FAIRYPROOF

Investment Vault

AUDIT REPORT

Version 1.0.0

Serial No. 20251221000012018

Presented by Fairyproof

December 21, 2025

01. Introduction

This document includes the results of the audit performed by the Fairyproof team on the Singularry InvestmentVault project.

Audit Start Time:

December 19, 2025

Audit End Time:

December 21, 2025

Audited Code's Github Repository:

https://github.com/singularry/sly-investment-vault/

Audited Code's Github Commit Number When Audit Started:

b1ff722129ab4eb4de2ccae49ba77abd4ba06bbe

Audited Code's Github Commit Number When Audit Ended:

ed6fd40aacce1367c10f01822b04c10e37692ce7

The goal of this audit is to review Singularry's solidity implementation for its InvestmentVault function, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

We make observations on specific areas of the code that present concrete problems, as well as general observations that traverse the entire codebase horizontally, which could improve its quality as a whole.

This audit only applies to the specified code, software or any materials supplied by the Singularry team for specified versions. Whenever the code, software, materials, settings, environment etc is changed, the comments of this audit will no longer apply.

— Disclaimer

Note that as of the date of publishing, the contents of this report reflect the current understanding of known security patterns and state of the art regarding system security. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk.

The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. If the audited source files are smart contract files, risks or issues introduced by using data feeds from offchain sources are not extended by this review either.

Given the size of the project, the findings detailed here are not to be considered exhaustive, and further testing and audit is recommended after the issues covered are fixed.

To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

— Methodology

The above files' code was studied in detail in order to acquire a clear impression of how the its specifications were implemented. The codebase was then subject to deep analysis and scrutiny, resulting in a series of observations. The problems and their potential solutions are discussed in this document and, whenever possible, we identify common sources for such problems and comment on them as well.

The Fairyproof auditing process follows a routine series of steps:

1. Code Review, Including:

- Project Diagnosis

Understanding the size, scope and functionality of your project's source code based on the specifications, sources, and instructions provided to Fairyproof.

- Manual Code Review

Reading your source code line-by-line to identify potential vulnerabilities.

- Specification Comparison

Determining whether your project's code successfully and efficiently accomplishes or executes its functions according to the specifications, sources, and instructions provided to Fairyproof.

2. Testing and Automated Analysis, Including:

- Test Coverage Analysis

Determining whether the test cases cover your code and how much of your code is exercised or executed when test cases are run.

- Symbolic Execution

Analyzing a program to determine the specific input that causes different parts of a program to execute its functions.

3. Best Practices Review

Reviewing the source code to improve maintainability, security, and control based on the latest established industry and academic practices, recommendations, and research.

— Structure of the document

This report contains a list of issues and comments on all the above source files. Each issue is assigned a severity level based on the potential impact of the issue and recommendations to fix it, if applicable. For ease of navigation, an index by topic and another by severity are both provided at the beginning of the report.

— Documentation

For this audit, we used the following source(s) of truth about how the token issuance function should work:

