Zellic

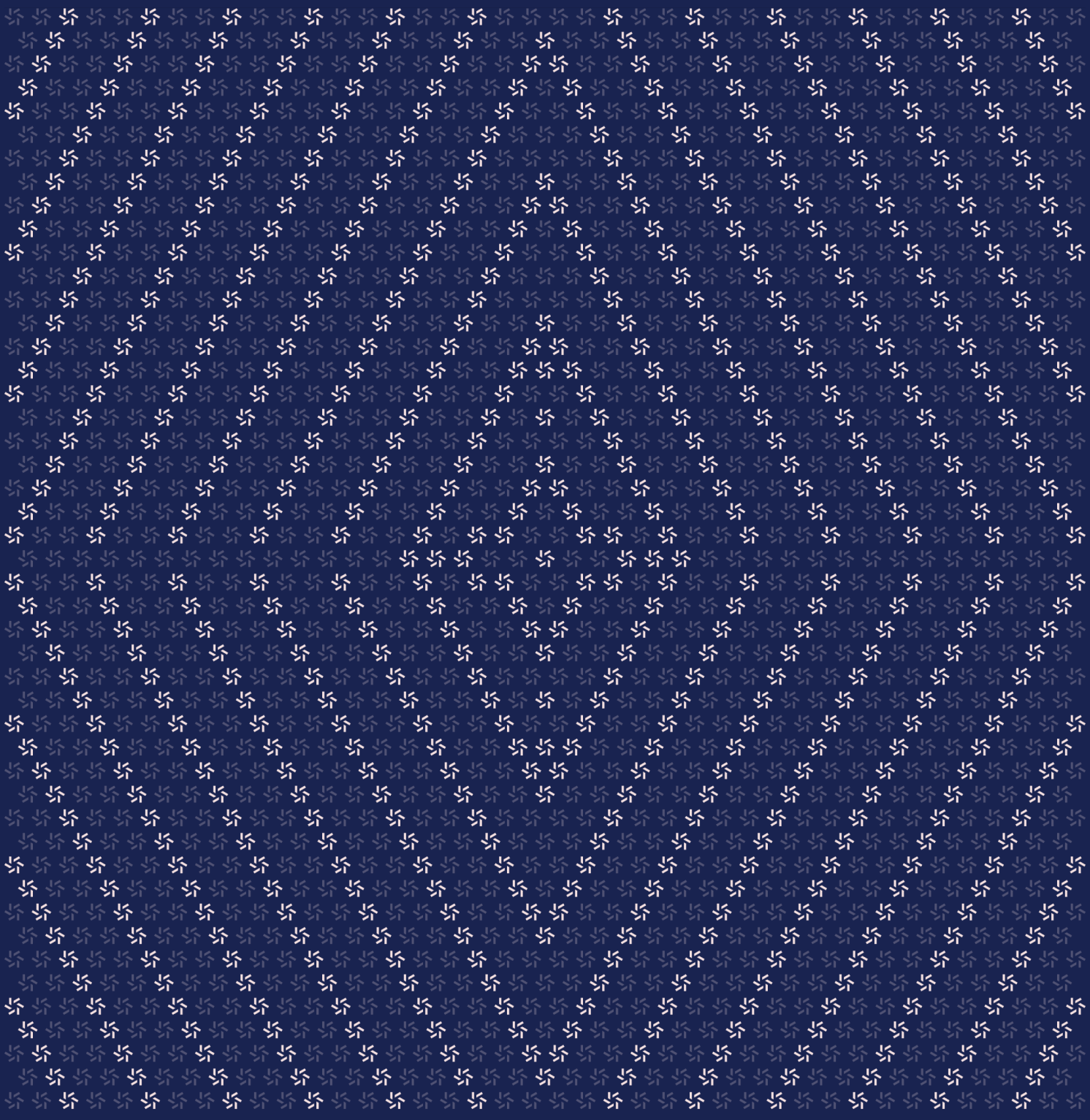
Prepared for
Luka Avbreht
Boštjan Kovač
Flare Network

Prepared by
Kritsada Dechawattana
Weipeng Lai
Zellic

November 26, 2025

Smart Accounts

Smart Contract Security Assessment



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 November 11th to November 18th, 2025. During this engagement, Zellic reviewed Smart Accounts's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following question:

- Is there a way for users to interact improperly with the system, assuming FDC works as expected?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

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

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Flare Network in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	3
Medium	2
Low	2
Informational	1

2. Introduction

2.1. About Smart Accounts

Flare Network contributed the following description of Smart Accounts:

> Smart Accounts provide an account abstraction layer for Flare's EVM that allows XRPL addresses to control and interact with EVM accounts directly, without leaving the XRPL environment.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood.

We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Smart Accounts Contracts

Type	Solidity
Platform	EVM-compatible
Target	smart-accounts
Repository	https://gitlab.com/flarenetwork/smart-accounts/flare-smart-accounts ↗
Version	`237d5cafe5d1a495b18a85dd72b46d412f21dd33`

Programs	`facets/*.sol`
	`implementation/*.sol`
	`library/*.sol`
	`proxy/PersonalAccountProxy.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.8 person-weeks. The assessment was conducted by two consultants over the course of six calendar days.

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 ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 11, 2025	Start of primary review period
November 12, 2025	Kick-off call
November 18, 2025	End of primary review period

3. Detailed Findings

3.1. Risk of nonconsensual execution on upgraded implementation

Target	PersonalAccount		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	High

Description

The PersonalAccount contract uses a beacon proxy-upgrade pattern where all existing personal account proxies automatically point to a new implementation when `PersonalAccountsFacet::setPersonalAccountImplementation()` is called by governance.

