Zellic

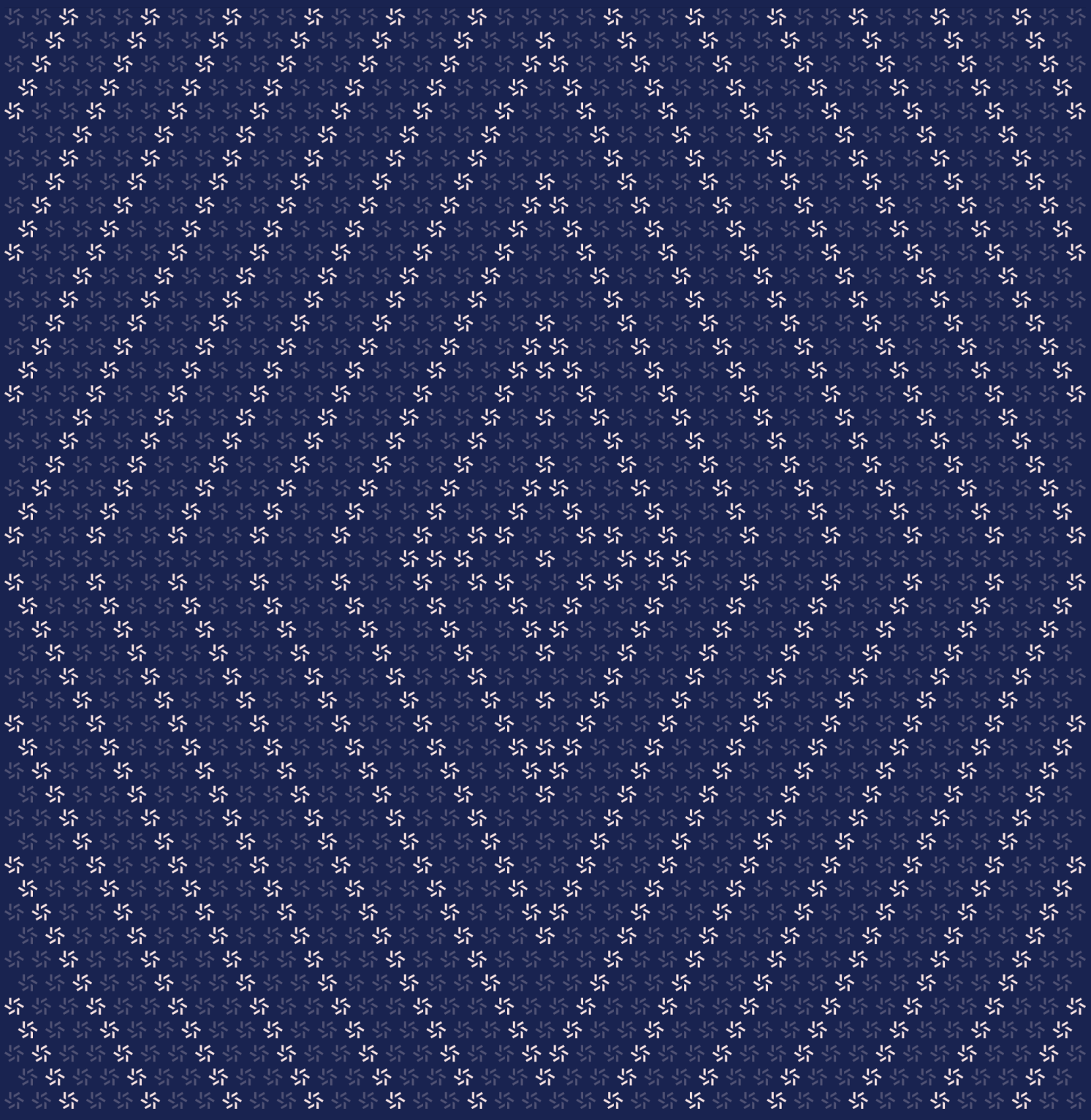
Prepared for
Luka Avbreht
Boštjan Kovač
Flare Network