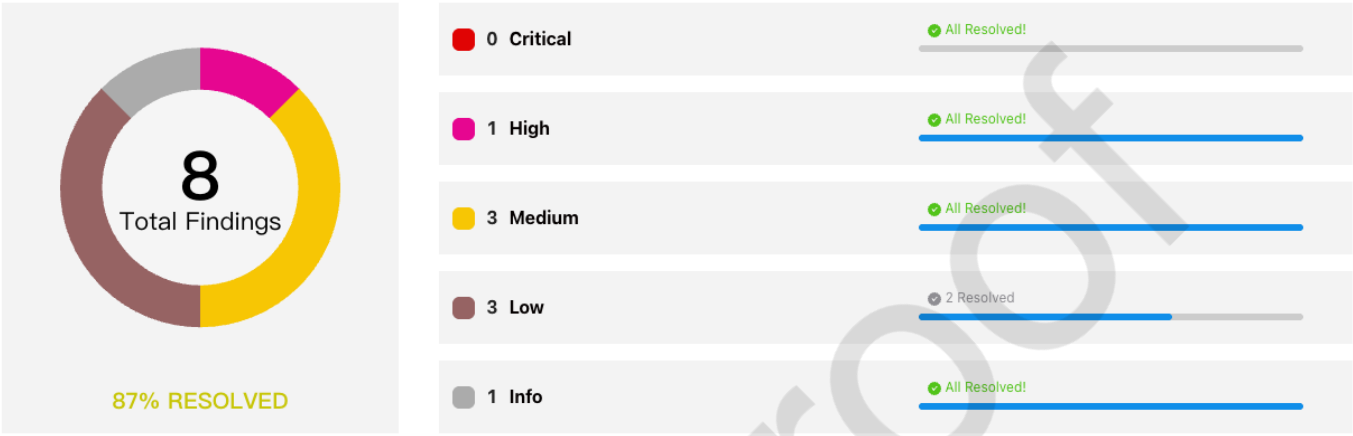
Website:https://singularry.org/

Source Code: https://github.com/singularry/sly-investment-vault/

These were considered the specification, and when discrepancies arose with the actual code behavior, we consulted with the Singularry team or reported an issue.

— Comments from Auditor

Serial Number	Auditor	Audit Time	Result
2025122100012018	Fairyproof Security Team	Dec 19, 2025 - Dec 21, 2025	Low Risk



Summary:

The Fairyproof security team used its auto analysis tools and manual work to audit the project. During the audit, one issue of high-severity, three issues of medium-severity, three issues of low-severity and one issue of info-severity were uncovered. The Singularry team fixed one issue of high, three issues of medium, two issues of low and one issue of info, and acknowledged the remaining issues.

02. About Fairyproof

Fairyproof is a leading technology firm in the blockchain industry, providing consulting and security audits for organizations. Fairyproof has developed industry security standards for designing and deploying blockchain applications.

03. Introduction to Singularry

Singularry is a platform of creating next-generation financial systems with intelligent, autonomous protocols and seamless cross-chain operations.

The above description is quoted from relevant documents of Singularry.

04. Major functions of audited code

The InvestmentVault contract implements a token locking and investment protocol with periodic yield distribution. Users lock tokens as collateral to gain eligibility for investment during defined periods. Each period has configurable parameters such as maximum total locked tokens, investment window, per-wallet caps, and yield distribution rules. The contract enforces state transitions for each period (INIT, INVEST, INVESTMENT_ONGOING, WITHDRAWN, YIELD, FINALISED, CANCELLED), manages user collateral and investments, and allows the owner to withdraw invested funds, deposit yield, and handle emergency cancellations. All user and admin actions are subject to strict checks to ensure fairness, cap enforcement, and secure fund management.

Note:

This contract is designed for scenarios where user investment principal is withdrawn from the contract for off-chain or external investment activities to generate yield, rather than being continuously held within the contract.

05. Coverage of issues

The issues that the Fairyproof team covered when conducting the audit include but are not limited to the following ones:

- Access Control
- Admin Rights
- Arithmetic Precision
- Code Improvement
- Contract Upgrade/Migration
- Delete Trap
- Design Vulnerability
- DoS Attack
- EOA Call Trap
- Fake Deposit
- Function Visibility
- Gas Consumption
- Implementation Vulnerability
- Inappropriate Callback Function
- Injection Attack
- Integer Overflow/Underflow
- IsContract Trap
- Miner's Advantage
- Misc
- Price Manipulation
- Proxy selector clashing
- Pseudo Random Number
- Re-entrancy Attack
- Replay Attack

- Rollback Attack
- Shadow Variable
- Slot Conflict
- Token Issuance
- Tx.origin Authentication
- Uninitialized Storage Pointer

06. Severity level reference

Every issue in this report was assigned a severity level from the following:

Critical severity issues need to be fixed as soon as possible.

High severity issues will probably bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Informational is not an issue or risk but a suggestion for code improvement.

07. Major areas that need attention

Based on the provided source code the Fairyproof team focused on the possible issues and risks related to the following functions or areas.

- Function Implementation

We checked whether or not the functions were correctly implemented.
We found some issues, for more details please refer to [FP-3,FP-4,FP-5,FP-7,FP-8] in "09. Issue description".

- Access Control

We checked each of the functions that could modify a state, especially those functions that could only be accessed by owner or administrator
We didn't find issues or risks in these functions or areas at the time of writing.

