Defense By Thesis

Security Audit Report

BitFi

Bitcoin Lending Protocol

Initial Report // June 07, 2025
Final Report // June 23, 2025

Team Members

soltho // Security Auditor

Table of Contents

About Thesis Defense

Defense is the security auditing arm of Thesis, Inc., the venture studio behind tBTC, Fold, Mezo, Acre, Taho, Etcher, and Embody. At Defense, we fight for the integrity and empowerment of the individual by strengthening the security of emerging technologies to promote a decentralized future and user freedom. Defense is the leading Bitcoin applied cryptography and security auditing firm. Our team of security auditors have carried out hundreds of security audits for decentralized systems across a number of ecosystems including Bitcoin, Ethereum + EVMs, Stacks, Cosmos SDK, NEAR and more. We offer our services within a variety of technologies including smart contracts, bridges, cryptography, node implementations, wallets and browser extensions, and dApps.

Defense will employ the Defense Audit Approach and Audit Process to the in scope service. In the event that certain processes and methodologies are not applicable to the in scope services, we will indicate as such in individual audit or design review SOWs. In addition, Thesis Defense provides clear guidance on successful Security Audit Preparation.

Section 1.0
Scope

Technical Scope

- **Repository:** https://github.com/BitFi-Lending/BitFi-contract
- **Audit Commit:** `69a58df07683684289d7c296320128217eb7371e`
- **Verficiation Commit:** `d3d7d967e53f4335b9de2d9b28caebd965bd17c7`
- Files in Scope:
  - /contracts/*
  - /hardhat.config.ts

Fork Reference

- **Fork Reference Repository:** https://github.com/aave/aave-v3-core.git
- **Reference Repository Commit:** `9a227019623547c6f0c51c25346f5d3eb28007fd`

Documentation

- README.md

Section 2.0
Executive Summary

Schedule

One security auditor conducted this audit from June 5th to June 7th, 2025, for a total of 3 person-days.

Overview

This security audit report outlines our approach and details the findings and outcomes of our security audit of the BitFi smart contracts. BitFi is a Bitcoin-native lending protocol forked from Aave V3, designed to enable users to collateralize Bitcoin and borrow stablecoins in a trustless, non-custodial manner on a BEVM (Bitcoin EVM Layer 2).

Our audit focused primarily on the core lending mechanics, including:

- The implementation of Bitcoin collateralization through native BTC bridges (Mezo, tBTC)
- Interest rate models and liquidation mechanisms adapted for Bitcoin collateral
- Cross-chain asset management and bridge integrations
- Governance and risk parameter management for BTC-backed lending
- ParaSwap integration for liquidity swapping and collateral management

The audit covered the core protocol contracts responsible for these functionalities, with particular attention to potential manipulation of collateral valuations, interest rates, liquidation processes, and cross-chain bridge security. We also examined the Bitcoin-specific adaptations and the security implications of operating on BEVM Layer 2.

Objectives

We identified the following objectives and areas of concern for our security audit:

- **Collateral Management Security**: Ensure proper handling of Bitcoin collateral through bridge integrations and accurate valuation mechanisms.
- **Lending Pool Integrity**: Verify core Aave V3 functionality works correctly for Bitcoin-backed lending scenarios.
- **Liquidation Mechanism Security**: Confirm liquidation processes protect both borrowers and lenders in volatile BTC markets.
- **Cross-Chain Bridge Safety**: Assess security of Bitcoin bridge integrations (Mezo, tBTC) and cross-chain asset management.
- **Interest Rate Model Accuracy**: Validate interest rate calculations and risk parameters for Bitcoin collateral.
- **ParaSwap Integration Security**: Evaluate the security of liquidity swapping and DEX aggregation components.
- **BEVM Compatibility**: Assess deployment compatibility and potential issues with BEVM Layer 2 infrastructure.

Methodology

- We conducted a threat modeling exercise to determine an appropriate threat and trust model for the smart contracts;
- We performed a line-by-line manual code review of security critical functionality, focusing on Bitcoin-specific adaptations and Aave V3 modifications;
- We analyzed the ParaSwap adapter implementations for potential vulnerabilities in swap logic and external calls;
- We tested attack surfaces related to flash loans and reentrancy.

