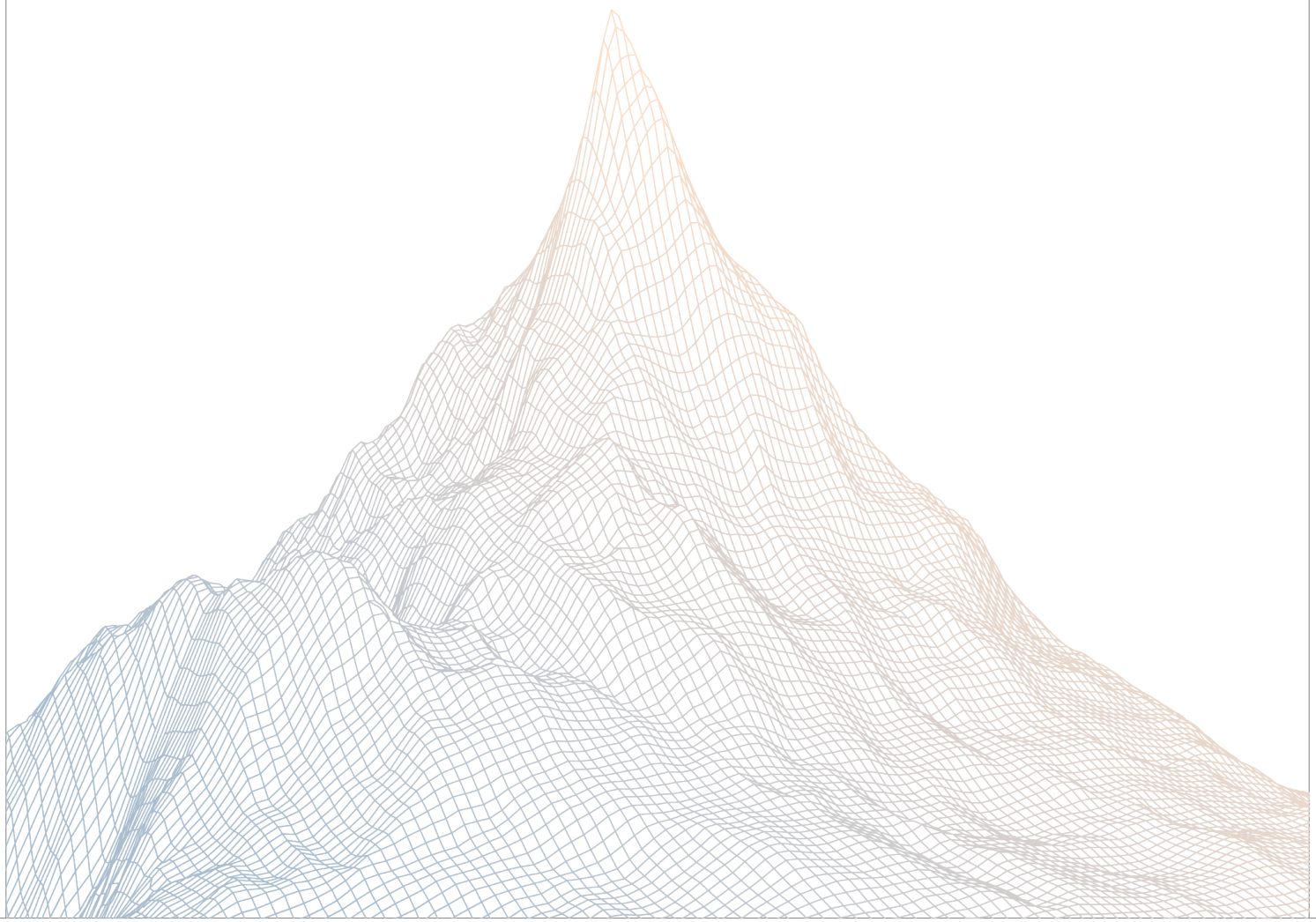
Zenith

Sonz

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: November 6th to November 7th, 2025

AUDITED BY: adriro
montecristo

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Sonz

A deeply social mafia game ecosystem on Telegram, thriving on its social graph.