- Token Issuance & Transfer

We examined token issuance and transfers for situations that could harm the interests of holders.
We didn't find issues or risks in these functions or areas at the time of writing.

- State Update

We checked some key state variables which should only be set at initialization.
We found one issue, for more details please refer to [FP-2] in "09. Issue description".

- Asset Security

We checked whether or not all the functions that transfer assets were safely handled.
We didn't find issues or risks in these functions or areas at the time of writing.

- Miscellaneous

We checked the code for optimization and robustness.
We found some issues, for more details please refer to [FP-1,FP-6] in "09. Issue description".

08. List of issues by severity

Index	Title	Issue/Risk	Severity	Status
FP-1	Trust Assumptions	Admin Rights	Low	Acknowledged
FP-2	Multiple Withdrawals of Investments	Design Vulnerability	High	✓ Fixed
FP-3	No Emergency Withdrawal or Period Termination Mechanism	Design Vulnerability	Medium	✓ Fixed
FP-4	Investment Window Fairness Issue	Design Vulnerability	Medium	✓ Fixed
FP-5	No Minimum Yield Deposit Validation	Design Vulnerability	Medium	✓ Fixed
FP-6	Period State Machine Enforcement Weakness	Design Vulnerability	Low	✓ Fixed
FP-7	Inconsistent Cap Parameters May Cause User Confusion	Design Vulnerability	Low	✓ Fixed
FP-8	State Management Consistency	Design Vulnerability	Info	✓ Fixed

09. Issue descriptions

[FP-1] Trust Assumptions

Admin Rights Low Acknowledged

Issue/Risk: Admin Rights

Description:

Users must trust the admin for all critical operations.

- Admin must return invested funds (no guaranteed minimum yield)
- Admin must complete the period lifecycle in reasonable time
- Admin must not pause the contract indefinitely
- Admin must set fair fee percentages
- Admin must use legitimate ERC20 tokens for investments

Recommendation:

Disclose trust assumptions clearly in documentation and UI. Consider governance or multi-signature admin for risk mitigation.

Update:

Risk acknowledged. Disclosure recommended.

Status:

The project team is already aware of this and plans to explain it in detail in the documentation.

[FP-2] Multiple Withdrawals of Investments

Design Vulnerability High ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

The admin can call withdrawInvestments multiple times for the same period, extracting the invested funds repeatedly as long as the contract balance allows. This is a critical business logic vulnerability.
After the first withdrawal, the period state remains INVESTMENT_ONGOING, allowing repeated withdrawals. This can result in loss of user funds and protocol insolvency.

This risk is especially severe if `investToken` and `lockToken` are the same, as repeated withdrawals can quickly drain the contract's balance and cause accounting inconsistencies.
Relevant code:

```
function withdrawInvestments(uint256 periodId) external onlyOwner nonReentrant {
    // ...existing code...
    if (currentState != PeriodState.INVESTMENT_ONGOING) revert InvalidState();
    // ...existing code...
    IERC20(period.investToken).safeTransfer(msg.sender, amount);
    // ...existing code...
}
```

Recommendation:

Introduce a new state (e.g., WITHDRAWN) in `PeriodState` and set it after the first withdrawal. Subsequent calls should revert if the state is not `INVESTMENT_ONGOING`. This ensures only one withdrawal per period and prevents repeated fund extraction.

Update:

The project team add `period.state = PeriodState.WITHDRAWN;` in withdrawInvestments. So the period state is set to `WITHDRAWN`, and any further calls to withdrawInvestments for the same period will revert.
This resolves the vulnerability and ensures correct state transitions regardless of token configuration.

Status:

Fixed at commit 12aecac574961ffc3894d1ee1be14384b7ee86b8.

[FP-3] No Emergency Withdrawal or Period Termination Mechanism

Design Vulnerability Medium ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

There is no emergency withdrawal or forced period termination mechanism. Users are fully dependent on the admin to finalize periods and return funds.
If the admin sets an excessively long period or fails to complete the period lifecycle, user funds may be locked indefinitely.
Relevant code:

```
function createPeriod(uint256 maxTotalLocked, uint256 periodLength, ...) external onlyOwner returns
(uint256 periodId) {
    // ...existing code...
    // No upper bound check for periodLength
}
```

Recommendation:

Add reasonable bounds for periodLength and investLength. Consider adding an emergency admin or user-triggered period termination function.