The issue is that XRPL users have no mechanism to specify which PersonalAccount implementation version they consent to having execute their transaction. When a user sends an XRPL transaction, they implicitly agree to the current implementation's behavior. However, if governance upgrades the PersonalAccount implementation before their transaction proof is submitted on Flare, their transaction will execute on a different implementation than they reviewed and agreed to use.

The upgraded implementation may change contract behavior by

- modifying function signatures or behavior for existing instruction types,
- introducing new validation logic that causes previously valid transactions to revert, or
- changing state handling in ways that produce unexpected outcomes.

This represents a consent violation because

- users craft XRPL payment references based on current implementation behavior,
- XRPL transactions are irreversible once sent, and
- users pay nonrefundable instruction fees up-front.

When `InstructionsFacet::executeInstruction()` processes the XRPL proof, it executes against whatever implementation the beacon currently points to, regardless of what version existed when the user initiated their XRPL transaction.

Impact

The system forces users to trust that future upgrades will not be malicious or buggy. Users cannot make a one-time trust decision about a specific implementation — they must continuously trust governance's upgrade decisions even after their transaction is in-flight. XRPL users may face several risks when upgrades occur:

- Valid XRPL transactions crafted for implementation V1 may revert when executed against V2, causing users to lose their instruction fees paid on XRPL without any state change on Flare.
- Since XRPL transactions are irreversible and users pay instruction fees up-front, they cannot recover these fees if an upgrade makes their transaction incompatible. The payment proof is consumed (marked as used in `usedTransactionIds`), preventing reexecution even after a rollback.
- A buggy upgrade could intentionally or accidentally make certain instruction types always revert, effectively freezing user funds in personal accounts until another upgrade is deployed.

Recommendations

For **short-term mitigation**, establish governance processes and timelock mechanisms without modifying core protocol logic, along with additional governance requirements:

- Announce every upgrade with a detailed changelog.
- Establish off-chain communication channels.
- Use governance commits to maintain backward compatibility for all payment reference formats.
- Maintain ability to quickly roll back to the previous implementation if issues are detected.

For **long-term mitigation**, modify the protocol to allow users to specify which PersonalAccount implementation version they consent to execute their transaction.

We recommend combining both approaches — implement version targeting for technical enforcement while maintaining governance upgrade timelocks as defense in depth. This provides multiple layers of protection and gives users both technical control and advance notice.

Remediation

This issue has been acknowledged by Flare Network, and fixes were implemented in the following commits:

- `cc8fb412` ↗
- `ecddae6a` ↗

3.2. Token decimal mismatch causes `swapWNatForUsdt0` to revert

Target	UniswapV3		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The UniswapV3 library's `executeSwap` function computes `minAmountOut` without accounting for differing decimals between `_tokenIn` and `_tokenOut`. The function calculates the value using `_tokenIn`'s decimal scale but passes it to Uniswap V3 as if it were expressed in `_tokenOut` units.

In SwapFacet, the `swapWNatForUsdt0` function initiates a swap from WNAT (18 decimals) to USDT0 (six decimals). The `executeSwap` function calculates the expected output amount as follows:

```
uint256 expectedAmountOut = Math.mulDiv(_amountIn, valuesInWei[0],
    valuesInWei[1]);
```

This formula assumes that `_tokenIn` and `_tokenOut` have the same decimals (since `valuesInWei` are normalized prices). When `_tokenIn` has 18 decimals and `_tokenOut` has six, the calculated `expectedAmountOut` (and consequently `minAmountOut`) is overstated by a factor of 10^{12}. Since `minAmountOut` is derived directly from `expectedAmountOut`, this causes the `amountOutMinimum` parameter passed to the Uniswap router to be drastically overstated.

Impact

For `swapWNatForUsdt0`, the `amountOutMinimum` passed to the Uniswap V3 router is unrealistically high. As a result, the router will always revert the transaction because the swap cannot satisfy the requested minimum output amount. This renders the `swapWNatForUsdt0` function unusable.

Recommendations

Adjust the `expectedAmountOut` calculation in `executeSwap` to account for token decimal differences. Fetch the decimals for both tokens, and scale the result accordingly.

```
function executeSwap(
    address _uniswapV3Router,
    address _tokenIn,
    bytes21 _tokenInFeedId,
    address _tokenOut,
```

```
    bytes21 _tokenOutFeedId,
    uint24 _poolFeeTierPPM,
    uint24 _maxSlippagePPM
)
    internal
    returns (uint256 _amountIn, uint256 _amountOut)
{
    // [...]
    // Calculate minimum amount out based on max slippage
    uint256 expectedAmountOut = Math.mulDiv(_amountIn, valuesInWei[0],
    valuesInWei[1]);
    uint256 decimalsIn = IERC20Metadata(_tokenIn).decimals();
    uint256 decimalsOut = IERC20Metadata(_tokenOut).decimals();
    if (decimalsOut > decimalsIn) {
        expectedAmountOut = expectedAmountOut * (10 ** (decimalsOut -
            decimalsIn));
    } else if (decimalsIn > decimalsOut) {
        expectedAmountOut = expectedAmountOut / (10 ** (decimalsIn -
            decimalsOut));
    }
    uint256 minAmountOut = Math.mulDiv(expectedAmountOut,
    1e6 - _maxSlippagePPM, 1e6);
    // [...]
}
```

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit 61243921 ↗.

3.3. Unprotected entry points enable unauthorized actions

Target	InstructionsFacet, SwapFacet		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The Flare Smart Accounts system implements a one-to-one mapping where each XRPL address is assigned a unique PersonalAccount controlled exclusively by the user through payment transactions (intents) on the XRPL network.