Here, Respect is currency, and Families rise and fall by cunning alliances, and engage in real, on-chain Wars.

2.2 Scope

The engagement involved a review of the following targets:

Target	telemafia-contracts
Repository	https://github.com/sonz-ai/telemafia-contracts
Commit Hash	`4cd2289b01077695af30fb430b7b0a2b2c7fdb2b`
Files	`FamilyVaultUpgradeable.sol` `VaultFactory.sol`

2.3 Audit Timeline

November 6, 2025	Audit start
November 7, 2025	Audit end
November 12, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	2
Informational	3
Total Issues	**7**

3

Findings Summary

ID	Description	Status
M-1	Vault ownership transfer can easily break	Resolved
M-2	Family owner can update vault to arbitrary smart contract	Resolved
L-1	Multiple pending implementations can lead to confusing upgrades	Resolved
L-2	Removing a token from whitelist can be DOS'ed by 1-wei donation	Resolved
I-1	Implement a proxy registry in VaultFactory	Resolved
I-2	Registry reference in VaultFactory can be immutable	Resolved
I-3	Incorrect storage gap calculation	Resolved

4

Findings

4.1 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] Vault ownership transfer can easily break

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- FamilyVaultUpgradeable.sol

Description:

When transferring ownership of a vault, the registry calls `setAdmin()` inside a `try/catch` block and ignores any error raised by the implementation.

```
if (_families[familyId].vault ≠ address(0)) {
    // Use try-catch to prevent malicious vault from blocking ownership
    transfer
    try IFamilyVault(_families[familyId].vault).setAdmin(newOwner) {
        // Success - vault admin updated
    } catch {
        // Vault admin update failed, but ownership transfer still succeeds
        emit VaultAdminUpdateFailed(familyId, _families[familyId].vault);
    }
}
```

The implementation of `setAdmin()` raises when member counts are not within the thresholds.

```
function setAdmin(address newAdmin) external {
    require(msg.sender == registry, "Only registry");
    require(newAdmin ≠ address(0), "Invalid admin");

    // Capture current admins and revoke them
    // Note: This design assumes a single admin per vault. If multiple admins
    // are granted manually, this could consume more gas. Consider revoking
```

```
    // manually before transferring if >10 admins exist.
    uint256 count = getRoleMemberCount(DEFAULT_ADMIN_ROLE);
    require(count <= 10, "Too many admins, manual cleanup required");

    uint256 treasurerCount = getRoleMemberCount(TREASURER_ROLE);
    require(treasurerCount <= 10, "Too many treasurers");

    uint256 warManagerCount = getRoleMemberCount(WAR_MANAGER_ROLE);
    require(warManagerCount <= 10, "Too many war managers");
```

The issue would leave an inconsistent state, in which the registry has the new family owner, but access control in the vault is still retained by the old family.

Additionally, note that even if the implementation of `setAdmin()` doesn't revert by itself, the caller can intentionally submit a low gas amount to make the inner call fail while the outer call to `transferFamilyOwnership()` succeeds.

Recommendations:

Instead of ignoring the revert, use EIP-165 to check if the vault implementation supports `setAdmin()`.

Sonz: Resolved with `@52ff93fca2...` and `@64fa8fe9d1...`.

Zenith: Verified.

[M-2] Family owner can update vault to arbitrary smart contract

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

FamilyRegistryUpgradeable::setFamilyVault

Description:

FamilyRegistry contract allows family owner to update vault address.

Normally, vault contract is deployed via `VaultFactory` and it should follow pre-defined `FamilyVaultUpgradeable` contract logic.