Prepared by
Kritsada Dechawattana
Weipeng Lai
Zellic

February 12, 2026

Smart Accounts

Smart Contract Patch Review



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

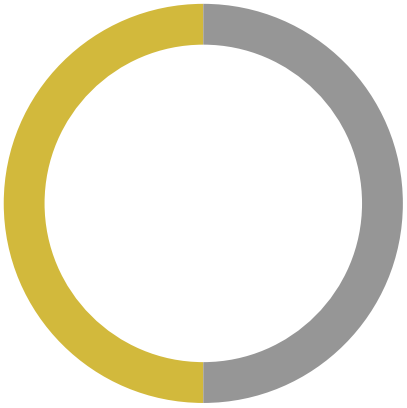
Zellic conducted a security assessment for Flare Network from February 6th to February 11th, 2026. During this engagement, Zellic reviewed Smart Accounts's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Results

During our assessment on the scoped Smart Accounts contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and the other finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	0
Informational	1



2. Introduction

We were asked to review a minor patch to Smart Accounts, which refactored some contract structures and introduced a few new contracts, based on the previous version.

2.1. Scope

The engagement involved a review of the following targets:

Smart Accounts Contracts

Type	Solidity
Platform	EVM-compatible
Target	Only changes between 1ee58c1...01a5140
Repository	https://github.com/flare-foundation/flare-smart-accounts ↗
Version	`01a514007017cbfd609dcbd985bd3c35467353b5`
Programs	`contracts/smartAccounts/**/*.sol`

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

The following consultants were engaged to conduct the assessment:

Kritsada Dechawattana
Engineer
kritsada@zellic.io

Weipeng Lai
Engineer
weipeng.lai@zellic.io

2.2. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.

3. Detailed Findings

3.1. Missing LP token approval in `requestRedeem` blocks Upshift vault withdrawals

Target	PersonalAccount		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `PersonalAccount.requestRedeem` function calls `IIVault(_vault).requestRedeem(_shares, address(this))` without first approving the vault to spend the PersonalAccount's LP tokens.

The test mock (`MyERC4626`) inherits from OpenZeppelin's ERC-4626, where the vault itself is the share token and can burn shares directly via the privileged internal `_burn` function — no approval required. However, the actual Upshift vault ↗ relies on a separate LP token contract and pulls tokens from the caller via `safeTransferFrom`:

```
SafeERC20Upgradeable.safeTransferFrom(
    IERC20Upgradeable(lpTokenAddress), msg.sender, address(this), shares
);
```

This requires the caller to have previously approved the vault to spend at least `shares` worth of LP tokens. Since `PersonalAccount.requestRedeem` never approves the LP token, the call will always revert.

Impact

All redemption requests from PersonalAccounts to Upshift vaults will revert, preventing users from withdrawing funds via the intended `requestRedeem` -> `claim` flow. LP tokens remain locked in the PersonalAccount until the code is fixed and upgraded.

Recommendations

Add LP token approval in `PersonalAccount.requestRedeem` before calling the vault. Additionally, extend the `IIVault` interface with an `lpTokenAddress()` getter to facilitate this approval.

```
function requestRedeem(
    address _vault,
    uint256 _shares
)
    external
```

```
    onlyController nonReentrant
    returns (uint256 _claimableEpoch, uint256 _year, uint256 _month,
    uint256 _day)
{
    address lpToken = IIVault(_vault).lpTokenAddress();
    IERC20(lpToken).forceApprove(_vault, _shares);

    (_claimableEpoch, _year, _month, _day) = IIVault(_vault).requestRedeem(
        _shares,
        address(this)
    );
    emit RedeemRequested(_vault, _shares, _claimableEpoch, _year, _month,
    _day);
}
```

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit 87c1e39a ↗.

3.2. Both `getAgentVaultId` and `getVaultId` do not validate that the parsed ID is nonzero

Target	PaymentReferenceParser		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Both `getAgentVaultId` and `getVaultId` in PaymentReferenceParser parse a `uint16` identifier from a payment reference but do not verify that the result is nonzero. Because `AgentVaultsFacet.addAgentVaults` enforces `require(agentVaultId > 0, ...)` and `VaultsFacet.addVaults` enforces `require(vaultId > 0, ...)`, ID 0 can never be registered and is therefore always invalid.