The intended workflow requires users to submit intent transactions on XRPL, which operators relay to the MasterAccountController with valid proofs to execute corresponding actions.

However, several entry points in the MasterAccountController violate this control mechanism. The functions `reserveCollateral`, `executeWithdrawal`, `swapWNatForUsdt0`, and `swapUsdt0ForFAsset` are exposed as permissionless `external` methods that lack proof verification for underlying XRPL transactions.

Consequently, any user on the Flare network can invoke these functions by providing a target `_xrplAddress` parameter. This enables attackers to execute operations on a victim's PersonalAccount without their authorization or knowledge.

Impact

This may lead to the following impacts.

- **Forced unfavorable swaps.** Attackers can trigger `swapWNatForUsdt0` or `swapUsdt0ForFAsset` when market prices disadvantage the victim, forcing swaps without the victim's consent or knowledge. Moreover, attackers could profitably backrun or sandwich these forced swaps, capturing the price differential at the victim's expense.
- **Reputation damage through `reserveCollateral` abuse.** While an attacker must pay the reservation fee to execute the call, they can intentionally create a minting request via `reserveCollateral` without ever sending the required XRP to complete the minting process. These requests eventually default. Repeated attacks accumulate default records on victims' accounts, potentially damaging their reputation within the FAssets protocol.
- **Disrupted user operations.** Attackers can front-run permissionless operations to disrupt user flows that assume the operation has not been performed yet (see Finding 3.4. ↗).

- **Incompatibility with instruction fee design.** Without proof requirement, these functions also lack any check for instruction fees. If the protocol configures fees for these specific operations, users can bypass the fee payment entirely by calling these EVM entry points directly instead of submitting a fee-paying intent transaction on the XRPL network.

Recommendations

Enforce proof verification for all MasterAccountController entry points. Modify `reserveCollateral`, `executeWithdrawal`, `swapWNatForUsdt0`, and `swapUsdt0ForFAsset` to require valid `IPayment.Proof` parameters before execution. This modification aligns these functions with the intended intent-based architecture.

Alternatively, implement access control that restricts these functions to allowlisted operators. Only trusted entities should invoke these entry points on behalf of users. Also, users must be informed of the permitted behaviors for these operators.

Remediation

This issue has been acknowledged by Flare Network, and a partial fix was implemented in commit `79bdea06` ↗.

Flare Network provided the following response to this finding:

> Regarding reserveCollateral we think that it is not an issue as method is payable and off-chain integration will only handle calls triggered by operators. Removed executeWithdrawal method. We kept swap methods open. The main goal for smart accounts is currently to be able to participate in XRP staking (depositing into vaults). The only way how user can get WNat or USDT0 is in case of defaulted FXRP redemption. In that case the user is paid with agents's collateral plus redemption default premium. The executor/operators will in that case trigger immediate swap back to FXRP. The premium should cover the slippage set on the contract and extra fee as the user will have to request the FXRP redemtion again.

3.4. Griefing attack via front-running `executeWithdrawal` disrupts user operations

Target	InstructionsFacet		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The protocol supports atomic multistep operations through instruction commands.

- Instruction (1,4): Claim withdrawal from Firelight vault + immediately redeem FXRP
- Instruction (2,4): Claim from Upshift vault + immediately redeem FXRP

However, `InstructionsFacet::executeWithdrawal()` allows anyone to call it to claim on behalf of any XRPL address without making a transaction on XRPL and providing proof. This enables a griefing attack where malicious actors front-run users' intended withdrawal claims and redeem a combined operation — claiming withdrawal and immediately redeeming FXRP, corresponding to instruction types (1,4) or (2,4) — causing the claim to be executed prematurely and potentially disrupting the user's planned transaction flow.

Impact

Consider this scenario where a user attempts to claim and redeem but is disrupted by an attacker.

1. The user creates a transaction to execute instruction type (1,4) — claiming withdrawal from Firelight vault and immediately redeeming FXRP.
2. An attacker observes when the user's withdrawal becomes claimable.
3. The attacker front-runs by calling `executeWithdrawal` on behalf of the user. (The claimed FXRP goes to the user's PersonalAccount.)
4. The user's instruction type (1,4) transaction fails because `executeWithdrawal` was called prematurely.
5. The user must now call a separate redeem FXRP instruction, paying additional fees.

While this does not directly steal funds (the claimed FXRP goes to the correct PersonalAccount), it disrupts the user's intended workflow and forces them to pay additional redeem fees and gas costs.

Recommendations

Align `InstructionsFacet::executeWithdrawal()` with the secure pattern used in `InstructionsFacet::executeInstruction()` by requiring proof verification. If the function must remain permissionless for executor or relayer services, consider implementing an authorized executor role that restricts execution to third-party automation explicitly approved by the account owner.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit 10104301 ↗.

Flare Network provided the following response to this finding:

> Removed claim+redeem combination as claim can be executed also directly on the vault (at least on Upshift vault).

3.5. Missing contract-code verification

Target	PersonalAccountsFacet		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The `PersonalAccountsFacet::setPersonalAccountImplementation()` only validates that the new implementation address is not the zero address; it fails to verify that it actually contains contract code. This is because the personal-account system uses a BeaconProxy pattern where all existing proxies immediately delegate calls to the new implementation address set in the MasterAccountController (which acts as the beacon via the `implementation` function).

```
function setPersonalAccountImplementation(
    address _newImplementation
)
    external
{
    LibDiamond.enforceIsContractOwner();

    require(_newImplementation != address(0), InvalidPersonalAccountImplementat
        ion());
    PersonalAccounts.State storage state = PersonalAccounts.getState();
    state.personalAccountImplementation = _newImplementation;
    emit PersonalAccountImplementationSet(_newImplementation);
}
```

While this function only prevents setting the zero address, it does not verify that `_newImplementation` has bytecode deployed at that address. This creates a dangerous scenario where an address without code could be set as the implementation, causing all proxy delegatecalls to fail.

