



ZEALYNX

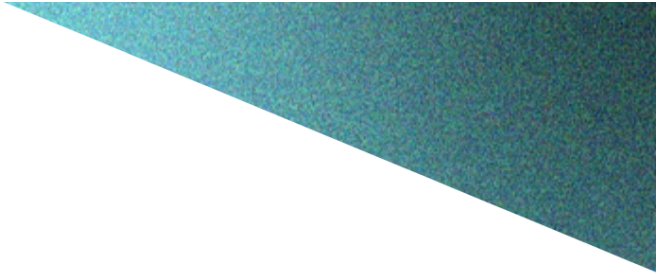
Web3 Security and AI Agent & MCP Audits

Coffer
Smart Contract Audit

22 October 2026

Zealynx

contact@zealynx.io



Contents

1. About Zealynx

2. Disclaimer

3. Overview

3.1 Project Summary

3.2 About Coffe

3.3 Audit Scope

4. Audit Methodology

5. Severity Classification

6. Executive Summary

6.1 The Key Findings

6.2 Architectural Security Observations

6.3 Security Strengths Observed

7. Audit Findings

7.1 Medium Severity Findings

7.2 Low Severity Findings

7.3 Informational Severity Findings

1. About Zealynx

Zealynx, founded in January 2024 by Carlos (Bloqarl), specializes in smart contract audits, and development. Our services include comprehensive smart contract audits, application security audits, such as pentesting, and AI Audits. We are trusted by clients such as Badger DAO, Ample protocol, Lido, Inverter, Matchain, and Golden Grid.

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our website [Zealynx.io](https://zealynx.io) and our blogs.

Zealynx has achieved public recognition, including winning 1st prize in the Uniswap Hook Hackathon and a Top 5 position in the Beanstalk Audit public contest. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

2. Disclaimer

A security review of a smart contract can't guarantee there are no vulnerabilities. It's a limited effort in terms of time, resources, and expertise to find as many vulnerabilities as possible. We can not assure 100% security post-review or guarantee the discovery of any issues with your smart contracts.

3. Overview

3.1 Project Summary

Coffer lets an Ethereum validator issue fixed-term, fixed-rate ETH bonds as NFTs against its own stake. The validator's Coffer is bound to its withdrawal credentials. If a matured bond is not paid, holders can enter default and anyone can request the validator's EIP-7002 exit, bringing the stake back to repay the bonds.

3.2 About Coffer

A validator deploys a non-upgradeable Coffer through a factory and points its withdrawal credentials at it. Buyers purchase bond NFTs whose maturity value is a claim on that Coffer. Principal, less protocol fees, goes to the validator. The validator must keep enough execution-layer ETH in the Coffer to pay every matured bond, using direct transfers or consensus withdrawals. If that duty is missed, default freezes sales and validator extraction, accelerates all bonds, and enables a permissionless validator exit. A separate EIP-712 marketplace supports ETH listings and WETH offers. Shared NFT, redemption escrow and fee-curve contracts serve every Coffer.

3.3 Audit Scope

Ten Solidity and deployment-script files totalling 1,178 nSLOC, frozen at core commit 793f6a7 and marketplace commit 9262181.

```
coffer-smart-contracts/src/Coffer.sol
coffer-smart-contracts/src/CofferFactory.sol
coffer-smart-contracts/src/CofferBondNft.sol
coffer-smart-contracts/src/FeeCurve.sol
coffer-smart-contracts/src/CofferRedemptionEscrow.sol
coffer-smart-contracts/src/libraries/Interest.sol
coffer-smart-contracts/script/CreateCoffer.s.sol
coffer-smart-contracts/script/DeployCofferFactory.s.sol
coffer-marketplace/src/CofferMarketplace.sol
coffer-marketplace/script/DeployCofferMarketplace.s.sol
```

4. Audit Methodology

Approach

During our security assessments, we uphold a rigorous approach to maintain high-quality standards. Our methodology encompasses thorough **Fuzz** and **Invariant Testing**, and **meticulous manual security review**.

Throughout the smart contract audit process, we prioritize the following aspects to uphold excellence:

- 1. Code Quality:** We diligently evaluate the overall quality of the code, aiming to identify any potential vulnerabilities or weaknesses.
 - 2. Best Practices:** Our assessments emphasize adherence to established best practices, ensuring that the smart contract follows industry-accepted guidelines and standards.
 - 3. Documentation and Comments:** We meticulously review code documentation and comments to ensure they accurately reflect the underlying logic and expected behavior of the contract.

5. Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

6. Executive Summary

This was a one-and-a-half-week review by one auditor of frozen core and marketplace commits. Work combined manual review, stateful fuzzing, invariants, symbolic execution, differential and mutation testing, mainnet-fork checks, and execution of the EIP-7002 path against the consensus specification.

The audit identified **11 findings: 2 Medium, 5 Low and 4 Informational**. Nine were reproduced mechanically and two were established by code trace. No Critical or High issue was found. The holder's atomic redeem-or-default and permissionless forced-exit paths held across the tested scenarios.

Mitigation review verified **7 fixes and 4 acknowledged residual risks**. The principal code fix makes marketplace cancellation epochs unpredictable. Application and documentation changes corrected redemption funding and consensus timing. The retained runtime behaviors are identified in their findings.

A fixed-duration review cannot be exhaustive, and the absence of a finding in an area is not assurance about that area.

6.1 The Key Findings

Cancellation could activate the order it was meant to revoke. The fix moves all six cancellation nonces by an unpredictable step.

Partial default claims can leave external NFT listings stale. External venues must validate the bond's remaining value during transfer.

Official guidance could cause an honest validator to default. Redemption funding and consensus withdrawal timing were corrected in the app and documentation.

Four runtime conditions were accepted. They concern the 2106 timestamp boundary, third-party default ordering, external NFT sales and primary-purchase slippage.

6.2 Architectural Security Observations

The central safety property spans two layers and time. A Coffey sells execution-layer claims backed by consensus-layer stake, but the contracts cannot observe validator balance, penalties, credential changes or queue congestion. `issueSize`, its buffer and buyer-side credential checks bridge that gap. The validator's real duty is aggregate and temporal: keep enough execution balance for every matured, unpaid bond. L-02, L-03 and L-04 arise where a single-bond check, selected batch or timing estimate is treated as that whole duty.

Immutability moves security into deployment and disclosure. Each Coffey is a non-upgradeable deterministic clone tied to shared NFT, escrow and fee contracts. This removes proxy-admin and implementation-substitution risk, but a wrong credential binding, shared-contract owner or permanent type limit cannot be repaired in place. Deployment scripts, post-deployment ownership checks and integration guidance therefore carry the same weight as runtime code.

Signed intent must remain valid across later state changes. Marketplace orders bind live maturity value and nonces, protecting against several stale-state fills. M-01 shows that cancellation becomes an activation boundary when a nonce advances predictably. L-05 shows the same time dimension on primary purchases: a transaction without a deadline can land under later parameters.

A bond ID identifies a claim whose value can change. During default, a partial claim reduces maturity value without changing the transferable ERC-721 ID. The Coffey marketplace rechecks value atomically; external venues and collateral integrations must do the same.

6.3 Security Strengths Observed

Holder recovery does not depend on validator cooperation. `holderRedeemBondOrElseDefault` atomically pays or enters default. In default, sales and validator withdrawals stop, settlement remains available, and `exitValidator` is permissionless and repeatable. The validator key is fixed in the clone and EIP-7002 authenticates the withdrawal credential, so callers cannot redirect the stake. The review exercised this path against system-contract behavior and the consensus specification, including pending partial withdrawals.

Capacity accounting is deliberately conservative. Issuance consumes `issueSize`; restricted parameter changes cannot weaken outstanding bonds; execution withdrawals remain capped while bonds exist; and the buffer leaves room for ordinary consensus losses. Stateful testing across funding, penalties and exit sweeps preserved the modeled solvency invariant.

Settlement isolates hostile recipients. Validator redemptions pay a pull-based escrow instead of calling each NFT holder. A contract that rejects ETH cannot block settlement or keep `outstandingBonds` live. The NFT restricts minting to factory-registered Coffers and burning or metadata updates to the issuing Coffer.

The marketplace has strong transaction-level controls. EIP-712 orders bind economic terms, fills compare signed maturity value with live state, and nonces are consumed before interaction. ETH seller payments use a gas cap with WETH fallback, while offer settlement avoids an ERC-721 receiver callback.

The project exposes assumptions clearly. READMEs and NatSpec state the cross-layer trust model, accepted first-come-first-served behavior and operational duties in testable terms. Remediation replies distinguish code fixes, documentation mitigations and accepted risks, making final status review auditable.