Update:

The project team added the `cancelPeriod` function to cancel investments in specific scenarios, allowing users to withdraw their funds.

Status:

Fixed at commit 12aecac574961ffc3894d1ee1be14384b7ee86b8.

[FP-4] Investment Window Fairness Issue

Design Vulnerability Medium ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

Users can invest at the last moment of the investment window, resulting in minimal lock time, while early investors are locked for longer.
This mechanism allows “tail-end” investors to avoid long lock periods, which is unfair to early participants and may encourage MEV/arbitrage behavior.
Relevant code:

```
function invest(uint256 periodId, uint256 amount) external nonReentrant whenNotPaused {
    // ...existing code...
    if (block.timestamp > period.investEndTime) revert InvestWindowClosed();
    // ...existing code...
}
```

Recommendation:

Disclose this risk in documentation. Consider mechanisms to equalize lock duration if business requires fairness.

Update:

The project team has addressed the "Investment Window Fairness" issue by introducing a `maxTotalLocked` cap for each period. This ensures that once the cap is reached, no further investments can be made, preventing latecomers from exploiting the system by locking tokens only at the end of the window. While early users may still need to lock their tokens slightly earlier than strictly necessary, the team clarified that no funds are actually invested until the investment window closes, and in practice, the investment window will be kept very short. This minimizes the impact of early locking and ensures fairer participation for all users. The risk of unfair advantage due to timing is now considered low and mitigated by both the cap and operational policy.

Status:

Fixed at commit 12aecac574961ffc3894d1ee1be14384b7ee86b8.

[FP-5] No Minimum Yield Deposit Validation

Design Vulnerability Medium ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

Admin can deposit less yield than the total invested principal, resulting in user losses.
There is no check to ensure the deposited yield covers the total principal. This allows the admin to return less than the invested amount, causing user losses.
Relevant code:

```
function depositYield(uint256 periodId, uint256 amount) external onlyOwner nonReentrant {
    // ...existing code...
    // No minimum principal check
    IERC20(period.investToken).safeTransferFrom(msg.sender, address(this), amount);
    // ...existing code...
}
```

Recommendation:

Add a check: require(amount >= period.totalInvested, "Yield must cover principal");
If business allows losses, disclose this risk clearly to users.

Update:

The project team added a amount validation in `depositYield`:

```
// Ensure yield covers at least the principal (user protection)
if (amount < period.totalInvested) revert InsufficientYieldDeposit();
```

Status:

Fixed at commit 12aecac574961ffc3894d1ee1be14384b7ee86b8.

[FP-6] Period State Machine Enforcement Weakness

Design Vulnerability Low ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

Insufficient enforcement of period state transitions allows certain admin operations (e.g., depositYield) to be called out of order, potentially breaking the intended lifecycle.

The contract should strictly require that `depositYield` can only be called after `withdrawInvestments` (i.e., when the period is in `INVESTMENT_ONGOING` state). However, the current implementation may allow `depositYield` to be called prematurely, if only relying on auto state transitions via `getPeriodState`. This can lead to inconsistent period states and unexpected behavior.
Relevant code:

```
function depositYield(uint256 periodId, uint256 amount) external onlyOwner nonReentrant {
    // ...existing code...
    // Should strictly require: period.state == PeriodState.INVESTMENT_ONGOING
    // ...existing code...
}
```

Recommendation:

Introduce a dedicated state (e.g., WITHDRAWN) in the `PeriodState` enum.
After a successful withdrawInvestments, set the period state to WITHDRAWN.
Require that `depositYield` can only be called when the period is in the `WITHDRAWN` state, not merely `INVESTMENT_ONGOING`.
This ensures strict separation of the withdrawal and yield deposit phases, prevents multiple withdrawals or out-of-order operations, and enforces a robust state machine for the period lifecycle.

Update:

Project team introduces a new period state (`WITHDRAWN`) and while calling withdrawInvestments, set the period state to `WITHDRAWN`

Status:

Fixed at commit 12aecac574961ffc3894d1ee1be14384b7ee86b8.