Impact

All existing personal-account proxies would become completely unusable. When any function is called on a personal account, the BeaconProxy delegates to `IBeacon(controllerAddress).implementation()`, which returns the invalid address. The delegatecall to an address without code will revert, bricking all personal accounts until the implementation address is updated to the correct one.

Recommendations

Add contract-code validation to ensure the implementation address contains deployed bytecode.

```
function setPersonalAccountImplementation(
    address _newImplementation
)
    external
{
    LibDiamond.enforceIsContractOwner();
    require(_newImplementation != address(0),
    InvalidPersonalAccountImplementation());

    require(_newImplementation.code.length > 0, InvalidPersonalAccountImplement
       ation());
    PersonalAccounts.State storage state = PersonalAccounts.getState();
    state.personalAccountImplementation = _newImplementation;
    emit PersonalAccountImplementationSet(_newImplementation);
}
```

Remediation

This issue has been acknowledged by Flare Network, and fixes were implemented in the following commits:

- 7b9e2ab0 ↗
- c0ade7ba ↗

3.6. Duplicate agent-vault address allows multiple IDs to reference the same vault

Target	AgentVaultsFacet		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `AgentVaultsFacet::addAgentVaults()` validates that agent-vault IDs are unique but does not verify that agent-vault addresses are unique across different IDs. This allows the contract owner to accidentally map multiple distinct `agentVaultId` values to the same `agentVaultAddress` or the registered agent vault. The function only checks that `state.agentVaults[agentVaultId] == address(0)` to prevent ID reuse; it does not perform a check that the agent-vault address is already registered. For example, an owner could call `addAgentVaults([1, 2], [0xAAA..., 0xAAA...])`, resulting in both ID 1 and ID 2 pointing to the same agent-vault address. Or they could repeatedly call `addAgentVaults([3], [0xAAA...])`, which would create more duplicate agent-vault addresses.

Impact

When `removeAgentVaults()` is called for one ID, the address remains accessible through other IDs. If the intent was to completely remove access to a specific agent vault (e.g., due to security concerns), the removal is incomplete and ineffective.

Recommendations

Implement a reversion on duplicated agent to ensure each address is only registered once.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit `4b78a5e6` ↗.

3.7. Incorrect parameter order in the `InstructionExecuted` event

Target	InstructionsFacet		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The `InstructionExecuted` event is emitted by `executeDepositAfterMinting` with incorrect parameter ordering.

The event is defined with the following parameter order:

```
event InstructionExecuted(
    address indexed personalAccount,
    bytes32 indexed transactionId,
    bytes32 indexed paymentReference,
    string xrplOwner,
    uint256 instructionId
);
```

However, in the `executeDepositAfterMinting` function, the event is emitted with `paymentReference` and `transactionId` swapped:

```
function executeDepositAfterMinting(
    uint256 _collateralReservationId,
    IPayment.Proof calldata _proof,
    string calldata _xrplAddress
)
    external
{
    //[...]
    emit InstructionExecuted(
        address(personalAccount),
        paymentReference,
        transactionId,
        _xrplAddress,
        PaymentReferenceParser.getInstructionId(paymentReference)
    );
}
```

Impact

Off-chain indexers or listeners relying on the event parameters by position will receive swapped values for `transactionId` and `paymentReference`.

Recommendations

Swap the arguments in the `emit` statement to match the event definition:

```
function executeDepositAfterMinting(
    uint256 _collateralReservationId,
    IPayment.Proof calldata _proof,
    string calldata _xrplAddress
)
    external
{
    // [...]
    emit InstructionExecuted(
        address(personalAccount),
        paymentReference,
        transactionId,
        transactionId,
        paymentReference,
        _xrplAddress,
        PaymentReferenceParser.getInstructionId(paymentReference)
    );
}
```

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit ab5245d1 ↗.

3.8. Missing `chainId` check in `verifyPayment`

Target	PaymentProofs		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `verifyPayment` function in the PaymentProofs library validates payment proofs but fails to check the `_proof.data.sourceId`. This field identifies the underlying blockchain network (e.g., XRPL) where the transaction occurred.

As of the time of writing, this omission is not exploitable because the Flare Data Connector (FDC) for XRPL is configured to only verify proofs from the XRPL mainnet. However, if the FDC is updated in the future to support additional networks that share the same address format, the contract would be unable to distinguish between payments made on XRPL and those made on other networks.

In that future scenario, a user could perform a transaction on a cheaper network (where fees or token value are lower than XRPL) and submit its proof to the MasterAccountController. Because `verifyPayment` does not validate the `sourceId`, the contract would accept the proof from the cheaper network as valid payment on XRPL for instruction fees.

Impact

There is no immediate impact under the current FDC configuration. However, this represents a latent vulnerability. If the FDC expands support to other networks in the future, the system will become vulnerable due to the missing `chainId` check. If a secondary network (such as the XRPL Testnet or a low-value fork) is supported, an attacker could pay fees using tokens with little to no value. The contract, failing to distinguish the source chain, would treat these as valid XRPL payments. This results in direct financial loss to the protocol, as services would be rendered without receiving the requisite value.