Summary of Findings:

Vulnerability	Severity	Status
[M-01] Cancel-all activates orders signed for the next global nonce instead of invalidating them	Medium	Fixed
[M-02] A partial default claim leaves an external sale valid for the bond's unreduced token id	Medium	Acknowledged
[L-01] Bonds bought after February 2106 are mature at purchase because startTimestamp is truncated to uint32	Low	Acknowledged
[L-02] The validator can void a third party's default declaration without paying	Low	Acknowledged
[L-03] Redeem Bonds guidance omits funding reserved for other matured bonds	Low	Fixed
[L-04] Docs time consensus funding at "roughly two days", but partial withdrawals wait in the exit queue, so a validator following them defaults	Low	Fixed
[L-05] Stale bond purchases can settle below the buyer's quoted maturity value	Low	Acknowledged
[I-01] Default documentation incorrectly states that every validator extraction path is frozen	Info	Fixed
[I-02] Marketplace checklist omits validator-to-Coffer credential binding	Info	Fixed
[I-03] Deployment leaves the shared FeeCurve owned by the deployer key	Info	Fixed
[I-04] Consensus withdrawal event reports the requested amount as withdrawn	Info	Fixed

7. Audit Findings

7.1 Medium Severity Findings

[M-01] Cancel-all activates orders signed for the next global nonce instead of invalidating them

Affected files

coffer-marketplace/src/CofferMarketplace.sol#L332-L338
coffer-marketplace/src/CofferMarketplace.sol#L415-L421
coffer-marketplace/README.md#L86

Description

cancelAllListings and cancelAllOffers are documented as safe sweeps that invalidate every order for the caller. Fills require the signed global nonce to equal the stored value, but each sweep changes that value to exactly $\text{current} + 1$. A signature over the next global nonce is therefore invalid before the sweep and valid immediately afterward.

This requires the maker to sign a future global nonce. The documented signing steps require the current per-bond nonce but do not say to read and sign the current global nonce, so a malicious or faulty signing interface can supply $G + 1$. The maker still authorized the signed trade terms; the defect is that the documented recovery action activates that authorization instead of revoking it.

Vulnerable Scenario: The following steps help understand the issue: 1. A maker signs an order whose globalNonce is the stored value plus one (e.g. supplied by a faulty or malicious signing interface); it cannot be filled yet. 2. The maker calls cancelAllListings / cancelAllOffers to invalidate everything it signed. 3. The sweep sets the stored global nonce to exactly the signed value. 4. Anyone holding the signature fills it on the signed terms.

Impact

If a maker signs an order at $G + 1$ and then relies on cancel-all, the signature holder can irreversibly sell the maker's bond or spend its WETH on the signed adverse terms.

PoC

Add to test/unit/ and run `forge test --match-path test/unit/CancelAllArmsNextNonce.t.sol -vv`. It uses the repository's own CofferMarketplaceTest fixture.

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.34;

import {CofferMarketplaceTest} from "../CofferMarketplace.t.sol";

contract CancelAllArmsNextNonce is CofferMarketplaceTest {
    function test_cancelAllListingsArmsASignatureForTheNextGlobalNonce() public {
        // A listing at a fraction of the bond's value, signed for the NEXT global
        nonce. It cannot be
        // filled today, so it looks harmless to the seller and to any transaction
        simulation.
        uint128 price = 0.1 ether;
        uint64 exp = uint64(block.timestamp + 30 days);
        uint256 nextGlobal = marketplace.sGlobalListingNonce(seller) + 1;
        bytes memory sig = _signListing(SELLER_PK, bondId, price, 1 ether, exp, 0,
        nextGlobal);
        uint256 fee = (1 ether - uint256(price)) * 900 / 10_000;
```

```

        vm.prank(buyer);
        vm.expectRevert();
        marketplace.buySignedListing{value: price + fee}(bondId, seller, price, 1
ether, exp, 0, nextGlobal, sig);

        // The seller runs the documented safe sweep to invalidate everything it may
have signed.
        vm.prank(seller);
        marketplace.cancelAllListings();

        // The sweep moved the global nonce onto the signed value and armed the
listing.
        uint256 sellerBefore = seller.balance;
        vm.prank(buyer);
        marketplace.buySignedListing{value: price + fee}(bondId, seller, price, 1
ether, exp, 0, nextGlobal, sig);

        assertEq(bondNft.ownerOf(bondId), buyer, "bond sold after the seller's cancel-
all");
        assertEq(seller.balance - sellerBefore, price, "seller receives 0.1 ETH for a 1
ETH bond");
    }

    function test_cancelAllOffersArmsASignatureForTheNextGlobalNonce() public {
        // An offer far above the bond's value, signed for the NEXT global offer nonce.
        uint128 amount = 3 ether;
        uint64 exp = uint64(block.timestamp + 30 days);
        uint256 nextGlobal = marketplace.sGlobalOfferNonce(buyer) + 1;
        bytes memory sig = _signOffer(BUYER_PK, bondId, amount, 1 ether, exp, 0,
nextGlobal);

        vm.prank(seller);
        vm.expectRevert();
        marketplace.acceptSignedOffer(bondId, buyer, amount, 1 ether, exp, 0,
nextGlobal, sig);

        vm.prank(buyer);
        marketplace.cancelAllOffers();

        uint256 buyerWethBefore = weth.balanceOf(buyer);
        vm.prank(seller);
        marketplace.acceptSignedOffer(bondId, buyer, amount, 1 ether, exp, 0,
nextGlobal, sig);

        assertEq(bondNft.ownerOf(bondId), buyer, "offer filled after the buyer's
cancel-all");
        assertEq(buyerWethBefore - weth.balanceOf(buyer), amount, "buyer pays 3 WETH
for a 1 ETH bond");
    }

```

```
}  
}
```

Output at 9262181:

[PASS] test_cancelAllListingsArmsASignatureForTheNextGlobalNonce() (gas: 154900)

[PASS] test_cancelAllOffersArmsASignatureForTheNextGlobalNonce() (gas: 179163)

With the recommended change applied, both tests revert at the fill (`ListingRevoked()` / `OfferRevoked()`).

Recommendations

Treat the global nonce as an opaque epoch and move it to a value that was unpredictable when an existing signature was produced. For example:

```
function cancelAllListings() external {  
-     uint256 newGlobalNonce = ++sGlobalListingNonce[msg.sender];  
+     uint256 newGlobalNonce =  
+         uint256(keccak256(abi.encode(msg.sender, block.prevrando,  
sGlobalListingNonce[msg.sender])));  
+     sGlobalListingNonce[msg.sender] = newGlobalNonce;  
    emit AllListingsCancelled(msg.sender, newGlobalNonce);  
}
```

Apply the same change to `cancelAllOffers` and to the per-bond cancels (`cancelListing`, `cancelListings`, `cancelOffer`, `cancelOffers`), which also advance their nonce by exactly one. Document that makers must sign both the current per-bond nonce and the current global nonce. The tradeoff is that integrations and tests must read the emitted or stored nonce instead of assuming it increases by one.

Mitigation

Fixed. Commit 65a9286 moves every global and per-bond cancellation nonce forward by one plus an unpredictable 128-bit step derived from the caller, `block.prevrando` and the current nonce. Existing signatures cannot anticipate the post-cancellation value. The client added regression and fuzz tests covering all six cancellation functions. The final marketplace head passed build, test, lint and Slither checks.

[M-02] A partial default claim leaves an external sale valid for the bond's unreduced token id

Affected files

coffer-smart-contracts/src/Coffer.sol#L681-L692

Description

While a default stands, `holderRedeemBondInDefault` pays the smaller of the Coffer balance and a bond's remaining maturity value. When the balance is smaller, the function reduces `bondMaturityValue` but leaves the same transferable ERC-721 token id alive. An existing external-venue order that binds only the token id and price therefore remains executable after the claim represented by that id has decreased. The Coffer marketplace avoids this because its signed order also binds the maturity value, but standard NFT transfers do not.

Vulnerable Scenario:

A seller lists a defaulted bond with approximately 10.50 ETH remaining for 10.29 ETH while the Coffer holds 6 ETH.

After the buyer quotes the live state, the seller claims the 6 ETH through that bond, either before the fill or from a contract-order callback during the fill.

The venue transfers the unchanged token id and pays the seller even though the bond now carries only approximately 4.50 ETH.

The buyer eventually collects the reduced claim and loses approximately 5.79 ETH. If the seller first tops the Coffer up to one wei below the remaining value, the claim returns the top-up and leaves a 1-wei bond.

Impact

An untrusted seller can irreversibly keep both the sale price and a pre-transfer partial claim, causing an external-venue buyer to lose up to nearly the entire purchase price.

PoC

Add the following test to `test/unit/PartialClaimResale.t.sol`. Run `forge build` once first (the `BaseTest` fixture loads the `DepositContract` mock from its build artifact), then `forge test --match-path test/unit/PartialClaimResale.t.sol -vv`:

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.34;

import {BaseTest} from "../BaseTest.sol";
import {Coffer} from "../../src/Coffer.sol";
import {IERC721} from "@openzeppelin/contracts/token/ERC721/IERC721.sol";

/// Minimal fixed-price NFT venue: like any standard ERC-721 order, the order binds the
token id and the price only.
contract FixedPriceVenue {
    mapping(address nft => mapping(uint256 id => address seller)) public sellerOf;
    mapping(address nft => mapping(uint256 id => uint256 price)) public priceOf;

    function list(address nft, uint256 id, uint256 price) external {
        sellerOf[nft][id] = msg.sender;
        priceOf[nft][id] = price;
    }

    function buy(address nft, uint256 id) external payable {
        address seller = sellerOf[nft][id];
```

```

        require(msg.value == priceOf[nft][id], "price");
        delete sellerOf[nft][id];
        IERC721(nft).transferFrom(seller, msg.sender, id);
        (bool ok,) = seller.call{value: msg.value}("");
        require(ok);
    }
}

contract PartialClaimResale is BaseTest {
    FixedPriceVenue internal venue;
    address internal seller = makeAddr("seller");
    address internal buyer = makeAddr("buyer");
    uint256 internal sellerBond;
    uint256 internal otherBond;
    uint256 internal constant TRANCHE = 6 ether;
    uint256 internal constant SWEEP = 32 ether;

    function setUp() public override {
        super.setUp();
        venue = new FixedPriceVenue();
        vm.deal(seller, 100 ether);
        vm.deal(buyer, 100 ether);
        coffer = Coffer(payable(createDefaultCoffer()));
        vm.prank(seller);
        sellerBond = coffer.buyBond{value: 10 ether}(ONE_YEAR, 1);
        vm.prank(holder2);
        otherBond = coffer.buyBond{value: 5 ether}(ONE_MONTH, 1);
        advanceTime(ONE_MONTH + 1);
        coffer.declareDefault(otherBond);
        vm.deal(address(coffer), TRANCHE);
    }

    function _remaining(uint256 id) internal view returns (uint256 v) {
        (v,,) = coffer.sHolderConditions(id);
    }

    function _listAndQuote() internal returns (uint256 price, uint256 remaining) {
        remaining = _remaining(sellerBond);
        assertGt(TRANCHE + SWEEP, remaining + _remaining(otherBond), "Coffer is covered");
        price = remaining * 98 / 100;
        vm.startPrank(seller);
        bondNft.setApprovalForAll(address(venue), true);
        venue.list(address(bondNft), sellerBond, price);
        vm.stopPrank();
    }

    function _buyerCollectsEverything() internal {
        if (address(coffer).balance > 0) {