[FP-7] Inconsistent Cap Parameters May Cause User Confusion

Design Vulnerability Low ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

If both maxLockPerWallet and maxInvestPerWallet are set, but the lock cap is insufficient to support the maximum investment (i.e., maxLockPerWallet < maxInvestPerWallet * 1e18 / fraction), users may be unable to reach the maximum allowed investment even if they reach their lock cap. This could lead to confusion or failed investment attempts.

Recommendation:

Add a parameter consistency check in `reatePeriod` to ensure that, when both caps are set, `maxLockPerWallet` is always sufficient to support `maxInvestPerWallet` according to the `fraction`.

Update:

The project team added a cap consistency validation in createPeriod:

```
if (maxLockPerWallet > 0 && maxInvestPerWallet > 0) {
    uint256 requiredLockForMaxInvest = (maxInvestPerWallet * FRACTION_SCALE) / fraction;
    if (maxLockPerWallet < requiredLockForMaxInvest) revert InconsistentCapParameters();
}
```

Status:

Fixed at commit ed6fd40aacce1367c10f01822b04c10e37692ce7.

[FP-8] State Management Consistency

Design Vulnerability Info ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

State transitions and updates are sometimes performed after reading state, rather than in a unified, atomic manner.
For best practice, state changes should be managed and updated in a single step to avoid inconsistencies and potential race conditions.

Recommendation:

Refactor state management to ensure atomic updates and transitions, especially for period state changes.

Update:

The project team did not use a unified function approach, but they have handled state transitions correctly.

Status:

Fixed at commit 12aecac574961ffc3894d1ee1be14384b7ee86b8.

10. Recommendations to enhance the overall security

We list some recommendations in this section. They are not mandatory but will enhance the overall security of the system if they are adopted.

- Consider managing the owner's access control with great care and transfering it to a multi-sig wallet or DAO when necessary.

11. Appendices

11.1 Unit Test

1. InvestmentVault.t.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import {InvestmentVault} from "../src/InvestmentVault.sol";
import {IInvestmentVault} from "../src/interfaces/IInvestmentVault.sol";
```

```
import {Test} from "forge-std/Test.sol";
import {IERC20} from "@openzeppelin-contracts/token/ERC20/IERC20.sol";
import {ERC20} from "@openzeppelin-contracts/token/ERC20/ERC20.sol";

contract MockERC20 is ERC20 {
    constructor(string memory name, string memory symbol) ERC20(name, symbol) {}

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }
}

contract InvestmentVaultTest is Test {
    InvestmentVault vault;
    MockERC20 lockToken;
    MockERC20 investToken;

    address owner = address(1);
    address user1 = address(2);
    address user2 = address(3);

    uint256 constant FRACTION_SCALE = 1e18;

    function setUp() public {
        lockToken = new MockERC20("Lock Token", "LOCK");
        investToken = new MockERC20("USDT", "USDT");

        vm.prank(owner);
        vault = new InvestmentVault(address(lockToken), owner);

        // Mint tokens to users
        lockToken.mint(user1, 10000 * 1e18);
        lockToken.mint(user2, 10000 * 1e18);
        investToken.mint(user1, 10000 * 1e18);
        investToken.mint(user2, 10000 * 1e18);
        investToken.mint(owner, 100000 * 1e18);

        // Approve vault
        vm.prank(user1);
        lockToken.approve(address(vault), type(uint256).max);
        vm.prank(user1);
        investToken.approve(address(vault), type(uint256).max);
        vm.prank(user2);
        lockToken.approve(address(vault), type(uint256).max);
        vm.prank(user2);
        investToken.approve(address(vault), type(uint256).max);
        vm.prank(owner);
        investToken.approve(address(vault), type(uint256).max);
    }

    // ============ Custom Supplementary Tests ============

    function test_Invest_LastMoment_UnlockAllAfterInvestWindow() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500, 0, 0);
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(1000 * 1e18);
```

