasETHBalance();  
error VaultHasTokenBalance();  
error VaultNotSet();  
error WalletRouterIsNotAContract();  
error WalletRouterMismatch();
```



Make the errors less or more descriptive. Document all errors and meanings in the README/spec.

Add unit tests asserting specific errors are thrown for each failure path.

Specific Fixed In Commit

0bf76efa69dd0c17663fba703a99c2e816f16a5f

QuillAudits' Response

All string error messages were removed and replaced with custom errors that declared in portal-wallet-contracts/contracts/error/Error.sol file. 7 new error messages were declared.



Unused Import in Timelock Contract

Resolved

Path

contracts/timelock/Timelock.sol

Description

The Timelock contract includes an unused import statement for the IVault interface, which is never referenced in the code. This increases deployment gas costs slightly and can create confusion for developers.

Recommendation

It is recommended to remove the unused IVault import.

Specific Fixed In Commit

0bf76efa69dd0c17663fba703a99c2e816f16a5f

QuillAudits' Response

Unused import was removed.



Functional Tests

Some of the tests performed are mentioned below:

✓ **1. Role Management Flow Test**

testRoleManager_ProposeAndExecuteRoleGrant

Purpose: Tests timelock-based role granting mechanism

Tests: Propose role grant → wait timelock → execute → verify role granted

✓ **2. Complete Deposit-Withdrawal Integration Test**

testIntegration_CompleteDepositWithdrawalFlow

Purpose: Tests end-to-end user deposit and withdrawal flow

Tests: User deposits ETH → vault tracks → operator withdraws → verify balances

✓ **3. Timelock Governance Flow Test**

testIntegration_TimelockGovernanceFlow

Purpose: Tests timelock-based governance for system parameter changes

Tests: Propose router change → wait timelock → execute → update vault

✓ **4. Access Control Bypass Vulnerability Test**

testAccessControlBypass

Purpose: Reveals critical access control flaw in timelock contract

Tests: Attacker bypasses access controls by passing admin address as parameter

✓ **5. Fund Recovery Security Test**

testVault_ProposeAndRecoverFunds

Purpose: Tests secure fund recovery mechanism with timelock protection

Tests: Propose recovery → wait timelock → execute → verify funds transferred

✓ **6. Pause/Unpause Functionality Test**

testEdgeCase_PausedOperations

Purpose: Tests emergency pause and resume functionality

Tests: Pause router → verify operations blocked → unpause → verify operations resume

✓ **7. Token Support Management Test**

testVault_AddAndRemoveSupportedToken

Purpose: Tests adding and removing supported tokens

Tests: Add token → verify supported → remove token → verify not supported



✓ 8. Router Configuration Test

testVault_ProposeAndSetWalletRouter

Purpose: Tests secure router replacement with timelock protection

Tests: Propose new router → wait timelock → execute → verify vault uses new router

✓ 9. Edge Case: Insufficient Funds Test

testEdgeCase_WithdrawMoreThanDeposited

Purpose: Tests financial safety - prevents over-withdrawal

Tests: Attempt withdraw more than deposited → verify transaction reverts

✓ 10. Proxy Initialization Test

testVaultInitialization

Purpose: Tests upgradeable vault proxy initialization

Tests: Verify vault proxy has correct role manager, router, and timelock addresses

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
Timelock	proposeSetWalletRouter()	Access Control Bypass - Attacker can pass admin address as parameter to bypass role checks
Timelock	proposeRecoverFunds()	Access Control Bypass - Attacker can propose fund recovery by passing admin address
Timelock	executeSetWalletRouter()	Timelock Bypass - Execution before timelock period expires
Timelock	executeRecoverFunds()	Timelock Bypass - Execution before timelock period expires
Vault	handleDeposit()	Invalid Input - Zero amount deposits, unsupported tokens
Vault	handleWithdrawal()	Insufficient Funds - Withdrawing more than deposited amount
Vault	recoverFunds()	Access Control - Non-admin attempting fund recovery
Vault	addSupportedToken()	Access Control - Non-vault-admin adding tokens
Vault	removeSupportedToken()	Business Logic - Removing token with existing deposits
Vault	setWalletRouter()	Access Control - Non-admin changing router address
Vault	pause() / unpause()	Access Control - Non-admin pausing/unpausing operations
WalletRouter	deposit()	Pause Bypass - Depositing while system is paused



Contract	Function	Threats
WalletRouter	withdraw()	Access Control - Non-operator attempting withdrawals
WalletRouter	setVault()	Access Control - Non-router-admin changing vault address
RoleManager	proposeGrantRole()	Timelock Bypass - Executing role grants before timelock expires
RoleManager	proposeRevokeRole()	Timelock Bypass - Executing role revokes before timelock expires
RoleManager	executeRoleAction()	Access Control - Non-admin executing role actions
RoleManager	cancelRoleAction()	Access Control - Non-admin cancelling role actions
RoleManager	proposeAdminTransfer()	Access Control - Non-admin proposing admin transfers
RoleManager	acceptAdminTransfer()	Access Control - Non-proposed admin accepting transfer

Closing Summary

In this report, we have considered the security of FXDX. We performed our audit according to the procedure described above. We conducted both automated and manual testing, including scenario-based attacks, to identify potential vulnerabilities and deviations from best practices.

While several issues of varying severity were identified, the majority can be addressed with straightforward code changes and adherence to established development patterns such as Checks-Effects-Interactions (CEI) and defensive coding practices. We commend the development team for their overall clean and modular codebase, which facilitated the audit process and indicates a thoughtful approach to security.

It is recommended to address all findings, regardless of severity, and conduct a final review after remediation. Security is an ongoing process, and we encourage the team to implement continuous testing, monitoring, and secure development practices throughout the lifecycle of the project.

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

August 2025

For

f[x]dx



Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillaudits.com