```

```

        vm.prank(buyer);
        coffer.holderRedeemBondInDefault(sellerBond);
    }
    vm.deal(address(coffer), address(coffer).balance + SWEEP);
    vm.prank(buyer);
    coffer.holderRedeemBondInDefault(sellerBond);
}

function test_control_honestSale_buyerGetsTheQuotedValue() public {
    (uint256 price, uint256 remaining) = _listAndQuote();
    uint256 b0 = buyer.balance;
    vm.prank(buyer);
    venue.buy{value: price}(address(bondNft), sellerBond);
    _buyerCollectsEverything();
    assertEq(buyer.balance + price - b0, remaining, "buyer receives the full
remaining value");
}

function test_sellerClaimsBeforeTheFill_buyerPaysForValueAlreadyTaken() public {
    (uint256 price, uint256 remaining) = _listAndQuote();
    uint256 b0 = buyer.balance;
    uint256 s0 = seller.balance;

    vm.prank(seller);
    coffer.holderRedeemBondInDefault(sellerBond);
    assertEq(_remaining(sellerBond), remaining - TRANCHE, "same token id, smaller
claim");

    vm.prank(buyer);
    venue.buy{value: price}(address(bondNft), sellerBond);
    _buyerCollectsEverything();

    assertEq(seller.balance - s0, price + TRANCHE, "seller keeps the price and the
claimed tranche");
    assertEq(buyer.balance + price - b0, remaining - TRANCHE, "buyer receives the
remaining value minus the tranche");
    emit log_named_decimal_int("buyer PnL", int256(buyer.balance) - int256(b0),
18);
}
}

```

Observed output:

```

[PASS] test_control_honestSale_buyerGetsTheQuotedValue()
[PASS] test_sellerClaimsBeforeTheFill_buyerPaysForValueAlreadyTaken()
    buyer PnL: -5.790100000000000000

```

Recommendations

Retire the old token id on every partial claim and mint the remainder under a new id. Extend BondRedeemedPartially with that new id so holders and indexers can follow the claim:

```
@@ -153,12 +153,14 @@ contract Coffe is Ownable2Step, Multicall, Initializable {
    /* solhint-disable gas-indexed-events */
    /// @notice Emitted when a holder redeems a bond partially - only ever fires while
the validator is defaulted
    /// @param holderAddress The address of the bond holder
-    /// @param bondId The ID of the bond NFT
+    /// @param oldBondId The ID of the bond NFT that was partially redeemed and burned
+    /// @param newBondId The ID of the bond NFT minted for the remaining value
    /// @param valueWithdrawn The amount withdrawn
    /// @param remainingBondMaturityValue The remaining bond maturity value
    event BondRedeemedPartially(
        address indexed holderAddress,
-        uint256 indexed bondId,
+        uint256 indexed oldBondId,
+        uint256 indexed newBondId,
        uint128 valueWithdrawn,
        uint128 remainingBondMaturityValue
    );
@@ -683,12 +685,14 @@ contract Coffe is Ownable2Step, Multicall, Initializable {
    // forge-lint: disable-next-line(unsafe-typecast) balance < remaining
(uint128), so it fits
    uint128 pay = uint128(balance);

-    holder.bondMaturityValue -= pay;
-
-    emit BondRedeemedPartially(msg.sender, _bondId, pay,
holder.bondMaturityValue);
-
-    ICoffeBondNft(iCoffeBondNftAddress()).emitMetadataUpdate(_bondId);
-
+    HolderConditions memory rest =
+        HolderConditions(holder.bondMaturityValue - pay, holder.duration,
holder.startTimestamp);
+    delete sHolderConditions[_bondId];
+    ICoffeBondNft nft = ICoffeBondNft(iCoffeBondNftAddress());
+    nft.burnCoffeBond(_bondId);
+    uint256 newId = nft.mintCoffeBond(msg.sender);
+    sHolderConditions[newId] = rest;
+    emit BondRedeemedPartially(msg.sender, _bondId, newId, pay,
rest.bondMaturityValue);
    Address.sendValue payable(msg.sender), pay);
} else {
    // Full settlement: the balance covers the bond's remaining value
```

If declareDefault is changed to reserve a covered bond's maturity value instead of reverting, release the old id's reservation in this branch before minting the new id: any path that deletes or re-keys a bond record must release that bond's

reservation. This invalidates every outstanding ERC-721 order for the old id. The tradeoff is an additional burn and mint per partial claim plus token-id changes for wallets and indexers. If stable ids are required, prevent transfers while defaulted or adopt an order mechanism that enforces the current remaining value; an ERC-7496 trait protects buyers only where the venue binds it.

Mitigation

Acknowledged. Commit a055d1e, the holder documentation and the risk guide now explain that a partial default claim reduces the remaining value without changing the ERC-721 token ID. Integrations are instructed to validate remaining value in the same transaction that transfers the bond. The deployed token behavior is unchanged, so stale listings on external venues remain an accepted integration risk.

7.2 Low Severity Findings

[L-01] Bonds bought after February 2106 are mature at purchase because startTimestamp is truncated to uint32

Affected files

```
coffer-smart-contracts/src/Coffer.sol#L62-L67
coffer-smart-contracts/src/Coffer.sol#L395-L402
coffer-smart-contracts/src/Coffer.sol#L636-L641
coffer-smart-contracts/src/Coffer.sol#L725-L730
```

Description

buyBond stores the purchase time as `uint32(block.timestamp)`. Once `block.timestamp` passes `type(uint32).max` (7 February 2106, 06:28:15 UTC) the cast silently wraps, so a bond bought at time $2^{32} + k$ records `startTimestamp = k`. Both maturity checks widen the stored values and compare them with the full timestamp:

```
uint256(holder.duration) + holder.startTimestamp <= block.timestamp
```

Every allowed duration is below 2^{32} (`MAX_DURATION` is 50 years), so $k + \text{duration} < k + 2^{32} \leq \text{block.timestamp}$ already holds in the purchase block: every bond sold after the rollover is mature the moment it is minted. The widened arithmetic and the `NatSpec` on the cast ("fits uint32 until year 2106") handle bonds bought before 2106 that mature after it, but not purchases made after the boundary. Coffer clones, the factory and the bond NFT are not upgradeable, so a Coffer deployed now that is still selling in 2106 cannot be repaired in place.

Impact: After February 2106, an ordinary buyer can immediately redeem a fresh long-duration bond and take the full term's interest from the validator or pooled execution balance that already covers matured holders; an uncovered fresh bond can instead trigger default and open the validator-exit path at purchase.