Recommendations

To future-proof the protocol and ensure payment transactions originate from the intended network, verify that the `sourceId` matches the chain ID for XRPL.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit `433bf825`.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Unresponsive executor risk

In the typical FXRP minting flow, after requesting collateral reservation and paying the agent on XRPL, either the minter or the executor must invoke the AssetManager's `executeMinting` function to complete the minting of FXRP to the minter. (While the agent-vault owner can also call this function to complete the minting, they are not obligated to do so.)

However, in the Flare Smart Accounts system, the PersonalAccount contract does not expose a way to call the AssetManager's `executeMinting` function. This means the PersonalAccount contract itself cannot complete the minting as the minter. Therefore, it is critical for the executor to call `executeMinting` to finalize the minting for the PersonalAccount. If the executor becomes unresponsive, users may not receive their entitled FXRP tokens.

Consequently, it is essential to ensure that the executor configured in the MasterAccountController remains responsive and reliable.

Flare Network provided the following response to this discussion point:

> It is in agent's interest to execute the minting in case of unresponsive executor in order to free locked collateral.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: AgentVaultsFacet.sol

Function: `addAgentVaults(uint256[] _agentVaultIds, address[] _agentVaultAddresses)`

This function adds agent vaults to the system by mapping vault IDs to their addresses. It validates that vault IDs are unique. It is only callable by the contract owner.

Inputs

- `_agentVaultIds`
  - **Control**: Full control.
  - **Constraints**: Must be unique.
  - **Impact**: The added agent-vault ID can be used in the instruction to interact with the agent vault.
- `_agentVaultAddresses`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address.
  - **Impact**: The added agent-vault address can be used in the instruction to interact with the agent vault.

Branches and code coverage

Intended branches

- Add agent vaults successfully.
  - ☑ Test coverage
- Emit the `AgentVaultAdded` event for each added agent vault.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.

☑ Negative test

- Revert if agent-vault IDs and address lengths are not the same.

☑ Negative test

- Revert if agent-vault addresses are zero addresses.

☑ Negative test

- Revert if agent-vault IDs are not unique.

☑ Negative test

- Revert if agent-vault addresses are not available.

☑ Negative test

- Revert if adding agent vaults with duplicate agent-vault addresses.

☐ Negative test

Function: `getAgentVaults()`

This function returns all agent-vault IDs and their corresponding addresses. It is a view function accessible to anyone.

Function: `removeAgentVaults(uint256[] _agentVaultIds)`

This function removes agent vaults from the system by deleting the ID-to-address mappings. It is only callable by the contract owner.

Inputs

- `_agentVaultIds`
  - **Control**: Full control.
  - **Constraints**: Must be valid agent-vault IDs that have been added.
  - **Impact**: The removed agent-vault ID can no longer be used in the instruction to interact with the agent vault.

Branches and code coverage

Intended branches

- Remove agent vaults successfully.

☑ Test coverage

- Emit the `AgentVaultRemoved` event for each removed agent vault.

☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test
- Revert if agent-vault IDs are not valid agent-vault IDs that have been added.
  - ☑ Negative test

5.2. Module: ExecutorsFacet.sol

Function: `getExecutorInfo()`

This function returns the current executor address and executor fee. It is a view function accessible to anyone.

Function: `setExecutorFee(uint256 _executorFee)`

This function configures the fee amount charged for executor operations. It is only callable by the contract owner.

Inputs

- `_executorFee`
  - **Control**: Full control.
  - **Constraints**: Must be greater than zero.
  - **Impact**: The executor fee can be used to charge for executor operations.

Branches and code coverage

Intended branches

- Set the executor fee successfully.
  - ☑ Test coverage
- Emit the `ExecutorFeeSet` event.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test

- Revert if `_executorFee` is zero.
  - ☑ Negative test

Function call analysis

- `Executors.setExecutorFee(_executorFee)`
  - **What is controllable?** `_executorFee`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert.

Function: `setExecutor(address payable _executor)`

This function sets the executor address authorized to perform crucial operations. It is only callable by the contract owner.

Inputs

- `_executor`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address.
  - **Impact**: The executor address can perform crucial operations.

Branches and code coverage

Intended branches

- Set the executor successfully.
  - ☑ Test coverage
- Emit the `ExecutorSet` event.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test
- Revert if `_executor` is the zero address.
  - ☑ Negative test

Function call analysis

- `Executors.setExecutor(_executor)`
  - **What is controllable?** `_executor`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert.

5.3. Module: InstructionFeesFacet.sol

Function: `getDefaultInstructionFee()`

This function returns the default instruction fee. It is a view function accessible to anyone.

Function: `getInstructionFee(uint256 _instructionId)`

This function returns the instruction fee for a specific instruction ID. It is a view function accessible to anyone.

Inputs

- `_instructionId`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: The instruction ID that needs to get the instruction fee.

Function: `removeInstructionFees(uint256[] _instructionIds)`

This function removes custom fees for specific instruction IDs, reverting them to the default fee. It is only callable by the contract owner.

Inputs

- `_instructionIds`
  - **Control**: Full control.
  - **Constraints**: Must be valid instruction IDs that have been set.
  - **Impact**: The custom instruction fee can be removed for a specific instruction.

Branches and code coverage

Intended branches

- Remove instruction fees successfully.
  - ☑ Test coverage
- Emit the `InstructionFeeRemoved` event for each removed instruction fee.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test
- Revert if instruction IDs are not valid instruction IDs that have been set.
  - ☐ Negative test

Function: `setDefaultInstructionFee(uint256 _defaultInstructionFee)`

This function sets the default fee for instructions that do not have specific fees configured. It is only callable by the contract owner.

