hexens x Fira

Security Review Report for Fira

February 2026

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This report covers the security review for Fira Protocol V1. Fira is a new protocol that built on top of Pendle and Morpho to realize permissionless fixed-term and fixed-interest loans.

Our security assessment was a full review of the code, spanning a total of 3 weeks.

During our review, we did not identify any major security vulnerability.

We did identify some minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

3. Security Review Details

▪ Review Led by

Jahyun Koo, Lead Security Researcher

▪ Scope

The analyzed resources are located on:

🔗 .../fira (main repo)
🔗 .../fira-lending-market (one file, src/lending_market/FiraLendingMarket.sol)

📌 **Commit:** `df79ffc4da474c7fe161f1c8045e00b72011ce73`

The issues described in this report were fixed. Corresponding PRs are mentioned in the description.

▪ Changelog

29 January 2026	Audit start
23 February 2026	Initial report
23 February 2026	Revision received
26 February 2026	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

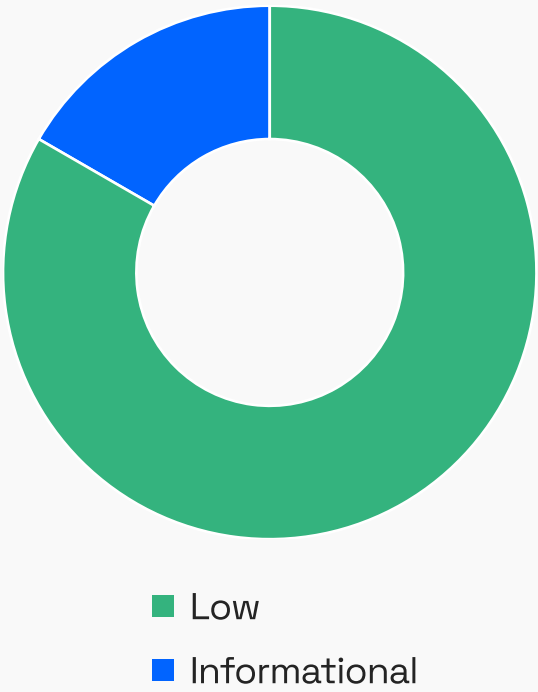
- **Issue Symbolic Codes**

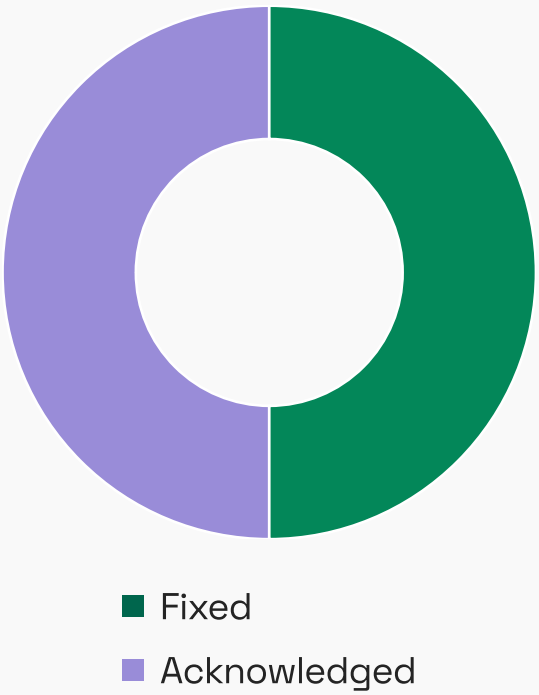
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	0
High	0
Medium	0
Low	5
Informational	1
Total:	**6**





6. Weaknesses

This section contains the list of discovered weaknesses.

USL5-6 | Post-expiry BT-only redemption lets lending-origin BT consume FW-backed liquidity

Acknowledged

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

src/fira_bonding/core/YieldContracts/CouponToken.sol

Description:

The protocol design assumes risk isolation between two domains: Lending Market (including LI-origin exposure) and FW-backed issuance through the USDCFW/CT path. LM-origin BT risk is expected to be resolved inside Lending Market accounting, while FW-backed BT is expected to be covered by FW reserves.