Recommendations

Widen the stored start time. `HolderConditions` still fits one storage slot ($128 + 32 + 64 = 224$ bits):

```
struct HolderConditions {
    uint128 bondMaturityValue;
    uint32 duration;
-   uint32 startTimestamp;
+   uint64 startTimestamp;
}
@@ buyBond
-   // forge-lint: disable-next-line(unsafe-typecast) block.timestamp fits
uint32 until year 2106
-   startTimestamp: uint32(block.timestamp),
+   startTimestamp: uint64(block.timestamp),
```

Apply the same type to the readers of the public `sHolderConditions` getter: `ICoffer.sHolderConditions` and `CofferBondNft.tokenURI` in this repository, and `ICoffer / CofferMarketplace.getBondData` in the marketplace repository, plus any off-chain decoder that reads the third return value as `uint32`.

A checked narrowing (`SafeCast.toUint32`) also removes the early maturity, but turns it into every `buyBond` reverting after 2106; widening keeps the Coffer usable.

Proof of Concept

Add the test below as `test/unit/TimestampRollover.t.sol` and run:

```
forge build # once: BaseTest loads the DepositContract mock from its build artifact
forge test --match-path test/unit/TimestampRollover.t.sol -vv
```

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.34;

import {BaseTest} from "../BaseTest.sol";
import {Coffer} from "../../src/Coffer.sol";

contract TimestampRolloverTest is BaseTest {
    Coffer private target;
    address private targetAddress;

    function setUp() public override {
        super.setUp();
        targetAddress = createDefaultCoffer(); // durations 1 month .. 1 year
        target = Coffer(payable(targetAddress));
    }

    function _defaulted() private view returns (bool d) {
        (,,,,,,,, d) = target.sValidatorConditions();
    }

    /// A one-year bond bought after the uint32 rollover is paid its full maturity
    value in the same block.
    function test_PostRollover_OneYearBondPaidImmediately() public {
        vm.warp(uint256(type(uint32).max) + 2);

        vm.prank(holder1);
        uint256 bondId = target.buyBond{value: 5 ether}(ONE_YEAR, 1);
        (uint128 maturityValue,,) = target.sHolderConditions(bondId);

        // The validator keeps the contract funded, as it must at all times
        vm.prank(validator);
        (bool funded,) = targetAddress.call{value: maturityValue}("");
        assertTrue(funded);

        uint256 holderBefore = holder1.balance;
        vm.prank(holder1); // same block as the purchase, no time has passed
        bool paidInFull = target.holderRedeemBondOrDefault(bondId);

        assertTrue(paidInFull, "bond not paid");
        assertEq(holder1.balance - holderBefore, maturityValue, "one year of interest
paid on day zero");
        assertGt(maturityValue, 5 ether, "maturity value includes the full year of
interest");
    }
}
```

```

    /// The same bond on an unfunded Coffey lets anyone declare a default in the
    purchase block.
    function test_PostRollover_FreshBondDefaultsImmediately() public {
        vm.warp(uint256(type(uint32).max) + 2);

        vm.prank(holder1);
        uint256 bondId = target.buyBond{value: 5 ether}(ONE_YEAR, 1);

        target.declareDefault(bondId); // anyone, same block
        assertTrue(_defaulted(), "fresh one-year bond triggered a default");
    }

    /// Sweep: every allowed duration, every post-rollover purchase time.
    function testFuzz_PostRollover_PurchasesImmediatelyMature(uint32 offset, uint32
duration) public {
        duration = uint32(bound(duration, defaultMinDuration, defaultMaxDuration));
        vm.warp(uint256(type(uint32).max) + 1 + uint256(offset));

        vm.prank(holder1);
        uint256 bondId = target.buyBond{value: 5 ether}(duration, 1);
        (uint128 maturityValue,,) = target.sHolderConditions(bondId);

        vm.prank validator);
        (bool funded,) = targetAddress.call{value: maturityValue}("");
        assertTrue(funded);

        uint256 holderBefore = holder1.balance;
        vm.prank(holder1);
        target.holderRedeemBondOrDefault(bondId);
        assertEq(holder1.balance - holderBefore, maturityValue, "matured at purchase");
    }
}

```

Output at 793f6a7:

Ran 3 tests for test/unit/TimestampRollover.t.sol:TimestampRolloverTest

[PASS] testFuzz_PostRollover_PurchasesImmediatelyMature(uint32,uint32) (runs: 1024, μ: 198951, ~: 199024)

[PASS] test_PostRollover_FreshBondDefaultsImmediately() (gas: 217108)

[PASS] test_PostRollover_OneYearBondPaidImmediately() (gas: 194050)

Suite result: ok. 3 passed; 0 failed; 0 skipped

With the uint64 change above applied, all three tests fail with `HoldersTimeHasNotExpiredYet()`, including the fuzzed sweep over every allowed duration and purchase time after the rollover, and a bond bought after the rollover redeems exactly duration seconds after purchase.

Mitigation

Acknowledged. Commit 55868a1 documents the February 2106 rollover, its effect on new purchases, and the need to stop sales before the boundary. The stored timestamp remains `uint32`, so the executable condition is retained as an accepted long-horizon risk.

[L-02] The validator can void a third party's default declaration without paying

Affected files

coffer-smart-contracts/src/Coffer.sol#L717-L753
coffer-smart-contracts/src/Coffer.sol#L340-L346
coffer-smart-contracts/src/Coffer.sol#L824-L846
coffer-smart-contracts/README.md#L195-L197

Description

`declareDefault` is permissionless, and its NatSpec says the only useful front-run is one that pays the holder; the README likewise says preventing a default requires payment.

`_declareDefault` instead decides from the instantaneous balance of the named bond alone. The validator can sandwich a public declaration: send the bond's maturity value through `receive()` just before it and withdraw the same value through `validatorWithdrawFromExecution` just after. `receive()` raises `issueSize`, so the withdrawal stays within bounds. The declaration reverts, nobody is paid, no default stands, and this repeats.

A second route needs no ordering. The README states the duty as the sum of all matured, unpaid bonds, but the check is per bond: with two or more matured bonds whose holders do not act, a balance equal to the largest one makes every third-party declaration revert.

The matured holder's own `holderRedeemBondOrDefault` is not affected: it pays that holder or declares the default atomically.

Impact: Other holders and keepers can lose timely permissionless default enforcement, acceleration, and validator exit while the validator sandwiches public declarations or holds the balance at the largest matured bond; the matured holder or a privately ordered declaration can still end the delay, so no direct loss is established.

Recommendations

When a covered `declareDefault` call arrives, reserve the named bond's maturity value instead of reverting. Track the reservation per bond and in an aggregate, and require `validatorWithdrawFromExecution` to leave the aggregate in the Coffer.

Keep a per-bond reservation and an aggregate. Release a bond's reservation, by subtracting exactly that bond's amount and deleting its entry, on every path that deletes or re-keys the bond's record, in either state (full settlement, `validatorRedeemBonds`, and any partial-claim path that re-issues the bond), and do so before any external call. Never reset the aggregate in bulk. Outside a default, test coverage and bound withdrawals against the balance minus other bonds' reservations; while defaulted, ignore reservations so claims keep paying the smaller of the balance and the remaining value. If another bond is genuinely uncovered after existing reservations are excluded, enter global default. `clearDefault` can additionally require the aggregate to be zero. This changes a covered declaration from a revert into a successful reservation and adds storage writes; tests that assert the old revert or top-up/re-extract behavior must be updated.

The reservation also closes the second route: a reserved bond is covered, so the next matured bond becomes declarable against the remaining balance.

If transient coverage and per-bond coverage are intended, instead correct the NatSpec and README to state that only `holderRedeemBondOrDefault` is ordering-safe, that third-party declarations require private ordering, and that a third party can declare a default only when a single matured bond exceeds the contract balance.

Proof of Concept

Add to `test/unit/`, run `forge build once` (the `BaseTest` fixture loads the `DepositContract` mock from its build artifact), then `forge test --match-path test/unit/ThirdPartyDefaultDodged.t.sol -vv`.

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.34;
```

```

import {BaseTest} from "../BaseTest.sol";
import {Coffer} from "../../src/Coffer.sol";

contract ThirdPartyDefaultDodged is BaseTest {
    function setUp() public override {
        super.setUp();
        coffer = Coffer(payable(createDefaultCoffer()));
    }

    function test_validatorVoidsThirdPartyDeclarationWithoutPaying() public {
        (,,,,, uint32 version,,,,) = coffer.sValidatorConditions();
        uint256 matured = buyBond(address(coffer), holder1, 5 ether,
defaultMinDuration, version); // holder1 inactive
        uint256 other = buyBond(address(coffer), holder2, 5 ether, defaultMaxDuration,
version); // holder2 enforces
        advanceTime(defaultMinDuration);
        (uint128 mv,,) = coffer.sHolderConditions(matured);
        assertEq(address(coffer).balance, 0, "matured bond is unfunded");
        uint256 validatorBefore = validator.balance;

        // Same block: the validator tops up just before holder2's declaration...
        vm.prank(validator);
        (bool ok,) = address(coffer).call{value: mv}("");
        assertTrue(ok);
        vm.prank(holder2);
        vm.expectRevert(Coffer.ValidatorNotDefaultable.selector);
        coffer.declareDefault(matured);
        // ...and takes the same ETH back right after it.
        vm.prank(validator);
        coffer.validatorWithdrawFromExecution(mv);

        (,,,,,,,, bool defaulted) = coffer.sValidatorConditions();
        assertEq(validator.balance, validatorBefore, "validator paid nothing");
        assertEq(address(coffer).balance, 0, "matured bond unfunded again");
        assertFalse(defaulted, "no default stands");
        (uint128 stillOwed,,) = coffer.sHolderConditions(matured);
        assertEq(stillOwed, mv, "matured bond still unpaid");

        // holder2 cannot accelerate its bond or force the exit; the validator can keep
        extracting.
        vm.prank(holder2);
        vm.expectRevert(Coffer.ValidatorNotInDefault.selector);
        coffer.holderRedeemBondInDefault(other);
    }

    /// Control: without the validator's top-up/withdraw around it, the same
    declaration lands.
    function test_control_declarationLands() public {