Inputs

- `_defaultInstructionFee`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: The default instruction fee can be used to charge for instructions that do not have specific fees configured.

Branches and code coverage

Intended branches

- Set the default instruction fee successfully.
  - ☑ Test coverage
- Emit the `DefaultInstructionFeeSet` event.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.

☑ Negative test

Function call analysis

- `InstructionFees.setDefaultInstructionFee(_defaultInstructionFee)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** There are not any problems here.

Function: `setInstructionFees(uint256[] _instructionIds, uint256[] _fees)`

This function sets custom fees for specific instruction IDs. It stores as 1-based values to distinguish from unset (0). It is only callable by the contract owner.

Inputs

- `_instructionIds`
  - **Control**: Full control.
  - **Constraints**: Must be unique.
  - **Impact**: The instruction id that specific the instruction fee.
- `_fees`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: The custom instruction fee can be used to charge for a specific instruction.

Branches and code coverage

Intended branches

- Set instruction fees successfully.
  - ☑ Test coverage
- Emit the `InstructionFeeSet` event for each set instruction fee.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test
- Revert if instruction IDs and fee lengths are not the same.
  - ☑ Negative test

5.4. Module: InstructionsFacet.sol

Function: `executeDepositAfterMinting(uint256 _collateralReservationId, IPayment.Proof _proof, string _xrplAddress)`

This function deposits minted FXRP into a vault after successful minting. This allows anyone to deposit minted FXRP into a vault as long as the payment proof is valid.

Inputs

- `_collateralReservationId`
  - **Control**: Full control.
  - **Constraints**: Must be a valid collateral-reservation ID.
  - **Impact**: The collateral-reservation ID will be used to deposit minted FXRP into a vault.
- `_proof`
  - **Control**: Full control.
  - **Constraints**: Must be a valid payment proof.
  - **Impact**: The payment proof will be used to deposit minted FXRP into a vault.
- `_xrplAddress`
  - **Control**: Full control.
  - **Constraints**: Must be a valid XRPL address.
  - **Impact**: The XRPL address will be used to deposit minted FXRP into a vault.

Branches and code coverage

Intended branches

- The deposit is executed successfully.
  - ☑ Test coverage
- The personal account should receive shares after the deposit.
  - ☐ Test coverage
- Emit the `Deposited` event after the deposit.

- ☑ Test coverage
- Emit the `InstructionExecuted` event after the deposit.
  - ☑ Test coverage

Negative behavior

- Revert if the payment reference is not a valid payment reference with instruction type 1 or 2 and instruction command zero.
  - ☑ Negative test
- Revert if the payment reference is not a valid payment reference with the transaction ID zero.
  - ☑ Negative test
- Revert if the collateral-reservation ID is not found in the mapping.
  - ☑ Negative test
- Revert if reservation info status is not successful.
  - ☑ Negative test
- Revert if the payment proof is not valid.
  - ☑ Negative test
- Revert if the minter and amount do not match.
  - ☑ Negative test
- Revert if the transaction is already executed.
  - ☐ Negative test

Function call analysis

- `Vault.deposit(personalAccount, vault, amount)`
  - **What is controllable?** `personalAccount`, `vault`, and `amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert.

Function: `executeInstruction(IPayment.Proof _proof, string _xrplAddress)`

This is a generic instruction executor that verifies the payment proof and routes to the appropriate instruction handler based on type and command.

Inputs

- `_proof`
  - **Control**: Full control.
  - **Constraints**: Must be a valid payment proof.
  - **Impact**: The payment proof will be used to execute the instruction.
- `_xrplAddress`
  - **Control**: Full control.
  - **Constraints**: Must be a valid XRPL address.
  - **Impact**: The XRPL address will be used to execute the instruction.

Branches and code coverage

Intended branches

- The payment proof is valid.
  - ☑ Test coverage
- The personal account is created successfully if it does not exist.
  - ☑ Test coverage
- The instruction is executed successfully.
  - ☑ Test coverage
- Emit the `InstructionExecuted` event after the instruction execution.
  - ☑ Test coverage

Negative behavior

- Revert if the received amount is less than the instruction fee.
  - ☑ Negative test
- Revert if the payment proof is not valid.
  - ☑ Negative test
- Revert if the transaction is already executed.
  - ☑ Negative test
- Revert if the instruction is invalid.
  - ☑ Negative test

Function call analysis

- `Instructions.executeInstruction(instructionType, instructionCommand, paymentReference, personalAccount)`
    - **What is controllable?** `instructionType`, `instructionCommand`, `paymentReference`, and `personalAccount`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert.

Function: `executeWithdrawal(string _xrplAddress, uint256 _vaultId, uint256 _epoch)`

This function claims a withdrawal from a vault (Firelight or Upshift) for a user's personal account at a specific epoch/date. This allows anyone to claim a withdrawal from a vault on behalf of a `_xrplAddress`.

Inputs

- `_xrplAddress`
    - **Control**: Full control.
    - **Constraints**: Must be a valid XRPL address.
    - **Impact**: The XRPL address will be used to claim a withdrawal from a vault.
- `_vaultId`
    - **Control**: Full control.
    - **Constraints**: Must be a valid vault ID.
    - **Impact**: The vault ID will be used to claim a withdrawal from a vault.
- `_epoch`
    - **Control**: Full control.
    - **Constraints**: Must be a valid epoch.
    - **Impact**: The epoch/date will be used to claim a withdrawal from a vault.