Threat Model

We conducted a threat model that included malicious users, MEV extractors, and cross-chain bridge attackers as the main threat actors. In doing so, we considered the protocol's governance to be sufficiently decentralized and not malicious. We assumed that external bridges (Mezo, tBTC) and ParaSwap contracts function as intended but are not trusted. The main threats we considered were:

- **Collateral Manipulation**: Attacks on Bitcoin valuation and bridge assets;
- **Flash Loan Exploitation**: Complex attacks leveraging Aave's flash loan functionality;
- **Cross-Chain Bridge Attacks**: Exploitation of Bitcoin bridge mechanisms;
- **Liquidation Frontrunning**: MEV attacks on liquidation processes;
- **ParaSwap Integration Exploits**: Attacks through DEX aggregation layer;

Recommendations

1. **Fix signature malleability vulnerabilities** in permit and delegation functions by implementing OpenZeppelin's ECDSA library.
2. **Implement proper flash loan repayment logic** in ParaSwap adapters to prevent transaction failures.
3. **Add comprehensive input validation** for ParaSwap swap operations and external calls.
4. **Replace unsafe ERC20 operations** with SafeERC20 throughout periphery contracts.
5. **Remove unnecessary L2 optimization contracts** that don't apply to BEVM's sidechain architecture.
6. **Implement secure private key management** for deployment and operational security.
7. **Address TODO comments** and incomplete implementations before production deployment.
8. **Implement a comprehensive test suite** that ensures security, correctness, performance, and upgrade safety of the protocol.

Section 3.0
Key Findings Table

Issues	Severity	Status
ISSUE #1 Signature Malleability Vulnerability in Multiple Functions	Low	Fixed
ISSUE #2 Unsafe ERC20 Operations - Missing SafeERC20 Usage	Low	Fixed
ISSUE #3 Private Key Security Risk in Hardhat Configuration	Low	Fixed
ISSUE #4 TODO Comments Found in Production Code	None	Fixed
ISSUE #5 Unnecessary L2 Optimization Contracts for BEVM Deployment	None	Fixed

Section 4.0
Findings

ISSUE #1

Signature Malleability Vulnerability in Multiple Functions

Low Fixed

Description

Multiple functions (`permit` in `BToken` and `delegationWithSig` in `DebtTokenBase`) are vulnerable to signature malleability attacks due to insufficient validation of ECDSA signature components. Both functions use `ecrecover` without validating the `s` value, allowing attackers to create alternative valid signatures for the same message. This vulnerability can lead to replay attacks and break systems that rely on unique signature hashes.

The primary issues are:

1. Missing validation that `s` is in the lower half of the curve order (preventing malleability)
2. Insufficient validation of `v` parameter (should be 27 or 28)
3. No validation that recovered address is not zero

BToken.sol - `permit` function:

```
function permit(
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) external override {
    require(owner != address(0), Errors.ZERO_ADDRESS_NOT_VALID);
    //solium-disable-next-line
    require(block.timestamp <= deadline, Errors.INVALID_EXPIRATION);
    uint256 currentValidNonce = _nonces[owner];
    bytes32 digest = keccak256(
        abi.encodePacked(
            '\x19\x01',
            DOMAIN_SEPARATOR(),
            keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value,
currentValidNonce, deadline))
        )
    );
    require(owner == ecrecover(digest, v, r, s), Errors.INVALID_SIGNATURE);
     nonces[owner] = currentValidNonce + 1;
    _approve(owner, spender, value);
}
```

DebtTokenBase.sol - `delegationWithSig` function:

```
function delegationWithSig(
    address delegator,
    address delegatee,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
```

```
    bytes32 s) external {    require(delegator != address(0),
Errors.ZERO_ADDRESS_NOT_VALID);    //solium-disable-next-line
require(block.timestamp <= deadline, Errors.INVALID_EXPIRATION);    uint256
currentValidNonce = _nonces[delegator];    bytes32 digest = keccak256(
abi.encodePacked(            '\x19\x01',            DOMAIN_SEPARATOR(),
keccak256(               abi.encode(DELEGATION_WITH_SIG_TYPEHASH, delegatee,
value, currentValidNonce, deadline)            )        )    );
require(delegator == ecrecover(digest, v, r, s), Errors.INVALID_SIGNATURE);
_nonces[delegator] = currentValidNonce + 1;    _approveDelegation(delegator,
delegatee, value);}
```

Location

BToken.sol#L170 DebtTokenBase.sol#L43

Recommendation

For the `permit` function in BToken.sol: Replace the current implementation with OpenZeppelin's battle-tested `ERC20Permit` extension, which includes proper signature malleability protection:

```
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Permit.sol";

contract BToken is ERC20Permit {
    constructor() ERC20Permit("BToken") ERC20("BToken". "BTK") {}
    // permit function is automatically included with proper security measures
}
```