```

```

        (,,,,, uint32 version,,,,) = coffer.sValidatorConditions();
        uint256 matured = buyBond(address(coffer), holder1, 5 ether,
defaultMinDuration, version);
        buyBond(address(coffer), holder2, 5 ether, defaultMaxDuration, version);
        advanceTime(defaultMinDuration);
        vm.prank(holder2);
        coffer.declareDefault(matured);
        (,,,,,,, bool defaulted) = coffer.sValidatorConditions();
        assertTrue(defaulted);
    }

    /// Second route: with two matured bonds whose holders do not act, a balance equal
    to the larger
    /// one blocks every third-party declaration, with no transaction ordering.
    function test_balanceEqualToLargestMaturedBondBlocksThirdPartyDefault() public {
        (,,,,, uint32 version,,,,) = coffer.sValidatorConditions();
        uint256 x = buyBond(address(coffer), holder1, 5 ether, defaultMinDuration,
version); // passive
        uint256 y = buyBond(address(coffer), holder2, 5 ether, defaultMinDuration,
version); // passive
        uint256 z = buyBond(address(coffer), holder3, 5 ether, defaultMaxDuration,
version); // enforces
        advanceTime(defaultMinDuration);
        (uint128 mvX,,) = coffer.sHolderConditions(x);
        (uint128 mvY,,) = coffer.sHolderConditions(y);

        vm.prank(validator);
        (bool ok,) = address(coffer).call{value: mvX}("");
        assertTrue(ok);
        assertLt(address(coffer).balance, uint256(mvX) + mvY, "matured debt not
covered");

        for (uint256 week = 0; week < 6; ++week) {
            vm.prank(holder3);
            vm.expectRevert(Coffer.ValidatorNotDefaultable.selector);
            coffer.declareDefault(x);
            vm.prank(holder3);
            vm.expectRevert(Coffer.ValidatorNotDefaultable.selector);
            coffer.declareDefault(y);
            vm.prank(holder3);
            vm.expectRevert(Coffer.ValidatorNotInDefault.selector);
            coffer.holderRedeemBondInDefault(z);
            advanceTime(1 weeks);
        }
        (,,,,,,, bool defaulted) = coffer.sValidatorConditions();
        assertFalse(defaulted, "six weeks later, still no default");
    }
}

```

Output at 793f6a7:

```
[PASS] test_balanceEqualToLargestMaturedBondBlocksThirdPartyDefault() (gas: 571602)
```

```
[PASS] test_control_declarationLands() (gas: 378593)
```

```
[PASS] test_validatorVoidsThirdPartyDeclarationWithoutPaying() (gas: 409627)
```

Mitigation

Acknowledged. Commit 2f4ecd9 documents that `holderRedeemBondOrDefault` is the ordering-safe path and that a third-party `declareDefault` should use private ordering. The contract does not reserve the named bond during a declaration, so the public third-party path remains front-runnable by a top-up and withdrawal.

[L-03] Redeem Bonds guidance omits funding reserved for other matured bonds

Affected files

coffer-smart-contracts/src/Coffer.sol#L436-L441

<https://docs.coffer.bond/validators/redeem-bonds/>

<https://docs.coffer.bond/validators/funds/>

Description

The Redeem Bonds guide describes the app's Value to send field as `max(0, selectedPayout - cofferBalance)` and instructs validators to send that amount. This omits funds needed by matured bonds outside the selected batch. The contract's NatSpec explicitly requires including those unpaid maturities; the contract itself checks only that the selected batch can be paid.

A validator can fund a matured bond, then redeem a different bond using the guide's amount. The redemption spends the first bond's funding. Its holder's subsequent redemption receives no ETH and declares a default, opening permissionless exit requests while that bond remains outstanding.

Following this guidance can therefore interrupt a funded holder's payment and expose the validator to an avoidable default. The validator can restore payment by topping up and settling the remaining claims; an accepted consensus-layer exit request cannot be recalled. This is a funding-guidance defect, rather than an additional contract guard failure. The guide's assurance that redemption is never detrimental should apply only to the selected holders.

Recommendations

While the Coffer is not defaulted, size the guide's recommended payment and the corresponding app field as:

```
required = selectedPayout + maturedUnpaidOutsideBatch
valueToSend = max(0, required - preTransactionCofferBalance)
```

Exclude selected bonds from the matured-unpaid sum. Read outstanding bond values, maturities and balance from consistent, fresh state, and account for bonds maturing while the transaction is pending. This can require more ETH from the validator and additional bond-state reads. Qualify the holder-benefit assurance and explain that claims during default follow the default settlement rules; this formula does not guarantee full funding of all accelerated claims or undo an exit request.

Apply the same reserve to the Withdraw · Execution panel's "Available" figure, which the Funds guide presents as the liquid balance bounded by the issue size. ETH sent to fund a maturity is also credited to `issueSize`, so until the holder redeems, that funding shows as withdrawable; display `min(issueSize, balance - maturedUnpaid)` instead.

PoC

Add to `test/unit/`, run `forge build once` (the `BaseTest` fixture loads the `DepositContract` mock from its build artifact), then `forge test --match-path test/unit/RedeemSizing.t.sol -vv`. The first test sizes `msg.value` as the app does; the second sizes it as the NatSpec does.

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.34;

import {BaseTest} from "../BaseTest.sol";
import {Coffer} from "../../src/Coffer.sol";

contract RedeemSizing is BaseTest {
    Coffer internal c;
```

```

    function _vc() internal view returns (uint128 issueSize, uint32 version, uint32
outstanding, uint16 bufferBps, bool defaulted) {
        (issueSize,,,,, version, outstanding, bufferBps,, defaulted) =
c.sValidatorConditions();
    }

    function _mv(uint256 id) internal view returns (uint128 m) {
        (m,,) = c.sHolderConditions(id);
    }

    function test_D1_docsSizedValidatorRedemptionDefundsMaturedHolder() public {
        c = Coffe(payable(createDefaultCoffe()));
        (, uint32 v,,, ) = _vc();
        uint256 b1 = buyBond(address(c), holder1, 10 ether, ONE_MONTH, v); // matures
first
        uint256 b2 = buyBond(address(c), holder2, 10 ether, ONE_YEAR, v); // long bond
        uint128 m1 = _mv(b1);
        uint128 m2 = _mv(b2);

        // Honest validator funds bond 1 before its maturity (docs: "keep the contract
funded")
        vm.prank(validator);
        (bool ok,) = address(c).call{value: m1}("");
        assertTrue(ok);
        advanceTime(ONE_MONTH + 1 days); // bond 1 matured, holder1 has not redeemed
yet (no deadline in docs)

        // Validator settles bond 2 early following the redeem-bonds page sizing:
max(0, payout - balance)
        uint256 docsValue = m2 > address(c).balance ? m2 - address(c).balance : 0;
        uint256[] memory ids = new uint256[](1);
        ids[0] = b2;
        vm.prank(validator);
        c.validatorRedeemBonds{value: docsValue}(ids);

        // holder1 now redeems at maturity, as documented
        uint256 h1Before = holder1.balance;
        vm.prank(holder1);
        bool paid = c.holderRedeemBondOrDefault(b1);
        (,,, bool defaulted) = _vc();
        assertFalse(paid, "holder1 not paid");
        assertTrue(defaulted, "default declared");
        assertEq(holder1.balance, h1Before, "holder1 received nothing");
        assertEq(address(c).balance, 0, "funding of bond 1 was spent on bond 2");