That boundary is not enforced in post-expiry redemption. BT is fully fungible, and BT-only redemption does not verify whether the BT being redeemed was FW-backed at issuance. As a result, LM-origin BT can be redeemed through the CT/FW redemption path after expiry and withdraw FW liquidity.

```
function _redeemBC(
    address[] memory receivers,
    uint256[] memory amountBcToRedeems
) internal returns (uint256[] memory amountFwOuts) {
    uint256 totalAmountBcToRedeem = amountBcToRedeems.sum();
    IBondToken(BT).burnByCT(address(this), totalAmountBcToRedeem);
    if (!isExpired()) _burn(address(this), totalAmountBcToRedeem);

    uint256 index = _bcIndexCurrent();
    uint256 totaFwInterestPostExpiry;
    amountFwOuts = new uint256[](receivers.length);

    for (uint256 i = 0; i < receivers.length; i++) {
        uint256 fwInterestPostExpiry;
        (amountFwOuts[i], fwInterestPostExpiry) = _calcFwRedeemableFromBC(
```

```
                amountBcToRedeems[i],
                index
            );
            _transferOut(FW, receivers[i], amountFwOuts[i]);
            totaFwInterestPostExpiry += fwInterestPostExpiry;

            emit Burn(
                msg.sender,
                receivers[i],
                amountBcToRedeems[i],
                amountFwOuts[i]
            );
        }
        if (totaFwInterestPostExpiry != 0) {
            postExpiry.totalFwInterestForTreasury += totaFwInterestPostExpiry
                .Uint128();
            emit TreasuryFwInterestAccrued(
                totaFwInterestPostExpiry,
                postExpiry.totalFwInterestForTreasury
            );
        }
    }
```

A liquidator may later source BT and repay debt to seize collateral, and liquidation bonus can support that behavior. However, this is market-dependent and not a protocol-level guarantee that FW coverage is restored. It can require external BT liquidity or new FW deposits to mint replacement backed BT.

If bad debt is realized and Lending Market debt is written down, Lending Market state is reduced, but FW already transferred out via post-expiry redemption is not automatically restored. BT circulation and FW reserve coverage can therefore diverge, creating a FW-side coverage shortfall. In that state, late BT/CT redeemers can face partial redemption or full redemption failure. Underwater conditions increase the incentive to take this path, but the root issue is the missing source-aware redemption boundary.

Remediation:

Introduce explicit redemption entitlement accounting that is separate from BT balance. Assign post-expiry BT-only redemption entitlement only to BT minted through the FW-backed path. BT minted through LI supply or LM borrowing should not carry that entitlement by default.

Propagate entitlement during transfers and consume entitlement during redemption checks, so balance fungibility does not imply redemption-right fungibility. Keep LM repayment accounting independent from FW redemption eligibility. Also block new borrowing after maturity, including

the grace boundary, so newly borrowed BT cannot enter an already open post-expiry redemption window.

Commentary from the client:

"The economic impact is bounded by existing mechanisms and the attack is never profitable.

1. *Impossible through standard flows: ActionBorrow.borrowSingleToken atomically converts BT → FW → USDC and sends USDC to the borrower. The borrower never holds BT. Executing this attack requires deliberately bypassing the router and calling LendingMarket.borrow() directly.*
2. *Never profitable due to over-collateralization: At 85% LLTV, borrowing 1M BT requires ~1.176M collateral. Post-expiry redemption yields ~909K FW (at BC index 1.1). The attacker always loses: collateral locked > FW extracted, regardless of price movement.*
3. *Self-healing via liquidation: After expiry + grace period, all positions are unconditionally liquidatable. Liquidators must mint BT (via USDC → FW → CT.mintBC()), which restores the exact FW extracted by the attacker. Net CT FW change = zero for solvent positions.*
4. *Residual risk = bad debt risk: The only scenario where CT holders face a shortfall is if collateral crashes so fast that liquidation cannot catch it — this is standard bad debt risk inherent to any lending protocol, already socialized via totalSupplyAssets -= badDebtAssets. The CT shortfall never exceeds the bad debt amount; it's the same risk in a different location, not additive.*
5. *Post-grace borrows are dangerous: After expiry + maturityGracePeriod, any new borrow can be liquidated in the same block by a monitoring bot. The attacker's window is a single transaction."*