```
        vm.warp(block.timestamp + 7 days - 1);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);


        vm.warp(block.timestamp + 2);
        vm.prank(user1);
        vault.unlock(1000 * 1e18);
        assertEq(vault.getUserLocked(user1), 0);
    }


    function test_DepositYield_RevertIfLessThanPrincipal() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 0, 0, 0);
        vm.prank(owner);
        vault.startPeriod(0);
        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);
        vm.warp(block.timestamp + 8 days);
        vm.prank(owner);
        vault.withdrawInvestments(0);
        vm.prank(owner);
        vm.expectRevert(IInvestmentVault.InsufficientYieldDeposit.selector);
        vault.depositYield(0, 80 * 1e18);
    }

    function test_WithdrawYield_RevertIfNoDepositYield() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 0, 0, 0);
        vm.prank(owner);
        vault.startPeriod(0);
        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);
        vm.warp(block.timestamp + 8 days);
        vm.prank(owner);
        vault.withdrawInvestments(0);
        vm.prank(user1);
        vm.expectRevert(IInvestmentVault.InvalidState.selector);
        vault.withdrawYield(0);
    }

    function test_MultiUser_ImbalancedInvestments() public {
        vm.prank(owner);
        vault.createPeriod(3000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 0, 0, 0);
        vm.prank(owner);
        vault.startPeriod(0);
        vm.prank(user1);
        vault.lock(3000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 299 * 1e18);
        vm.prank(user2);
        vault.lock(10 * 1e18);
        vm.prank(user2);
        vault.invest(0, 1 * 1e18);
        vm.warp(block.timestamp + 8 days);
        vm.prank(owner);
```

```
        vault.withdrawInvestments(0);
        vm.prank(owner);
        vault.depositYield(0, 300 * 1e18);
        (uint256 gross1,,) = vault.calculateUserYield(user1, 0);
        (uint256 gross2,,) = vault.calculateUserYield(user2, 0);
        assertEq(gross1, 299 * 1e18);
        assertEq(gross2, 1 * 1e18);
    }

    function test_Invest_RevertIfCollateralJustEnough() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 0, 0, 0);
        vm.prank(owner);
        vault.startPeriod(0);
        vm.prank(user1);
        vault.lock(99 * 1e18);
        vm.prank(user1);
        vm.expectRevert(IInvestmentVault.InsufficientCollateral.selector);
        vault.invest(0, 100 * 1e18);
    }

    function test_Unlock_RevertIfCollateralRequired() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 0, 0, 0);
        vm.prank(owner);
        vault.startPeriod(0);
        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);
        vm.prank(user1);
        vm.expectRevert(IInvestmentVault.InsufficientCollateral.selector);
        vault.unlock(1000 * 1e18);
    }

    function test_DepositYield_RevertIfNotWithdrawn() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 0, 0, 0);
        vm.prank(owner);
        vault.startPeriod(0);
        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);
        vm.warp(block.timestamp + 8 days);
        vm.prank(owner);
        vm.expectRevert(IInvestmentVault.InvalidState.selector);
        vault.depositYield(0, 100 * 1e18);
    }

    function test_MEV_TemporaryCollateral() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 0, 0, 0);
        vm.prank(owner);
        vault.startPeriod(0);
        vm.warp(block.timestamp + 7 days - 1);
        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);
        vm.warp(block.timestamp + 2);
```

```
        vm.prank(user1);
        vault.unlock(1000 * 1e18);
        assertEq(vault.getUserLocked(user1), 0);
    }

    function test_DepositYield_RevertIfZeroAmount() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 0, 0, 0);
        vm.prank(owner);
        vault.startPeriod(0);
        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);
        vm.warp(block.timestamp + 8 days);
        vm.prank(owner);
        vault.withdrawInvestments(0);
        vm.prank(owner);
        vm.expectRevert(IInvestmentVault.InvalidAmount.selector);
        vault.depositYield(0, 0);
    }

    function test_WithdrawYield_RevertTwice() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 0, 0, 0);
        vm.prank(owner);
        vault.startPeriod(0);
        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);
        vm.warp(block.timestamp + 8 days);
        vm.prank(owner);
        vault.withdrawInvestments(0);
        vm.prank(owner);
        vault.depositYield(0, 100 * 1e18);
        vm.prank(user1);
        vault.withdrawYield(0);
        vm.prank(user1);
        vm.expectRevert(IInvestmentVault.AlreadyClaimed.selector);
        vault.withdrawYield(0);
    }

    function test_Pause_AllUserOpsBlocked() public {
        vm.prank(owner);
        vault.pause();
        vm.prank(user1);
        vm.expectRevert();
        vault.lock(100 * 1e18);
        vm.prank(user1);
        vm.expectRevert();
        vault.unlock(100 * 1e18);
        vm.prank(user1);
        vm.expectRevert();
        vault.invest(0, 10 * 1e18);
        vm.prank(user1);
        vm.expectRevert();
        vault.withdrawYield(0);
    }