        // and anyone can now force the honest validator's full exit
        vm.deal(unauthorizedUser, 1 ether);
        vm.prank(unauthorizedUser);
        c.exitValidator{value: getWithdrawalFee()}();
    }

```

```

    }

    /// Counterfactual: NatSpec sizing (payout + maturedUnpaid - balance) keeps holder1
    paid.
    function test_D1_natspecSizedRedemptionKeepsMaturedHolderPaid() public {
        c = Coffer(payable(createDefaultCoffer()));
        (, uint32 v,,,) = _vc();
        uint256 b1 = buyBond(address(c), holder1, 10 ether, ONE_MONTH, v);
        uint256 b2 = buyBond(address(c), holder2, 10 ether, ONE_YEAR, v);
        uint128 m1 = _mv(b1);
        uint128 m2 = _mv(b2);
        vm.prank(validator);
        (bool ok,) = address(c).call{value: m1}("");
        assertTrue(ok);
        advanceTime(ONE_MONTH + 1 days);
        uint256 natspecValue = uint256(m2) + m1 - address(c).balance;
        uint256[] memory ids = new uint256[](1);
        ids[0] = b2;
        vm.prank(validator);
        c.validatorRedeemBonds{value: natspecValue}(ids);
        vm.prank(holder1);
        assertTrue(c.holderRedeemBondOrDefault(b1));
    }
}

```

[PASS] test_D1_docsSizedValidatorRedemptionDefundsMaturedHolder()

[PASS] test_D1_natspecSizedRedemptionKeepsMaturedHolderPaid()

Mitigation

Fixed. Commit fdd83b8, the public documentation and the live application now size a validator redemption as the selected payout plus matured unpaid bonds outside the batch, minus the Coffer balance. The application refreshes this minimum before submission and excludes matured unpaid liabilities from the amount shown as withdrawable.

[L-04] Docs time consensus funding at "roughly two days", but partial withdrawals wait in the exit queue, so a validator following them defaults

Affected files

<https://docs.coffer.bond/validators/funds/>
<https://docs.coffer.bond/validators/overview/>
[coffer-smart-contracts/src/Coffer.sol#L873-L896](https://github.com/coffer-smart-contracts/src/Coffer.sol#L873-L896)
[coffer-smart-contracts/README.md#L195](https://github.com/coffer-smart-contracts/README.md#L195)

Description

The README requires the contract balance to cover every matured, unpaid bond and correctly says consensus partials take about 28 hours only with an empty exit queue.

The public Funds guide instead says a request "waits in the beacon queue at least about 27 hours" and arrives "roughly two days after the request". Electra schedules a partial through `compute_exit_epoch_and_update_churn` and then adds 256 epochs. On 15 October 2026 the exit queue was about 5.4 days, so a request arrived after about 6.5 days.

A validator that follows the guide and requests three days ahead is short at maturity. The holder's `holderRedeemBondOrDefault` declares default, every bond accelerates, and anyone can force the exit. A pending partial makes the consensus layer drop the first exit request, but `exitValidator` is permissionless and retryable after the partial is processed. Holders are paid in full; the validator loses its validator and repays a 365-day bond about 271 days early-about 0.45 ETH of time value on 30 ETH at 2%.

The Validators overview adds that consensus rewards "flow to the contract on their own", which a 0x02 validator below 2,048 ETH does not get.

Recommendations

State the consensus funding time as the current exit-queue wait plus about 27 hours, and for an Exit up to one more withdrawal-sweep cycle (about a week at today's validator count), link a live queue source, and show the expected arrival epoch in the Withdraw · Consensus panel before the validator signs. Refresh the estimate when the panel opens and present it as approximate because queue state can change. Point to `receive()` top-ups as the only instant way to fund a maturity, and say that a request should be counted as funding only once it has arrived.

Apply the same correction to the "about two days" on the validator and holder Default pages. Remove "Your consensus rewards flow to the contract on their own" from the Validators overview, or limit it to the excess above 2,048 ETH for 0x02 validators.

PoC

Add to `test/unit/`, run `forge build` once (the `BaseTest` fixture loads the `DepositContract` mock from its build artifact), then `forge test --match-path test/unit/Y03FundingLatency.t.sol -vv`. The consensus layer is modelled: the partial withdrawal arrives after the exit-queue wait measured on 15 October 2026 (about 5.4 days) plus 256 epochs. The control uses the guide's two days, and the third test funds with the queue-aware timing.

```
//SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.34;

import {BaseTest} from "../BaseTest.sol";
import {Coffer} from "../../src/Coffer.sol";

/// Y03: an honest validator funds a maturity through Withdraw · Consensus, timed by
the docs'
/// "the ETH arrives in the contract with the beacon sweep, roughly two days after the
```

```

request".
/// The consensus layer is MODELLED: Electra process_withdrawal_request puts a partial
through
/// compute_exit_epoch_and_update_churn (the shared exit churn, 256 ETH/epoch) and then
/// MIN_VALIDATOR_WITHDRAWABILITY_DELAY (256 epochs ~27.3h). Live exit queue 2026-10-15
/// (measured from beacon state, 2026-10-15): about 5.4 days. The partial's arrival is
applied with deal().
contract Y03FundingLatency is BaseTest {
    uint32 constant RATE_2PCT = 2_000_000;
    uint32 constant D90 = 90 days;
    uint32 constant D365 = 365 days;
    uint16 constant BUF = 200;
    uint128 constant START = 64 ether; // a 0x02 validator with excess above 32 ETH, so
partials can fund
    uint256 constant WITHDRAWABILITY = 256 * 384; // 256 epochs * 6.4 min
    uint256 constant EXIT_QUEUE_NOW = 5 days + 9 hours; // measured 2026-10-15 from
beacon state: newest pending partial withdrawable_epoch - 256 - head = 1,209 epochs
    uint256 constant DOCS_LATENCY = 2 days; // "roughly two days after the request"

    Coffer c;
    uint256 bondA;
    uint256 bondB;
    uint128 mA;
    uint128 mB;
    uint256 t0;

    function setUp() public override {
        super.setUp();
        address a = createCoffer(
            validator, validPublicKeyPart1, validPublicKeyPart2, RATE_2PCT, D90, D365,
1 ether, BUF, START
        );
        c = Coffer(payable(a));
        t0 = block.timestamp;
        (,,,,, uint32 v,,,,) = c.sValidatorConditions();
        bondA = buyBond(a, holder1, 20 ether, D90, v); // matures day 90
        bondB = buyBond(a, holder2, 30 ether, D365, v); // matures day 365
        (mA,,) = c.sHolderConditions(bondA);
        (mB,,) = c.sHolderConditions(bondB);
    }

    /// Validator requests a partial of mA `lead` seconds before bond A matures; it
lands `latency` later.
    function _run(uint256 lead, uint256 latency) internal returns (bool defaulted,
uint256 bPaidAt) {
        uint256 maturityA = t0 + D90;
        vm.warp(maturityA - lead);
        uint256 fee = uint256(bytes32(_fee()));
        vm.prank(validator);

```

```

        c.validatorWithdrawFromConsensus{value: fee * 6}(uint64(mA / 1 gwei) + 1); //
app sends 6x the fee
        uint256 landsAt = maturityA - lead + latency;
        uint256 bal = address(c).balance;

        if (landsAt <= maturityA) {
            vm.warp(landsAt);
            vm.deal(address(c), address(c).balance + mA + 1 gwei);
        }
        // Holder A redeems at maturity, as the docs tell it to
        vm.warp(maturityA);
        vm.prank(holder1);
        bool paid = c.holderRedeemBondOrDefault(bondA);
        (,,,,,, defaulted) = c.sValidatorConditions();
        assertEq(paid, !defaulted);

        if (defaulted) {
            // Holder B forces the validator's exit (irrevocable) and waits for funds
            uint256 f2 = uint256(bytes32(_fee()));
            vm.deal(holder2, holder2.balance + f2);
            vm.prank(holder2);
            c.exitValidator{value: f2}();
            // The consensus partial finally lands
            vm.warp(landsAt);
            vm.deal(address(c), address(c).balance + mA + 1 gwei);
            vm.prank(holder1);
            c.holderRedeemBondInDefault(bondA);
            // B is accelerated: claimable now at full maturity value; the rest of the
            stake sweeps in later
            vm.deal(address(c), address(c).balance + 44 ether); // modelled exit sweep
            of the remaining stake
            uint256 before = holder2.balance;
            vm.prank(holder2);
            c.holderRedeemBondInDefault(bondB);
            assertEq(holder2.balance - before, mB, "B paid in full, early");
            bPaidAt = block.timestamp;
        } else {
            bPaidAt = t0 + D365;
        }
        bal;
    }

    function _fee() internal view returns (bytes memory d) {
        (, d) = address(0x00000961Ef480Eb55e80D19ad83579A64c007002).staticcall("");
    }