USL5-5 | CouponToken index initialization attack leads to no interest for CT holders

Acknowledged

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

fira/src/fira_bonding/core/YieldContracts/CouponToken.sol:_bcIndexCurrent#L416-L426

Description:

The CouponToken can mint BT and CT according to the provided FW. It effectively tokenizes the yield generated from the FW. The yield from the FW comes from the **exchangeRate()**.

As the exchange rate grows, the amount of underlying FW between the BT and CT shifts from the BT to the CT, such that when expired 1 BT would equal enough FW equal to 1 underlying asset and 1 CT would equal the rest (the generated yield).

This index is cached in **_bcIndexCurrent** as the maximum between the stored value and the actual live value:

```
uint128 index128 = PMath.max(IFiraWrappedStandardized(FW).exchangeRate(), _bcIndexStored).Uint128();
```

This makes it so that the CT index can only increase, as the interest distribution logic would otherwise break. However, because the FW exchange rate can be easily manipulated, it becomes an attack vector during initialization.

The FW exchange rate is the effective share rate between the total underlying and the total supply:

```
function exchangeRate() public view override returns (uint256) {
    uint256 supply = totalSupply();
    if (supply == 0) {
        return 0; // Undefined before first mint
    }
    return _totalUnderlying().divDown(supply);
}
```

Where the total underlying is simply the idle assets plus the value of the SisuVault shares:

```
function _totalUnderlying() internal view returns (uint256) {
    if (vaultShares == 0) {
        return idleAssets;
    }
    return idleAssets + vault.convertToAssets(vaultShares);
}
```

The vault shares are cached and cannot directly be donated to, but it is possible to directly donate to the LendingMarket for the SisuVault and inflate the value of the shares indirectly.

Using an index reset with inflation attack, it becomes possible to have the CouponToken cache a very high index, while afterwards resetting the index back to normal.

Consider this example:

1. A new USDC FW, BT and CT are deployed.
2. Attacker mints 1 wei of FW using 1 wei USDC.
3. Attacker donates 1 USDC to the LendingMarket of the SisuVault of the FW.
4. The share rate increases to **1,000,000x** (since 1 USDC is **1e6**).
5. Attacker calls **CouponToken.bcIndexCurrent()** which stores this high index permanently.
6. Attacker burns their 1 wei FW for their 1 USDC, resetting the share rate to 1x.
7. Attacker can deposit 1 USDC into FW as normal, making everything seem as normal.
8. Any FW deposits into CT will create BT and CT as normal, but any interest generated will not be seen (as it only takes the stored high index) and thus CT holders will not get this interest.

After the attack, the share rate seems to be like normal. Only over time will the CT holders notice that they are not generating interest and the interest is actually going to the BT holders.

```
function _bcIndexCurrent() internal returns (uint256 currentIndex) {
    if (doCacheIndexSameBlock && bcIndexLastUpdatedBlock == block.number) return _bcIndexStored;

    uint128 index128 = PMath.max(IFiraWrappedStandardized(FW).exchangeRate(),
_bcIndexStored).Uint128();

    currentIndex = index128;
    _bcIndexStored = index128;
    bcIndexLastUpdatedBlock = uint128(block.number);

    emit NewInterestIndex(currentIndex);
}
```

Remediation:

We are still considering remediation. One potential option:

- Don't cache the rate when the CT contract is not holding any FW.

Commentary from the client:

"Our deployment procedure seeds FW with initial USDC, ensuring non-zero protocol-owned supply from genesis. With any seed, the attacker cannot burn the protocol's shares, cannot drop supply to zero, and cannot recover the donation. With even a 1 USDC seed, the donation only doubles the exchange rate (diluted by protocol shares) and becomes a permanent loss for the attacker — pure griefing with no profit."