    // ============ Constructor Tests ============
```

```
    function test_Constructor() public view {
        assertEq(address(vault.lockToken()), address(lockToken));
        assertEq(vault.owner(), owner);
        assertEq(vault.periodCount(), 0);
    }

    function test_Constructor_RevertZeroAddress() public {
        vm.expectRevert(IInvestmentVault.InvalidAddress.selector);
        new InvestmentVault(address(0), owner);
    }

    // ============ Lock/Unlock Tests ============

    function test_Lock() public {
        uint256 amount = 1000 * 1e18;

        vm.prank(user1);
        vault.lock(amount);

        assertEq(vault.getUserLocked(user1), amount);
        assertEq(vault.totalLockedTokens(), amount);
        assertEq(lockToken.balanceOf(address(vault)), amount);
    }

    function test_Lock_RevertZeroAmount() public {
        vm.prank(user1);
        vm.expectRevert(IInvestmentVault.InvalidAmount.selector);
        vault.lock(0);
    }

    function test_Unlock() public {
        uint256 lockAmount = 1000 * 1e18;
        uint256 unlockAmount = 400 * 1e18;

        vm.prank(user1);
        vault.lock(lockAmount);

        vm.prank(user1);
        vault.unlock(unlockAmount);

        assertEq(vault.getUserLocked(user1), lockAmount - unlockAmount);
        assertEq(vault.totalLockedTokens(), lockAmount - unlockAmount);
    }

    function test_Unlock_RevertInsufficientBalance() public {
        vm.prank(user1);
        vault.lock(100 * 1e18);

        vm.prank(user1);
        vm.expectRevert(IInvestmentVault.InsufficientBalance.selector);
        vault.unlock(200 * 1e18);
    }

    // ============ Period Creation Tests ============

    function test_CreatePeriod() public {
        vm.prank(owner);
        uint256 periodId = vault.createPeriod(
            1000 * 1e18,    // maxTotalLocked
            30 days,        // periodLength
```

```
            7 days,          // investLength
            1e17,            // fraction (0.1 in 1e18 scale)
            address(investToken),
            500,             // 5% fee
            2000 * 1e18,     // maxLockPerWallet
            300 * 1e18       // maxInvestPerWallet
        );

        assertEq(periodId, 0);
        assertEq(vault.periodCount(), 1);

        InvestmentVault.Period memory period = vault.getPeriod(0);
        assertEq(period.maxTotalLocked, 1000 * 1e18);
        assertEq(period.periodLength, 30 days);
        assertEq(period.investLength, 7 days);
        assertEq(period.fraction, 1e17);
        assertEq(period.investToken, address(investToken));
        assertEq(period.feePercentage, 500);
        assertEq(period.maxLockPerWallet, 2000 * 1e18);
        assertEq(period.maxInvestPerWallet, 300 * 1e18);
        assertEq(uint(period.state), uint(IInvestmentVault.PeriodState.INIT));
    }

    function test_CreatePeriod_RevertInvalidParameters() public {
        // Zero investLength
        vm.prank(owner);
        vm.expectRevert(IInvestmentVault.InvalidParameters.selector);
        vault.createPeriod(1000 * 1e18, 30 days, 0, 1e17, address(investToken), 500, 0, 0);

        // periodLength < investLength
        vm.prank(owner);
        vm.expectRevert(IInvestmentVault.InvalidParameters.selector);
        vault.createPeriod(1000 * 1e18, 5 days, 7 days, 1e17, address(investToken), 500, 0, 0);

        // Zero fraction
        vm.prank(owner);
        vm.expectRevert(IInvestmentVault.InvalidParameters.selector);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 0, address(investToken), 500, 0, 0);

        // Fee > 100%
        vm.prank(owner);
        vm.expectRevert(IInvestmentVault.InvalidParameters.selector);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 10001, 0, 0);
    }

    function test_CreatePeriod_RevertNonOwner() public {
        vm.prank(user1);
        vm.expectRevert();
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500, 0, 0);
    }