For both functions (if custom implementation is required): Use OpenZeppelin's `ECDSA` library which automatically handles signature malleability:

```
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";

// Use OpenZeppelin's ECDSA library which handles malleability protection
address recoveredAddress = ECDSA.recover(digest. v. r. s):
require(recoveredAddress == owner, Errors.INVALID_SIGNATURE);
```

The OpenZeppelin libraries automatically handle:

- Validation that `s` is in the lower half of the curve order
- Proper `v` parameter validation
- Protection against zero address recovery
- Additional edge case protections

ISSUE #2

Unsafe ERC20 Operations - Missing SafeERC20 Usage

Low Fixed

Description

Multiple periphery contracts use unsafe ERC20 operations (`approve` , `transfer` , `transferFrom`) instead of OpenZeppelin's SafeERC20 library. This creates compatibility issues with non-standard ERC20 tokens that don't return boolean values or have other deviations from the standard.

The main risks include:

- **Transaction failures** with tokens like USDT that don't return boolean values
- **Silent failures** where operations appear successful but actually fail
- **Inconsistent behavior** across different token implementations

Common problematic patterns found:

```
// Unsafe - may fail with non-standard tokens
IERC20(token).approve(spender, amount);
IERC20(token).transfer(recipient, amount);
IERC20(token).transferFrom(sender, recipient, amount);
```

Location

- StakedTokenTransferStrategy.sol
- Collector.sol
- CollectorController.sol

Recommendation

Replace all direct ERC20 operations with OpenZeppelin's SafeERC20 library:

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract Example {
    using SafeERC20 for IERC20;

    function safeOperations(address token, address spender, uint256 amount)
external {
        // Safe operations that work with all ERC20 variants
        IERC20(token).safeApprove(spender, amount);
        IERC20(token).safeTransfer(recipient, amount);
        IERC20(token).safeTransferFrom(sender, recipient, amount);
    }
}
```

SafeERC20 handles:

- Tokens that don't return boolean values (e.g., USDT)
- Tokens with different return value behaviors
- Proper error handling and reversion on failures

ISSUE #3

Private Key Security Risk in Hardhat Configuration

Low Fixed

Description

The Hardhat configuration directly reads private keys from environment variables, which poses security risks if the `.env` file is accidentally committed to version control or accessed by unauthorized parties. This is particularly concerning for deployment keys that control critical protocol functions.

Current implementation:

```
networks: {
  mezo: {
    url: "https://jsonrpc-mezo.boar.network",
    accounts: [process.env.PRIVATE_KEY || ""], // Unsafe private key handling
    gas: 5000000,
  },
}
```

Location

hardhat.config.ts#L27

Recommendation

Consider implementing more secure private key management practices:

Hardware Wallet Integration:

- Use hardware wallets (Ledger, Trezor) for mainnet deployments
- Hardhat supports hardware wallet plugins for secure signing

Encrypted Keystores:

- Store private keys in encrypted JSON keystores
- Use password-protected keystore files instead of plain text

Runtime Input:

- Prompt for private keys at deployment time via CLI
- Avoid storing sensitive keys in configuration files

Additional Security Measures:

- Ensure `.env` files are never committed to version control
- Use separate keys for testing vs production environments
- Consider multi-signature wallets for critical protocol operations
- Implement proper key rotation policies

Choose the approach that best fits your deployment workflow and security requirements.

ISSUE #4

TODO Comments Found in Production Code

None Fixed

Description

The codebase contains TODO comments that indicate pending work items or incomplete implementation. These should be reviewed and either implemented or removed before production deployment.

```
 // Get bTokens rewards information
// TODO: check that this is deployed correctly on contract and remove casting
IRewardsController bTokenIncentiveController = IRewardsController(
  address(IncentivizedERC20(baseData.bTokenAddress).getIncentivesController())
);
```

Location

UiIncentiveDataProviderV3.sol#L52

Recommendation

Review the TODO comment and either:

1. **Implement the pending verification** - Verify that incentives controllers are deployed correctly and add proper validation
2. **Remove the TODO** if the casting has been verified as safe
3. **Add proper error handling** around the casting operation to prevent runtime failures

Consider implementing validation before casting:

```
 address controllerAddress =
IncentivizedERC20(baseData.bTokenAddress).getIncentivesController();
require(controllerAddress != address(0), "Invalid incentives controller");
IRewardsController bTokenIncentiveController =
IRewardsController(controllerAddress);
```

ISSUE #5

Unnecessary L2 Optimization Contracts for BEVM Deployment

None Fixed

Description

The codebase includes L2-specific optimization contracts (`L2Pool` , `L2Encoder` , `CalldataLogic`) designed for rollup chains where calldata compression reduces transaction costs. Since BEVM is a Bitcoin Layer 2 sidechain rather than an Ethereum rollup, these optimizations provide no benefit and add unnecessary complexity to the deployment.

L2 optimizations are specifically designed for:

- **Ethereum rollups** (Arbitrum, Optimism, Polygon zkEVM) where calldata is expensive
- **Calldata compression** to reduce L1 posting costs
- **Batch transaction encoding** for rollup efficiency

BEVM characteristics that make L2 contracts unnecessary:

- **Sidechain architecture** rather than rollup
- **Bitcoin-based** gas fees instead of Ethereum L1 calldata costs
- **Different cost structure** where calldata optimization is irrelevant

Location

- `L2Pool.sol` - Calldata-optimized pool contract
- `L2Encoder.sol` - Transaction encoding for calldata compression
- `CalldataLogic.sol` - Calldata optimization logic

Recommendation

Remove unnecessary L2 contracts from the BEVM deployment to:

1. **Reduce deployment complexity** and potential attack surface
2. **Eliminate unused code** that adds no value on BEVM
3. **Simplify maintenance** by removing rollup-specific optimizations
4. **Use standard Pool contract** which is sufficient for sidechain deployments

This change aligns the deployment with BEVM's sidechain architecture and removes Ethereum rollup-specific optimizations that don't apply to Bitcoin Layer 2 networks.

Section 5.0
Appendix A

Severity Rating Definitions

At Thesis Defense, we utilize the Immunefi Vulnerability Severity Classification System - v2.3.

Severity	Definition
Critical	• Manipulation of governance voting result deviating from voted outcome and resulting in a direct change from intended effect of original results • Direct theft of any user funds, whether at-rest or in-motion, other than unclaimed yield • Direct theft of any user NFTs, whether at-rest or in-motion, other than unclaimed royalties • Permanent freezing of funds • Permanent freezing of NFTs • Unauthorized minting of NFTs • Predictable or manipulable RNG that results in abuse of the principal or NFT • Unintended alteration of what the NFT represents (e.g. token URI, payload, artistic content) • Protocol insolvency
High	• Theft of unclaimed yield • Theft of unclaimed royalties • Permanent freezing of unclaimed yield • Permanent freezing of unclaimed royalties • Temporary freezing of funds • Temporary freezing NFTs
Medium	• Smart contract unable to operate due to lack of token funds • Enabling/disabling notifications • Griefing (e.g. no profit motive for an attacker, but damage to the users or the protocol) • Theft of gas • Unbounded gas consumption
Low	• Contract fails to deliver promised returns, but doesn't lose value
None	• We make note of issues of no severity that reflect best practice recommendations or opportunities for optimization, including, but not limited to, gas optimization, the divergence from standard coding practices, code readability issues, the incorrect use of dependencies, insufficient test coverage, or the absence of documentation or code comments.

Section 6.0
Appendix B

Thesis Defense Disclaimer

Thesis Defense conducts its security audits and other services provided based on agreed-upon and specific scopes of work (SOWs) with our Customers. The analysis provided in our reports is based solely on the information available and the state of the systems at the time of review. While Thesis Defense strives to provide thorough and accurate analysis, our reports do not constitute a guarantee of the project's security and should not be interpreted as assurances of error-free or risk-free project operations. It is imperative to acknowledge that all technological evaluations are inherently subject to risks and uncertainties due to the emergent nature of cryptographic technologies.

Our reports are not intended to be utilized as financial, investment, legal, tax, or regulatory advice, nor should they be perceived as an endorsement of any particular technology or project. No third party should rely on these reports for the purpose of making investment decisions or consider them as a guarantee of project security.

Links to external websites and references to third-party information within our reports are provided solely for the user's convenience. Thesis Defense does not control, endorse, or assume responsibility for the content or privacy practices of any linked external sites. Users should exercise caution and independently verify any information obtained from third-party sources.

The contents of our reports, including methodologies, data analysis, and conclusions, are the proprietary intellectual property of Thesis Defense and are provided exclusively for the specified use of our Customers. Unauthorized disclosure, reproduction, or distribution of this material is strictly prohibited unless explicitly authorized by Thesis Defense. Thesis Defense does not assume any obligation to update the information contained within our reports post-publication, nor do we owe a duty to any third party by virtue of making these analyses available.