Proof of concept:

```
import "./USDCFWBaseTest.t.sol";

contract PoC is USDCFWBaseTest {
    function test_poc() public {
        SisuVault sisuVault = variableRateVault;
        uint256 amount = 1000e6;
        (address BT, address CT, address lpMarket) = initFiraMarkets_WBTC_WETH();
        address FW = IBondToken(BT).FW();
        IFiraWrappedStandardized fw = IFiraWrappedStandardized(FW);
        IBondToken bt = IBondToken(BT);
        IBCToken ct = IBCToken(CT);
        IERC20 usdc = IERC20(USDC_MAINNET);
        ILendingMarket lending_market = sisuVault.LENDING_MARKET();
        FWuUSDC.setRehypothecationModule(address(new MockRehypothecationModule(0, 0, 0)));

        initVariableRateMarketAndPushToVault(1000000e6);
        deal(USDC_MAINNET, alice, amount);

        vm.startPrank(alice);
        usdc.approve(FW, type(uint).max);
        usdc.approve(address(sisuVault), type(uint).max);
        usdc.approve(address(lending_market), type(uint).max);

        MarketParams memory mParams0 = lending_market.idToMarketParams(sisuVault.supplyQueue(0));
        MarketParams memory mParams1 = lending_market.idToMarketParams(sisuVault.supplyQueue(1));

        uint256 initialFwBalance = fw.balanceOf(alice);
        uint256 initialVaultShareBalance = IERC20(address(sisuVault)).balanceOf(FW);
        console.log("initialFwBalance: %d", initialFwBalance);
        console.log("initialVaultShareBalance: %d", initialVaultShareBalance);
        console.log("fw.exchangeRate() = %d", fw.exchangeRate());

        fw.deposit(alice, USDC_MAINNET, 1, 0);

        uint256 finalFwBalance = fw.balanceOf(alice);
        uint256 finalVaultShareBalance = sisuVault.balanceOf(FW);
        console.log("finalFwBalance: %d", finalFwBalance);
        console.log("finalVaultShareBalance: %d", finalVaultShareBalance);
        console.log("fw.exchangeRate() = %d", fw.exchangeRate());

        // donation
```

```
        lending_market.supply(mParams0, 1e6, 0, address(sisuVault), "");

        console.log("fw.exchangeRate() = %d", fw.exchangeRate());
        console.log("ct.bcIndexCurrent() = %d", ct.bcIndexCurrent());

        // reset FW rate
        fw.redeem(alice, 1, USDC_MAINNET, 0, false);

        console.log("fw.exchangeRate() = %d", fw.exchangeRate());
        console.log("ct.bcIndexCurrent() = %d", ct.bcIndexCurrent());

        // deposit like normal
        fw.deposit(alice, USDC_MAINNET, 100e6, 0);

        console.log("fw.exchangeRate() = %d", fw.exchangeRate());
        console.log("ct.bcIndexCurrent() = %d", ct.bcIndexCurrent());

        // mint BT + CT
        fw.transfer(CT, 50e6);
        ct.mintBC(alice, alice);

        console.log("bt.balanceOf(alice) = %d", bt.balanceOf(alice));
        console.log("ct.balanceOf(alice) = %d", ct.balanceOf(alice));

        // Simulate interest
        lending_market.supply(mParams0, 100e6, 0, address(sisuVault), "");

        (uint interest, ) = ct.redeemDueInterestAndRewards(alice, true, false);
        // CT holders get 0
        console.log("interest = %d", interest);

        bt.transfer(CT, bt.balanceOf(alice));
        ct.transfer(CT, ct.balanceOf(alice));
        uint fwOut = ct.redeemBC(alice);

        // Same FW out, which includes the interest, so the BT holders actually get the interest
        console.log("fwOut = %d", fwOut);
    }
}
```

USL5-1 | Unrestricted update of blockCycleNumerator in BCLpOracle

Fixed in PR112

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

src/fira_bonding/oracles/BCLpOracle.sol#L135-L142

Description:

In **BCLpOracle**, the function **setBlockCycleNumerator()** is labeled as owner-only but has no access control. As a result, any account can change **blockCycleNumerator**, which influences the cardinality requirement returned by **getOracleState()**. This does not directly affect the BT/CT/LP rate outputs, but it can lead to inconsistent or misleading oracle state information.

```
function _setBlockCycleNumerator(uint16 newBlockCycleNumerator) internal {
    if (newBlockCycleNumerator < BLOCK_CYCLE_DENOMINATOR) {
        revert InvalidBlockRate(newBlockCycleNumerator);
    }

    blockCycleNumerator = newBlockCycleNumerator;
    emit SetBlockCycleNumerator(newBlockCycleNumerator);
}
```

Remediation:

Add access control to restrict **setBlockCycleNumerator()** to an authorized account, or remove the external setter and keep the value fixed after deployment.

USL5-2 | Unit Mismatch in borrowSingleToken Slippage Check Leads to Unnecessary Reverts

Fixed in PR113

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

src/fira_bonding/router/ActionBorrow.sol#L76-L107

Description:

In **borrowSingleToken**, the **minTokenOut** value from **TokenOutput** is passed directly as **minFwOut** to **_swapExactBtForFw**. **minTokenOut** is denominated in the final output token (e.g., USDC), while **minFwOut** is denominated in FW shares. Because these units can differ due to the exchange rate, the check can be overly strict and cause a swap to revert even when the eventual redemption would meet the user's minimum output. The function later enforces **minTokenOut** during **_redeemFwToToken**, so this issue does not weaken the final slippage protection, but it can make the operation fail unnecessarily.

```
function borrowSingleToken(
    MarketParams memory marketParams,
    ILendingMarket lendingMarket,
    IPMarketV3 firaMarket,
    uint256 tokensToBorrow,
    TokenOutput calldata output,
    LimitOrderData calldata limitOrderData,
    address receiver
) external returns (uint256 btBorrowed, uint256 tokenBorrowed) {
    require(tokensToBorrow > 0, "AB: zero borrow");
    require(receiver != address(0), "AB: zero receiver");
    require(address(firaMarket) != address(0), "AB: zero firaMarket");
    require(address(lendingMarket) != address(0), "AB: zero lendingMarket");
    // Borrow BT from LM
    (btBorrowed,) = ILendingMarket(lendingMarket)
        .borrow(
            marketParams,
            tokensToBorrow,
            0,
            msg.sender, // Debt ownership
            address(this) // Token receiver
        );
```

```
        (IFiraWrappedStandardized FW, IBondToken BT,) = firaMarket.readTokens();
        _transferOut(address(BT), _entry_swapExactBtForFw(address(firaMarket), limitOrderData), btBorrowed);
        (uint256 fwOut,) =
            _swapExactBtForFw(address(this), address(firaMarket), btBorrowed, output.minTokenOut,
limitOrderData);
        // Transfer FW to FW to be burned as shares
        _transferOut(address(FW), address(FW), fwOut);
        tokenBorrowed = _redeemFwToToken(address(this), address(FW), fwOut, output, false);
        // 5. Transfer the borrowed token to receiver
        _transferOut(output.tokenOut, receiver, tokenBorrowed);
    }
```

Remediation:

Ensure that the minimum output used for the BT→FW swap is expressed in FW units. This can be done by either omitting the FW-level minimum and relying on the final **minTokenOut** check, or by introducing a separate FW-denominated minimum parameter (or derived value) for the intermediate swap.

USL5-3 | Use of convertToAssets/convertToShares instead of previewMint/previewWithdraw can cause minor rounding mismatches

Fixed in PR114

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

src/fira_bonding/StandardizedYield/implementations/USDCFW.sol

Description:

The USDCFW contract uses **convertToAssets** and **convertToShares** from the ERC-4626 vault in places where **previewMint** or **previewWithdraw** would be more accurate. In ERC-4626, **previewMint** and **previewWithdraw** apply ceiling rounding, while **convertTo*** uses floor rounding. With SisuVault's fee and decimals offset logic, this can lead to small rounding differences, resulting in slightly conservative calculations causing minor under-estimation in internal accounting or user outcomes.