Branches and code coverage

Intended branches

- The withdrawal is claimed successfully from the Firelight vault.
    - ☑ Test coverage
- The withdrawal is claimed successfully from the Upshift vault.

- ☑ Test coverage
- Emit the `WithdrawalExecuted` event after the withdrawal execution.
  - ☑ Test coverage

Negative behavior

- Revert if the vault is not found.
  - ☑ Negative test
- Revert if the vault type is invalid.
  - ☑ Negative test

Function call analysis

- `Vault.claimWithdrawal(personalAccount, vaultInfo.vaultAddress, _epoch)`
  - **What is controllable?** `personalAccount`, `vaultInfo.vaultAddress`, and `_epoch`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert.
- `Vault.claim(personalAccount, vaultInfo.vaultAddress, _epoch)`
  - **What is controllable?** `personalAccount`, `vaultInfo.vaultAddress`, and `_epoch`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert.

Function: `getTransactionIdForCollateralReservation(uint256 _collateralReservationId)`

This function returns the transaction ID associated with a collateral reservation ID.

Inputs

- `_collateralReservationId`
  - **Control**: N/A.
  - **Constraints**: N/A.

- **Impact**: The collateral reservation ID that needs to get the mapped transaction ID.

Function: `isTransactionIdUsed(byte[32] _transactionId)`

This function checks if a transaction ID has already been used to prevent replay attacks.

Inputs

- `_transactionId`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: The transaction ID that needs to get the used status.

Function: `reserveCollateral(string _xrplAddress, byte[32] _paymentReference, byte[32] _transactionId)`

This function reserves collateral for minting FXRP and maps collateral reservation ID to transaction ID. This allows anyone to reserve collateral for minting FXRP.

Inputs

- `_xrplAddress`
  - **Control**: Full control.
  - **Constraints**: Must be a valid XRPL address.
  - **Impact**: The XRPL address will be used to reserve collateral.
- `_paymentReference`
  - **Control**: Full control.
  - **Constraints**: Must be a valid payment reference with instruction type 1 or 2 and instruction command 0.
  - **Impact**: The payment reference will be used to reserve collateral.
- `_transactionId`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: The reservation ID will be mapped to the transaction ID.

Branches and code coverage

Intended branches

- The collateral is reserved successfully.
  - ☑ Test coverage
- The personal account is created successfully if it does not exist.
  - ☑ Test coverage
- Emit the `CollateralReserved` event.
  - ☑ Test coverage
- Map collateral-reservation ID to transaction ID successfully.
  - ☑ Test coverage

Negative behavior

- Revert if `_paymentReference` is not a valid payment reference with instruction type 0, 1, or 2 and instruction command 0.
  - ☑ Negative test
- Revert if `_transactionId` is zero.
  - ☑ Negative test
- Revert if `_paymentReference` is not a valid payment reference with invalid agent-vault ID.
  - ☑ Negative test
- Revert if `_paymentReference` is not a valid payment reference with value zero.
  - ☑ Negative test

Function call analysis

- `FXrp.reserveCollateral(_personalAccount, _agentVault, _lots, _transactionId, _paymentReference, _xrplAddress)`
  - **What is controllable?** `_personalAccount`, `_agentVault`, `_lots`, `_transactionId`, `_paymentReference`, and `_xrplAddress`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is the collateral-reservation ID that will be used to map to the transaction ID.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert.

5.5. Module: MasterAccountControllerInit.sol

Function: `init(address payable _executor, uint256 _executorFee, uint256 _paymentProofValidityDurationSeconds, uint256 _defaultInstructionFee, address _personalAccountImplementation)`

This is a diamond initialization function called during deployment. It sets up the ERC-165 interfaces and initializes all facet state variables (executor, fees, payment proof duration, personal-account implementation).

Inputs

- `_executor`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address.
  - **Impact**: The executor address can perform crucial operations.
- `_executorFee`
  - **Control**: Full control.
  - **Constraints**: Must be greater than zero.
  - **Impact**: The executor fee can be used to charge for executor operations.
- `_paymentProofValidityDurationSeconds`
  - **Control**: Full control.
  - **Constraints**: Must be greater than zero.
  - **Impact**: The payment-proof–validity duration can be used to limit the validity of payment proofs.
- `_defaultInstructionFee`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: The default instruction fee can be used to charge for instructions when no specific fee is set.
- `_personalAccountImplementation`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address.
  - **Impact**: The personal-account implementation is used as the implementation contract for all personal accounts.

Branches and code coverage

Intended branches

- All state variables are set successfully.
  - ☐ Test coverage

Negative behavior

- Revert if `_executor` is the zero address.
  - ☐ Negative test
- Revert if `_executorFee` is zero.
  - ☐ Negative test
- Revert if `_paymentProofValidityDurationSeconds` is zero.
  - ☐ Negative test
- Revert if `_personalAccountImplementation` is the zero address.
  - ☐ Negative test

Function call analysis

- `Executors.setExecutor(_executor)`
  - **What is controllable?** `_executor`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert.
- `Executors.setExecutorFee(_executorFee)`
  - **What is controllable?** `_executorFee`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert.
- `PaymentProofs.setPaymentProofValidityDuration(_paymentProofValidityDurationSeconds)`
  - **What is controllable?** `_paymentProofValidityDurationSeconds`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert.
- `InstructionFees.setDefaultInstructionFee(_defaultInstructionFee)`
  - **What is controllable?** `_defaultInstructionFee`.

- **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
- **What happens if it reverts, reenters or does other unusual control flow?** The function will revert.

- `PersonalAccounts.setPersonalAccountImplementation(_personalAccountImplementation)`
  - **What is controllable?** `_personalAccountImplementation`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert.