However, vault updating logic does not check whether the new vault is using family vault logic or it has been deployed by `VaultFactory`:

File: contracts/core/FamilyRegistryUpgradeable.sol

```
function setFamilyVault(uint256 familyId, address vault) external {
  require(_families[familyId].exists, "Family does not exist");
  require(vault ≠ address(0), "Invalid vault address");

  // CRITICAL: Prevent vault address reuse across different families
  require(
    _vaultToFamilyId[vault] == 0 || _vaultToFamilyId[vault] == familyId,
    "Vault already assigned to another family"
  );

  if (authorizedFactories[msg.sender]) {
    ...
  } else {
  // Only family owner can modify existing vault
  require(msg.sender == _families[familyId].owner, "Not family owner");

  // Prevent orphaning funds: old vault must be empty before replacement
  address oldVault = _families[familyId].vault;
    if (oldVault ≠ address(0)) {
      require(oldVault.balance == 0, "Old vault must be empty");
      // Remove old vault from reverse mapping
```

```
      delete _vaultToFamilyId[oldVault];
    }
    _families[familyId].vault = vault;
    _vaultToFamilyId[vault] = familyId;
    emit FamilyVaultSet(familyId, vault);
    }
```

Impact:

- An attacker can run a family with the vault that does not follow the expected game rule.
- This vault will not be upgraded along with VaultFactory's beacon upgrade

POC

File: test/integration/RegistryVaultFlow.test.js

```
it("family owner can set vault to arbitrary address", async function () {
  const newVaultAddress = "0x0000000000000000000000000000000000000123";
  await registry.connect(alice).createFamily("AliceClan");
  await registry.connect(alice).setFamilyVault(1, newVaultAddress);
  const isValid = await registry.isValidVault(newVaultAddress);
  expect(isValid).to.be.true;
});
```

Recommendations:

Verify if the new vault is deployed by `VaultFactory`.

Alterantively, check if vault's implementation address equals to `VaultFactory` beacon's implementation address

Sonz: Resolved with `@1285cffed1 ...`.

Zenith: Verified.

4.2 Low Risk

A total of 2 low risk findings were identified.

[L-1] Multiple pending implementations can lead to confusing upgrades

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- VaultFactory.sol

Description:

The VaultFactory supports multiple concurrent pending upgrades of the vault implementation.

This can lead to confusing mechanics about how and when upgrades would be applied. For example, schedule two at the same time, then apply one when ready and leave one dormant to eventually apply it.

Recommendations:

Change the logic to support at most one upgrade at the same time. This would also contribute to a simpler implementation.

Sonz: Resolved with `@1f3923feea ...`.

Zenith: Verified.

[L-2] Removing a token from whitelist can be DOS'ed by 1-wei donation

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

FamilyVaultUpgradeable::setTokenWhitelist

Description:

When removing a token from whitelist, there is a check that vault should not hold any token:

File: contracts/core/FamilyVaultUpgradeable.sol

```
function setTokenWhitelist(address token, bool status)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
    {
        require(token ≠ address(0), "Invalid token");

        // Prevent removing from whitelist if vault holds tokens
        // This prevents admin from bypassing recoverERC20 protection
        if (!status && whitelistedTokens[token]) {
            require(IERC20(token).balanceOf(address(this)) == 0,
                "Cannot remove token with balance");
        }

        whitelistedTokens[token] = status;
        emit TokenWhitelisted(token, status);
    }
```

If an attacker donates 1-wei token to the vault, the admin cannot remove the token from the whitelist until treasury manager withdraws it with `withdrawERC20` method.

After treasurer withdraws the donation, the attacker can donate 1 wei before admin calls `setTokenWhitelist`.

The attacker can replay this attack indefinitely with little token balance.

Recommendations:

Remove such constraint, because:

- The token is already going to be removed from whitelist. And non-whitelisted tokens can be recovered by admin.
- It doesn't make sense to restrict admin action. Because admin has top privilege. If they want to, they can bypass `recoverERC20` protection by granting them treasurer role

Sonz: Resolved with `@ed2aab9c2e ...`.

Zenith: Verified.

4.3 Informational

A total of 3 informational findings were identified.