As of the time of writing, the callers of `getAgentVaultId` and `getVaultId` reject zero IDs indirectly — `AgentVaults.getAgentVaultAddress` looks up the unregistered ID in a mapping, receives `address(0)`, and reverts via `require(_agentVault != address(0), ...)`. Likewise, `Vaults.getVaultAddress` reverts with `require(_vault != address(0), ...)`. As a result, a zero ID cannot propagate into downstream logic.

Adding an explicit `require(id > 0, ...)` inside each parser function would be beneficial

- for consistency — other parser helpers (`getValue`, `getAddress`) already validate parsed outputs;
- for defense in depth — future callers may not apply equivalent downstream checks; and
- for an earlier revert — reverting at the parser level avoids unnecessary gas spent.

Impact

No exploit path was identified. Existing downstream checks cause invalid zero IDs to revert. The risk is limited to future code paths that may call these functions without an equivalent guard.

Recommendations

Add nonzero validation to `getAgentVaultId` and `getVaultId` in PaymentReferenceParser, mirroring the pattern used by `getValue` and `getAddress`:

```
function getAgentVaultId(bytes32 _paymentReference) internal pure returns (
    uint256) {
function getAgentVaultId(bytes32 _paymentReference) internal pure returns (
```

```
    uint256 _agentVaultId) {
    // bytes 12-13: agent vault id
    return (uint256(_paymentReference) >> 144) & ((uint256(1) << 16) - 1);
    _agentVaultId = (uint256(_paymentReference) >> 144) & ((uint256(1) << 16)
        - 1);
    require(_agentVaultId > 0, ...);
}

function getVaultId(bytes32 _paymentReference) internal pure returns (
    uint256) {
function getVaultId(bytes32 _paymentReference) internal pure returns (
    uint256 _vaultId) {
    // bytes 14-15: vault id
    return (uint256(_paymentReference) >> 128) & ((uint256(1) << 16) - 1);

    _vaultId = (uint256(_paymentReference) >> 128) & ((uint256(1) << 16) - 1);
    require(_vaultId > 0, ...);
}
```

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit `fdb32415` ↗.

4. Patch Review

The purpose of this section is to document the exact diffs of the codebase that were considered in scope for this audit.

As requested by Flare Network, we focused on the changes made between commit `1ee58c16` ↗ and the commit `01a51400` ↗.

4.1. Introduction of new contracts

The update introduced the Timelock library, along with the FacetBase and TimelockFacet contracts.

The Timelock library uses the ERC-7201 storage pattern to avoid storage collisions in a Diamond proxy architecture. It manages timelocked function calls that must wait a specified duration before execution. It contains the following functions:

- `checkOnlyOwner`, which ensures only the contract owner can perform certain operations.
- `recordTimelockedCall`, which records a function call to be executed after the timelock period.
- `beforeExecute`, which validates execution permissions before running a function call.
- `timeToExecute`, which checks whether enough time has passed to execute a call.

In the FacetBase contract, the `onlyOwner` modifier checks whether the caller is the owner, and the `onlyOwnerWithTimelock` modifier performs a similar owner check while adding the timelock mechanism. These functions use logic from the Timelock library.

The TimelockFacet contract is a Diamond facet that provides the public interface for managing and executing timelocked function calls. It works in conjunction with the Timelock library to implement a two-step execution pattern for sensitive operations. It contains the following functions and modifiers:

- `executeTimelockedCall`, which executes a previously scheduled timelocked call after the delay period has passed.
- `cancelTimelockedCall`, which allows the owner to cancel a previously scheduled timelocked call.
- `setTimelockDuration`, which changes the timelock duration for future scheduled calls.
- `getTimelockDurationSeconds`, which returns the current timelock duration in seconds.
- `getExecuteTimelockedCallTimestamp`, which returns the timestamp when a specific scheduled call can be executed.
- `_passReturnOrRevert`, which forwards the exact return data or revert reason from the executed timelocked call.

4.2. Changes in contract structure

These changes affect several contracts and libraries:

- All libraries that provide convenient storage access for facets now use the ERC-7201 standard for namespaced storage layouts.

- All facets inherit from the FacetBase contract and use its modifiers for access control and timelock restrictions instead of LibDiamond library.