5.6. Module: PaymentProofsFacet.sol

Function: `getPaymentProofValidityDurationSeconds()`

This function returns the current payment-proof–validity duration in seconds.

Function: `setPaymentProofValidityDuration(uint256 _paymentProofValidityDurationSeconds)`

This function sets the time window during which payment proofs are considered valid. It is only callable by the contract owner.

Inputs

- `_paymentProofValidityDurationSeconds`
  - **Control**: Full control.
  - **Constraints**: Must be greater than zero.
  - **Impact**: The payment-proof–validity duration can be used to limit the validity of payment proofs.

Branches and code coverage

Intended branches

- The payment-proof–validity duration is set successfully.
  - ☑ Test coverage
- Emit the `PaymentProofValidityDurationSecondsSet` event after the

payment-proof–validity duration is set.

- ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test
- Revert if the payment-proof–validity duration is zero.
  - ☑ Negative test

Function call analysis

- `PaymentProofs.setPaymentProofValidityDuration(_paymentProofValidityDurationSeconds)`
  - **What is controllable?** `_paymentProofValidityDurationSeconds`.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert.

5.7. Module: PersonalAccountsFacet.sol

Function: `getPersonalAccount(string _xrplOwner)`

This function returns the personal-account address for a given XRPL address. If not deployed yet, it returns the computed deterministic address.

Inputs

- `_xrplOwner`
  - **Control**: N/A.
  - **Constraints**: N/A.
  - **Impact**: The xrpl address that needs to get the personal account.

Function: `implementation()`

This function returns the current personal-account implementation address (implements the IBeacon interface).

Function: `setPersonalAccountImplementation(address _newImplementation)`

This function sets the implementation address for personal-account proxies (beacon pattern). It is only callable by the contract owner.

Inputs

- `_newImplementation`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address.
  - **Impact**: The personal-account implementation used as the implementation contract for all personal accounts.

Branches and code coverage

Intended branches

- The new implementation address is set successfully.
  - ☑ Test coverage
- Emit the `PersonalAccountImplementationSet` event after the new implementation address is set.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test
- Revert if the new implementation address is the zero address.
  - ☑ Negative test

5.8. Module: VaultsFacet.sol

Function: `addVaults(uint256[] _vaultIds, address[] _vaultAddresses, uint8[] _vaultTypes)`

This function adds vaults to the system with their IDs, addresses, and types (1 = Firelight, 2 = Upshift). It is only callable by the contract owner.

Inputs

- `_vaultIds`

    - **Control**: Full control.
    - **Constraints**: Must be unique — the length of the array must be the same as the length of the vault addresses.
    - **Impact**: The added vault IDs can be used in the instruction to interact with the vault.
- `_vaultAddresses`
    - **Control**: Full control.
    - **Constraints**: Must not be the zero address — the length of the array must be the same as the length of the vault IDs.
    - **Impact**: The added vault addresses can be used in the instruction to interact with the vault.
- `_vaultTypes`
    - **Control**: Full control.
    - **Constraints**: Must be 1 or 2.
    - **Impact**: The added vault type can be used to determine the type of vault.

Branches and code coverage

Intended branches

- Add vaults successfully.
    - ☑ Test coverage
- Emit the `VaultAdded` event for each added vault.
    - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
    - ☑ Negative test
- Revert if the vault IDs and address lengths are not the same.
    - ☑ Negative test
- Revert if the vault addresses are zero addresses.
    - ☑ Negative test
- Revert if the vault types are not 1 or 2.
    - ☑ Negative test
- Revert if the vault IDs are not unique.
    - ☑ Negative test

Function: `getVaults()`

This function returns all vault IDs, addresses, and types.

5.9. Module: XrplProviderWalletsFacet.sol

Function: `addXrplProviderWallets(string[] _xrplProviderWallets)`

This function adds XRPL provider wallet addresses to the whitelist by using a 1-based index to store the wallet in the array. It is only callable by the contract owner.

Inputs

- `_xrplProviderWallets`
  - **Control**: Full control.
  - **Constraints**: Must not be the zero address — the length of the array must be greater than zero.
  - **Impact**: The added XRPL provider wallet addresses can be used to validate the payment proofs.

Branches and code coverage

Intended branches

- Add XRPL provider wallet addresses successfully.
  - ☑ Test coverage
- Emit the `XrplProviderWalletAdded` event for each added XRPL provider wallet address.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☑ Negative test
- Revert if the XRPL provider wallet addresses are zero addresses.
  - ☑ Negative test
- Revert if the XRPL provider wallet addresses already exist.
  - ☑ Negative test

Function: `getXrplProviderWallets()`

This function returns all whitelisted XRPL provider wallet addresses.

Function: `removeXrplProviderWallets(string[] _xrplProviderWallets)`

This function removes XRPL provider wallet addresses from the whitelist. It uses swap-and-pop for efficient array removal while maintaining hash-index integrity. It is only callable by the contract owner.

Inputs

- `_xrplProviderWallets`
  - **Control**: Full control.
  - **Constraints**: Must exist in the whitelist.
  - **Impact**: The removed XRPL provider wallet addresses can no longer be used to validate the payment proofs.

Branches and code coverage

Intended branches

- Remove XRPL provider wallet addresses successfully.
  - ☑ Test coverage
- Emit the `XrplProviderWalletRemoved` event for each removed XRPL provider wallet address.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - ☐ Negative test
- Revert if some of the XRPL provider wallet addresses do not exist in the whitelist.
  - ☐ Negative test

6. Assessment Results

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

6.1. 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.