- fira_bonding/StandardizedYield/implementations/USDCFW.sol#L158

When calculating the required assets for **sharesToMint** which represents the target share amount, **convertToAssets** is used → **previewMint** with ceiling rounding better matches the intended behavior

```
function _redeemVaultShares(uint256 underlyingOut) private returns (uint256 sharesOut) {
    sharesOut = vault.convertToShares(underlyingOut);

    if (sharesOut <= vaultShares) {
        // Use existing vault shares
        vaultShares -= sharesOut;
    } else {
        // Need to mint additional shares from idle assets
        uint256 sharesToMint = sharesOut - vaultShares;
        uint256 assetsNeeded = vault.convertToAssets(sharesToMint);
```

- fira_bonding/StandardizedYield/implementations/USDCFW.sol#L205

When calculating the shares required to subtract **fromVault** which represents the target asset amount, **convertToShares** is used → **previewWithdraw** with ceiling rounding better matches the intended behavior

```
        if (fromVault > 0) {
            uint256 sharesToBurn = vault.convertToShares(fromVault);
            if (sharesToBurn > vaultShares) {
                sharesToBurn = vaultShares;
                fromVault = vault.convertToAssets(sharesToBurn);
                assetsOut = fromIdle + fromVault;
            }
```

Remediation:

Consider replacing **convertToAssets** with **previewMint** when calculating required assets for a target share amount, and replacing **convertToShares** with **previewWithdraw** when estimating shares needed for a target asset amount.

USL5-4 | Market Constants Can Be Set by First Creator Due to ID Derivation

Acknowledged

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

lending_market/LendingMarket.sol#L216-L242

Description:

createMarket is permissionless, and the market ID is derived only from **MarketParams**. As a result, **MarketConstants** are not part of the uniqueness check. A party can preemptively create a market with the same parameters and supply arbitrary constants. Subsequent attempts to create the intended market revert as already created. This can lock in unfavorable constants for that parameter set.

```
    function createMarket(MarketParams memory marketParams, MarketConstants memory constants) external
virtual {
        _createMarket(marketParams, constants);
    }

    function _createMarket(MarketParams memory marketParams, MarketConstants memory constants)
internal {
        Id id = marketParams.id();
        require(isIrmEnabled[marketParams.irm], ErrorsLib.IRM_NOT_ENABLED);
        require(isLltvEnabled[marketParams.lltv], ErrorsLib.LLTV_NOT_ENABLED);
        require(isLtvEnabled[marketParams.ltv] || marketParams.ltv == 0, ErrorsLib.LTV_NOT_ENABLED);
        require(marketParams.ltv < marketParams.lltv || marketParams.ltv == 0,
ErrorsLib.LTV_EXCEEDS_LLTV);
        require(market[id].lastUpdate == 0, ErrorsLib.MARKET_ALREADY_CREATED);
        require(
            constants.liquidationIncentive <= MAX_LIQUIDATION_INCENTIVE_FACTOR - WAD,
            ErrorsLib.LIQUIDATION_INCENTIVE_NOT_SET
        );

        // Safe "unchecked" cast.
        market[id].lastUpdate = uint128(block.timestamp);
        idToMarketParams[id] = marketParams;
        marketConstants[id] = constants;
        emit EventsLib.CreateMarket(id, marketParams);
```

```
        // Call to initialize the IRM in case it is stateful.
        if (marketParams.irm != address(0)) {
            IIrm(marketParams.irm).borrowRate(marketParams, market[id]);
        }
    }
```

Remediation:

Consider ensuring that **MarketConstants** are part of the market's uniqueness, or restrict **createMarket** so only approved callers can set constants for a given parameter set.

Commentary from the client:

"In our deployment, we create markets through our own scripts in a controlled sequence — so there's no race condition. If someone front-runs, the deploy script would simply revert ("market already exists") and we'd detect it. And even if we get front-run, we have plenty of room to deploy other markets just by changing LTV by 1 wei. There is no point of griefing this"

hexens x Fira