[I-1] Implement a proxy registry in VaultFactory

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- VaultFactory.sol

Description:

Since vaults can be created through the VaultFactory or directly by the family owner, it would be beneficial and future-proof to know which vaults were deployed via the factory, so it can determined which ones follow a known implementation.

Recommendations:

Add a mapping to signal which proxies have been created through the factory.

```
function createVault(uint256 familyId) external returns (address proxy) {
    require(IFamilyRegistry(registry).getFamilyOwner(familyId) ==
    msg.sender, "Not family owner");

    // Prevent vault overwrite to avoid orphaning existing vaults with funds
    IFamilyRegistry.Family memory family
    = IFamilyRegistry(registry).getFamily(familyId);
    require(family.vault == address(0), "Vault already exists");

    bytes memory init = abi.encodeCall(FamilyVaultUpgradeable.initialize,
    (msg.sender, familyId, registry));
    proxy = address(new BeaconProxy(address(beacon), init));

       isVault[proxy] = true;
```

```
        IFamilyRegistry(registry).setFamilyVault(familyId, proxy);
        emit VaultCreated(familyId, proxy, msg.sender);
    }
```

Sonz: Resolved with @77bac3e70a ....

Zenith: Verified.

[I-2] Registry reference in VaultFactory can be immutable

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- VaultFactory.sol

Description:

The `registry` address doesn't change during the VaultFactory contract's lifecycle.

Recommendations:

Change `registry` to be immutable.

Sonz: Resolved with `@70c3347259 ...`.

Zenith: Verified.

[I-3] Incorrect storage gap calculation

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- FamilyRegistryUpgradeable
- FamilyVaultUpgradeable
- TeleMafiaCheckInUpgradeable

Description:

Upgradeable contracts are using storage gap to avoid storage conflicts for future upgrade.

But gap caulcations are inconsistent across smart contracts.

For example:

File: contracts/core/FamilyRegistryUpgradeable.sol

```
/**
* @dev Storage gap for future upgrades
* Calculation:
* - Target: 50 slots (OpenZeppelin standard)
* - Initializable: 1 slot
* - OwnableUpgradeable: 1 slot
* - UUPSUpgradeable: 0 slots (no storage)
* - ReentrancyGuardUpgradeable: 1 slot
* - Own storage: 2 slots (familyCount) + 4 mapping references
    (pendingUpgrades added)
* - Total used: ~7 slots
* - Gap: 50 - 7 = 43 slots
*/
uint256[43] private __gap;
```

Here, the calculation is incorrect, because:

- OpenZeppelin's `Initializable`, `OwnableUpgradeable`, `ReentrancyGuardUpgradeable` are all following ERC-7201, so they do not consume normal storage slots
- Moreover, since constant variables do not consume storage slots,

`FamilyRegistryUpgradeable` only has 5 storage slots

File: contracts/core/FamilyRegistryUpgradeable.sol

```
mapping(uint256 ⇒ Family) private _families;
uint256 public familyCount;
mapping(address ⇒ bool) public authorizedFactories;
/// @notice Reverse mapping: vault address ⇒ family ID (for O(1) validation)
mapping(address ⇒ uint256) private _vaultToFamilyId;
/// @notice Timelock for upgrades: 48 hours
uint256 public constant UPGRADE_DELAY = 48 hours; // @audit constants do not
    consume storage slot
/// @notice Mapping of pending upgrade implementations to their unlock
    timestamps
mapping(address ⇒ uint256) public pendingUpgrades;
```

Thus, the correct gap should be 50 - 0 - 5 = 45.

However, the idea of gaps is to leave some space for future additions in base contracts. So, if a contract is not meant to be extended, there is no point to include a stroage gap.

Recommendations:

Gaps are only useful in base contracts, there is no point in using them in the top contract as there is no derived that could break the layout.

Or, follow ERC-7201 Namespaced Storage Layout in order to avoid future storage collision and drop confusing storage gap concept.

Sonz: Resolved with `@b5182066e1...`.

Zenith: Verified.