Library changes

There are changes made to specific libraries.

These changes are related to **AgentVaults**:

- The `agentVaults` state was renamed to `agentVaultIdToAgentVaultAddress`.
- The new `agentVaultAddressToAgentVaultId` was added to map from agent vault address to agent vault ID.

These changes are related to **Instructions**:

- The `executeInstruction` function now calls `Vault.deposit()` with the `_instructionType` that corresponds to the `PersonalAccount.deposit()` change.
- The support claim withdrawal and redeem FXRP instruction were removed.
- The support claim and redeem FXRP instruction were removed.

These changes are related to **PaymentProofs**:

- The `sourceId` state used for payment verification was added.
- The `setSourceId` function, which allows setting the `sourceId`, was added.
- The `setPaymentProofValidityDuration` function now checks that `_proof.data.sourceId == state.sourceId` to prevent cross-chain replay attacks.

These changes are related to **PersonalAccount**:

- The `_vaultType` parameter was added to the `deposit` function. This supports only `_vaultType = 1` or `_vaultType = 2` (1 is Firelight, 2 is Upshift), which calls the corresponding vault type.
- The `requestRedeem` call within the `requestRedeem` function now returns `year`, `_month`, and `_day` instead of `_assets` and emits those values.

These changes are related to **Swap**:

- The `usdt0` state was renamed to `stableCoin`.
- The `wNatUsdt0PoolFeeTierPPM` state was renamed to `wNatStableCoinPoolFeeTierPPM`.
- The `usdt0FXrpPoolFeeTierPPM` state was renamed to `stableCoinFXrpPoolFeeTierPPM`.
- The `stableCoinUsdFeedId` state was added.
- The `wNatUsdFeedId` state was added.

This change is related to **UniswapV3**:

- The `executeSwap` function now supports swapping tokens with different decimals.

These changes are related to **Vaults**:

- The `vaults` state was renamed to `vaultIdToVaultInfo`.
- The new `vaultAddressToVaultId` was added to map from vault address to vault ID.

Contract changes

There are changes made to specific contracts.

These changes are related to **AgentVaultsFacet**:

- The `addAgentVaults` function now prevents adding an agent vault ID of zero.
- The `addAgentVaults` function verifies that both conditions are met before allowing the addition:
  - `state.agentVaultIdToAgentVaultAddress[agentVaultId] == address(0)`
  - `state.agentVaultAddressToAgentVaultId[agentVaultAddress] == 0`
- The `removeAgentVaults` and `getAgentVaults` functions now use the new state structure.

These changes are related to **InstructionsFacet**:

- The `executeDepositAfterMinting` function now calls `Vault.deposit()` with the `instructionType` that corresponds to the `PersonalAccount.deposit()` change.
- The `executeWithdrawal` function was removed.

These changes are related to **PaymentProofsFacet**:

- The `getSourceId` state used for payment verification was added.
- The `getSourceId` function, which allows getting the current `sourceId`, was added.

This change is related to **MasterAccountControllerInit**:

- The `_sourceId` parameter was added to the init function. This calls `PaymentProofs.setSourceId()` to set the source ID during initialization.

This change is related to **PersonalAccountsFacet**:

- The `_sourceId` parameter was added to the init function. This calls `PaymentProofs.setSourceId()` to set the source ID during initialization.

This change is related to **PersonalAccounts**:

- The `setPersonalAccountImplementation` function now checks that `_implementation.code.length > 0`, ensuring it can only be set to a contract address.

This change is related to **PersonalAccountsFacet**:

- Instead of internal logic, `PersonalAccounts.setPersonalAccountImplementation()` is used.

These changes are related to **SwapFacet**:

- The `FLR_USD_FEED_ID` constant was removed and replaced by `state.wNatUsdFeedId`.

- The `USDT_USD_FEED_ID` constant was removed and replaced by `state.stableCoinUsdFeedId`.
- All function logic was updated to use the new state variable names.

These changes are related to **VaultsFacet**:

- The `addVaults` function now prevents adding a vault ID of zero.
- Contract functions now use the new state and state names.

Additionally, the contract contract/diamond/DiamondCutFacet.sol has been moved to contract/smartAccounts/facets/DiamondCutFacet.sol.