    /// Docs-literal: request 3 days ahead (more than "roughly two days"); today's
    queue applies.
    function test_Y03_docsTimingDefaultsHonestValidator() public {

```

```

        (bool defaulted, uint256 bPaidAt) = _run(3 days, EXIT_QUEUE_NOW +
WITHDRAWABILITY);
        assertTrue(defaulted, "honest validator defaulted");
        uint256 early = t0 + D365 - bPaidAt;
        emit log_named_uint("mA (wei)", mA);
        emit log_named_uint("mB (wei)", mB);
        emit log_named_uint("B paid days before its maturity", early / 1 days);
        // Validator pays B ~259 days early (cost at its own 2% rate) and loses its
validator (forced exit)
        emit log_named_uint("time-value cost of acceleration at 2%/yr (wei)",
uint256(mB) * 2 * early / (100 * 365 days));
        assertGt(early, 250 days);
    }

    /// Control A: the docs' latency holds (empty exit queue) -> no default.
    function test_Y03_control_emptyQueue() public {
        (bool defaulted,) = _run(3 days, 5 * 384 + WITHDRAWABILITY);
        assertFalse(defaulted);
    }

    /// Mutation: guidance says "exit-queue wait + ~27h; check the live queue" ->
validator requests in time.
    function test_Y03_mutation_queueAwareGuidance() public {
        (bool defaulted,) = _run(EXIT_QUEUE_NOW + WITHDRAWABILITY + 1 days,
EXIT_QUEUE_NOW + WITHDRAWABILITY);
        assertFalse(defaulted);
    }
}

```

Output:

```

[PASS] test_Y03_control_emptyQueue()
[PASS] test_Y03_docsTimingDefaultsHonestValidator()
    B paid days before its maturity: 271
    time-value cost of acceleration at 2%/yr (wei): 455116716529680365
[PASS] test_Y03_mutation_queueAwareGuidance()

```

Additional validated locations for the funding-timing fix

The same fixed-time assumption appears in the in-scope NatSpec for `exitValidator` and `validatorWithdrawFromConsensus`. Both describe the pending tail as about 28 hours without stating that the exit-queue backlog at request time is added first. These comments should be corrected with the public guides.

The fix should also apply the issue's "count only once arrived" rule to `validatorAddFundsToConsensus`. That function increases `issueSize` immediately, while the deposit waits for consensus processing. Its all-zero signature cannot register a new validator, so a top-up made before a valid initial deposit reaches the key can be discarded. Buyer guidance should treat the added capacity as backed only after the validator's beacon balance or pending deposits show the amount, and should state that the function is intended for an already registered key or one whose valid initial deposit is earlier in the queue.

Mitigation

Fixed. Commits 83551d4 and 268c2c3 correct the in-scope NatSpec and consensus top-up guidance. The public documentation and application now state that a partial withdrawal waits in the exit queue and then roughly 28 hours, that a full exit also waits for the sweep, and that direct ETH transfer is the instant funding path. The application does not yet calculate a live pre-sign arrival estimate, but the reported fixed two-day assumption has been removed.

[L-05] Stale bond purchases can settle below the buyer's quoted maturity value

Affected files

coffer-smart-contracts/src/Coffer.sol#L361-L365
coffer-smart-contracts/src/Coffer.sol#L376-L382
coffer-smart-contracts/src/Coffer.sol#L396-L423
coffer-smart-contracts/src/FeeCurve.sol#L88-L107

Description

buyBond lets a buyer bind the duration and ValidatorConditions.version, but it reads the protocol fee from the time-dependent FeeCurve only when the transaction executes. Fee-curve time movement does not change the Coffer version. The call has no deadline, maximum fee, or minimum maturity value, so a queued or rebroadcast transaction can remain valid after the displayed quote has worsened.

The higher fee is deducted from interest before bondMaturityValue is stored, and the exact difference accrues to FeeCurve. At valid 1 ETH principal, 100% rate, and 50-year duration terms, crossing the day-89/day-90 boundary reduces the claim by 0.01 ETH. In the valid 1 ETH, 100%, 10-year long-staleness example, the same calldata stores 10.01 ETH instead of the quoted 10.9 ETH. Public transactions will usually be included or nonce-replaced sooner, and ordinary builders do not directly profit, which limits practical likelihood.

Recommendations

Require every purchase to bind a deadline, maximum fee rate, and minimum maturity value. Check the deadline before pricing, then check the fee and maturity bounds immediately after reading FeeCurve and before changing state or making external calls. For example, replace the unprotected entry point with:

```
function buyBond(  
    uint32 duration,  
    uint32 version,  
    uint256 maxFeeBps,  
    uint256 minMaturityValue,  
    uint256 deadline  
) external payable returns (uint256 bondId);
```

The frontend should derive these bounds from the quote shown to the buyer. Remove or disable the legacy unbounded selector in new deployments so integrations cannot silently keep using it.

Proof of Concept

Add the following test as test/FeeSlippage.t.sol and run:

```
forge test --match-path test/FeeSlippage.t.sol -vv
```

```
// SPDX-License-Identifier: BUSL-1.1  
pragma solidity 0.8.34;  
  
import {BaseTest} from "../unit/BaseTest.sol";  
import {Coffer} from "../src/Coffer.sol";  
import {Interest} from "../src/libraries/Interest.sol";  
  
contract FeeSlippageTest is BaseTest {  
    uint32 private constant FIFTY_YEARS = 1_576_800_000;
```

```

uint32 private constant TEN_YEARS = 315_360_000;
uint32 private constant MAX_RATE = 100_000_000;

function _newCoffer(bytes32 key, uint32 duration, uint128 balance)
    private
    returns (Coffer target)
{
    target = Coffer(payable(createCoffer(
        validator,
        key,
        bytes16(uint128(uint256(key))),
        MAX_RATE,
        duration,
        duration,
        1,
        0,
        balance
    )));
}

function _maturity(uint256 principal, uint32 duration, uint256 feeBps)
    private
    pure
    returns (uint256)
{
    uint256 interest = Interest.calculateInterest(principal, duration, MAX_RATE);
    return principal + interest - (interest * feeBps) / 10_000;
}

function test_adjacentFeeDayExecutesBelowQuotedClaim() public {
    Coffer target = _newCoffer(bytes32(uint256(101)), FIFTY_YEARS, 51 ether);
    uint256 start = feeCurve.START_TIME();
    vm.warp(start + 89 days + 23 hours + 59 minutes + 59 seconds);

    uint256 quotedClaim = _maturity(1 ether, FIFTY_YEARS,
feeCurve.currentFeeBps());
    bytes memory callData = abi.encodeCall(Coffer.buyBond, (FIFTY_YEARS, 1));

    vm.warp(start + 90 days);
    uint256 feeBefore = feeCurve.sAccruedFees();
    vm.prank(holder1);
    (bool ok, bytes memory ret) = address(target).call{value: 1 ether}(callData);
    assertTrue(ok);

    uint256 bondId = abi.decode(ret, (uint256));
    (uint128 storedClaim,,) = target.sHolderConditions(bondId);
    uint256 feeDelta = feeCurve.sAccruedFees() - feeBefore;

    assertEq(quotedClaim - storedClaim, 0.01 ether);
}

```

```

    assertEq(feeDelta, 0.985 ether);
    assertEq(feeDelta - 0.975 ether, quotedClaim - storedClaim);
}

function test_longStaleTransactionExecutesBelowQuotedClaim() public {
    Coffer target = _newCoffer(bytes32(uint256(202)), TEN_YEARS, 12 ether);
    uint256 start = feeCurve.START_TIME();
    vm.warp(start);

    uint256 quotedClaim = _maturity(1 ether, TEN_YEARS, feeCurve.currentFeeBps());
    bytes memory callData = abi.encodeCall(Coffer.buyBond, (TEN_YEARS, 1));

    vm.warp(start + 3650 days);
    uint256 feeBefore = feeCurve.sAccruedFees();
    vm.prank(holder1);
    (bool ok, bytes memory ret) = address(target).call{value: 1 ether}(callData);
    assertTrue(ok);

    uint256 bondId = abi.decode(ret, (uint256));
    (uint128 storedClaim,,) = target.sHolderConditions(bondId);
    uint256 feeDelta = feeCurve.sAccruedFees() - feeBefore;

    assertEq(quotedClaim, 10.9 ether);
    assertEq(storedClaim, 10.01 ether);
    assertEq(quotedClaim - storedClaim, 0.89 ether);
    assertEq(feeDelta - 0.1 ether, quotedClaim - storedClaim);
}
}

```

Observed output at 793f6a78d805bfa2c83320633dc5984eca1038db:

```

[PASS] test_adjacentFeeDayExecutesBelowQuotedClaim()
[PASS] test_longStaleTransactionExecutesBelowQuotedClaim()
Suite result: ok. 2 passed; 0 failed; 0 skipped.

```

Additional purchase-state binding

When implementing the quoted-value and deadline bounds, also consider binding the Coffer owner or bumping version when ownership changes. `buyBond` validates version and then pays the current owner(), while inherited `Ownable2Step` ownership acceptance does not increment the Coffer version.

A transaction prepared while one validator address owns the Coffer can therefore execute after another address accepts ownership, paying the new owner without forcing the buyer to re-confirm. The bond terms and backing do not change, so this does not warrant a separate severity finding, but an `expectedOwner` bound or a version bump on ownership transfer closes the same stale-purchase trust gap while the purchase interface is being strengthened.

Mitigation

Acknowledged. Commits `fcdfbbb` and `1309c97` document that `buyBond` uses the fee, bond terms and owner that exist when the transaction lands. No deadline, maximum fee, minimum maturity value or expected owner is bound in `calldata`, so adverse landing-time changes remain an accepted risk.

7.3 Informational Severity Findings

[I-01] Default documentation incorrectly states that every validator extraction path is frozen

Description

The `validatorDefaulted`, `NatSpec`, `declareDefault`, and `holderRedeemBondOrDefault` state that every validator extraction path is frozen during a default.

`validatorRedeemBonds` has no default check and deliberately remains callable. It lets the validator select bonds and pay their full maturity values from the pooled Coffer balance into the redemption escrow, including bonds the validator acquired through another address. The README later describes this exception correctly, so the executable behavior is consistent with the intended first-come-first-served settlement model, but the earlier `NatSpec` and README statements are false.

Holders and integrations relying on those statements may incorrectly assume that the validator cannot direct any Coffer balance while a default stands.

Recommendations

State consistently that every validator extraction path **except `validatorRedeemBonds`** is frozen during a default. Add the exception to the `validatorDefaulted`, `declareDefault`, and `holderRedeemBondOrDefault` `NatSpec`, and document on `validatorRedeemBonds` that it remains available as the settlement path and lets the validator choose which full-value claims the pooled balance settles first.

Mitigation

Fixed. Commit `dc1c05f` corrects the README, `NatSpec` and test comments to state that `validatorRedeemBonds` remains available during default as a settlement path into the pull-payment escrow.

[I-02] Marketplace checklist omits validator-to-Coffer credential binding

Description

The marketplace README's bond-assessment checklist tells a buyer to read the stake from the beacon chain, but it does not say to derive the validator public key from the Coffer and confirm that the validator's withdrawal credentials point back to that Coffer.

The core README makes this the holder's first duty because `CofferFactory.createCoffer` accepts any public key and the contracts cannot read beacon state. If a buyer reads another validator's stake, or relies on a deposit event for a key that has not yet registered, that stake may never reach the Coffer and its exit requests may be ignored.

Secondary-market buyers following only the marketplace checklist can therefore price a bond as stake-backed when the named validator is not bound to its Coffer.

Recommendations

Make credential binding the first marketplace assessment step. Buyers should read `iPublicKeyPart1()` and `iPublicKeyPart2()` from the Coffer, find that validator in beacon state, and verify `0x01` or `0x02` withdrawal credentials whose address equals the Coffer address.

State that an unregistered public key cannot pass this check and that deposit events are not proof of registered withdrawal credentials. Repeat the requirement in the marketplace's default and impairment guidance, before asking buyers to evaluate stake or coverage.

Mitigation

Fixed. Commit 9383984 makes validator-to-Coffer credential binding the first marketplace assessment step. It explains how to reconstruct the withdrawal credential, rejects unregistered and `0x00` credentials, and states that a deposit event is not proof of binding.

[I-03] Deployment leaves the shared FeeCurve owned by the deployer key

Description

The in-scope `DeployCofferFactory` script broadcasts new `CofferFactory(feeRecipient)` from the deployer key and performs no ownership handover. The `CofferFactory` constructor creates the shared `FeeCurve` with `msg.sender` as owner, so the documented deployment leaves that role on the deployer EOA with no pending transfer.

The `FeeCurve` owner can replace the fee recipient, and `claim()` sends every pooled, unclaimed fee to the recipient configured at claim time. Until a manual two-step handover finishes, compromise or misuse of the deployer key can redirect fees collected across all Coffers.

This is an operational deployment defect rather than an unprivileged protocol attack, so Informational severity is appropriate before mainnet deployment.

Recommendations

Accept a `PROTOCOL_ADMIN Safe` address in the deployment flow, require it to contain code, and call `FeeCurve.transferOwnership(PROTOCOL_ADMIN)` in the broadcast. Complete `acceptOwnership()` from the Safe before the first bond purchase and make the launch check fail unless `owner()` equals the intended Safe and `pendingOwner()` is zero.

A constructor parameter for the initial `FeeCurve` owner is cleaner if the factory bytecode can still change, because the deployer key would never hold the role.

Proof of Concept

The shipped deployment script leaves the shared `FeeCurve` owned by the broadcasting key and does not start an ownership transfer:

```
function test_deployScriptLeavesFeeCurveWithDeployerEoa() public {
    address safeRecipient = makeAddr("protocolSafe");
    vm.setEnv("FEE_RECIPIENT", vm.toString(safeRecipient));

    CofferFactory factory = new DeployCofferFactory().run();
    FeeCurve feeCurve = FeeCurve(factory.I_FEE_CURVE_ADDRESS());
    address deployer = tx.origin;

    assertEq(feeCurve.owner(), deployer);
    assertEq(deployer.code.length, 0);
    assertEq(feeCurve.pendingOwner(), address(0));

    vm.deal(address(this), 5 ether);
    feeCurve.collectFee{value: 5 ether}();
    address replacement = makeAddr("replacement");
    vm.prank(deployer);
    feeCurve.setFeeRecipient(replacement);
    feeCurve.claim();

    assertEq(replacement.balance, 5 ether);
    assertEq(safeRecipient.balance, 0);
}
```

Run from `coffer-smart-contracts` at commit 793f6a7:

```
forge build
forge test --match-test test_deployScriptLeavesFeeCurveWithDeployerEoa -vv
```

Observed result:

```
[PASS] test_deployScriptLeavesFeeCurveWithDeployerEoa()
FeeCurve owner after DeployCofferFactory: 0x1804c8AB1F12E6bbf3894d4083f33e07309d1f38
```

Mitigation

Fixed. Mainnet FeeCurve ownership was transferred to the protocol admin Safe and accepted before any fee was pooled or bond was sold. On-chain verification found `owner()` equal to `0xb5B0A1de34EBd56F5b807ec289E4094066b91dB5` and `pendingOwner()` equal to zero. Commit `e8967f7` adds `PROTOCOL_ADMIN` code validation and initiates the two-step handover in the published deployment flow. The client stated that the Safe will move from 1-of-1 to 2-of-3 before the first bond purchase.

[I-04] Consensus withdrawal event reports the requested amount as withdrawn

Description

`ValidatorWithdrawFromConsensus` documents its indexed amount as "the amount withdrawn in wei." `validatorWithdrawFromConsensus` instead emits the caller's requested amount before sending an EIP-7002 request.

A successful execution-layer call only enqueues the request. Consensus processing happens later and may ignore it or clamp it to the amount available above the validator's activation balance. In a reproduced boundary case, the Coffey emitted 50 ETH as withdrawn for a 40 ETH validator while Electra queued an 8 ETH partial withdrawal and the consensus balance had not yet moved.

No contract accounting depends on this event, and the current backend reads consensus balances directly. The event remains misleading to indexers, monitoring, and future integrations that treat it as completed collateral movement.

Recommendations

Rename the event to describe a request, preserve the unit supplied to EIP-7002, and emit it after the predeploy call succeeds:

```
event ValidatorWithdrawalRequested(uint64 indexed requestedAmountGwei);
```

Document that consensus may later ignore or clamp the request. Integrations consuming an already deployed ABI should treat the existing value only as `requestedAmountWei` and derive actual movement from beacon state and the Coffers execution-layer balance.

Proof of Concept

The production event and real EIP-7002 request were composed at mainnet block 22,500,000. A factory-created Coffey requested 50 ETH from a validator modeled with a 40 ETH consensus balance and 0x02 credentials.

The Coffer emitted:

[illegible]

The real predeploy returned this exact EIP-7685 request:

[illegible]

Processing those bytes with the pinned official Electra process withdrawal request produced:

```
product_event_requested_we=50000000000000000000
electra_pending_we=80000000000000000000
balance_at_request_processing_we=40000000000000000000
ELECTRA_MISMATCH_CONFIRMED
```

The event therefore reports the requested 50 ETH as withdrawn even though consensus queues only the available 8 ETH excess and the validator balance has not yet changed.

Mitigation

Fixed. Commit 389600b documents the existing event as a withdrawal request rather than completed movement. Its NatSpec now states that the amount is requested wei, that zero requests a full exit, and that consensus may pay less or drop the request. Integrations are directed to beacon state and the Coffer balance for actual movement. The ABI rename is reserved for a later deployment.