    // ============ Period Start Tests ============

    function test_StartPeriod() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);

        vm.prank(owner);
        vault.startPeriod(0);

        assertEq(uint(vault.getPeriodState(0)), uint(IInvestmentVault.PeriodState.INVEST));
```

```
        InvestmentVault.Period memory period = vault.getPeriod(0);
        assertEq(period.startTime, block.timestamp);
        assertEq(period.investEndTime, block.timestamp + 7 days);
        assertEq(period.periodEndTime, block.timestamp + 30 days);
    }

    function test_StartPeriod_RevertNotInit() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);

        vm.prank(owner);
        vault.startPeriod(0);

        // Try to start again
        vm.prank(owner);
        vm.expectRevert(IInvestmentVault.InvalidState.selector);
        vault.startPeriod(0);
    }

    // ============ Investment Tests ============

    function test_Invest() public {
        // Setup: Create and start period
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        // User locks tokens
        vm.prank(user1);
        vault.lock(1000 * 1e18);

        // User invests (fraction = 0.1, so 1000 locked allows 100 invest)
        uint256 investAmount = 100 * 1e18;
        vm.prank(user1);
        vault.invest(0, investAmount);

        InvestmentVault.UserInvestment memory userInv = vault.getUserInvestment(user1, 0);
        assertEq(userInv.amountInvested, investAmount);

        InvestmentVault.Period memory period = vault.getPeriod(0);
        assertEq(period.totalInvested, investAmount);
    }

    function test_Invest_Multiple() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(1000 * 1e18);

        // First investment
        vm.prank(user1);
        vault.invest(0, 50 * 1e18);

        // Second investment
        vm.prank(user1);
        vault.invest(0, 30 * 1e18);
```

```
        InvestmentVault.UserInvestment memory userInv = vault.getUserInvestment(user1, 0);
        assertEq(userInv.amountInvested, 80 * 1e18);
    }

    function test_Invest_RevertInsufficientCollateral() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(100 * 1e18);

        // Try to invest 20 (requires 200 locked at fraction 0.1)
        vm.prank(user1);
        vm.expectRevert(IInvestmentVault.InsufficientCollateral.selector);
        vault.invest(0, 20 * 1e18);
    }

    function test_Invest_RevertAfterInvestWindow() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(1000 * 1e18);

        // Warp past invest window - state auto-transitions to INVESTMENT_ONGOING
        vm.warp(block.timestamp + 8 days);

        vm.prank(user1);
        vm.expectRevert(IInvestmentVault.InvalidState.selector);
        vault.invest(0, 50 * 1e18);
    }

    // ============ Unlock with Active Investment Tests ============

    function test_Unlock_RevertWithActiveInvestment() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(1000 * 1e18);

        // Invest 50 (requires 500 locked)
        vm.prank(user1);
        vault.invest(0, 50 * 1e18);

        // Try to unlock more than allowed
        vm.prank(user1);
        vm.expectRevert(IInvestmentVault.InsufficientCollateral.selector);
        vault.unlock(600 * 1e18);
    }

    function test_Unlock_PartialWithActiveInvestment() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
```

```
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(1000 * 1e18);

        // Invest 50 (requires 500 locked)
        vm.prank(user1);
        vault.invest(0, 50 * 1e18);

        // Can unlock up to 500
        vm.prank(user1);
        vault.unlock(499 * 1e18);

        assertEq(vault.getUserLocked(user1), 501 * 1e18);
    }

    // ============ Auto State Transition Tests ============

    function test_GetPeriodState_AutoTransition() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        assertEq(uint(vault.getPeriodState(0)), uint(IInvestmentVault.PeriodState.INVEST));

        // Warp past invest window
        vm.warp(block.timestamp + 8 days);

        // State should auto-transition
        assertEq(uint(vault.getPeriodState(0)), uint(IInvestmentVault.PeriodState.INVESTMENT_ONGOING));
    }

    // ============ Admin Withdraw/Deposit Tests ============

    function test_WithdrawInvestments() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(1000 * 1e18);

        vm.prank(user1);
        vault.invest(0, 100 * 1e18);

        // Warp past invest window
        vm.warp(block.timestamp + 8 days);

        uint256 ownerBalBefore = investToken.balanceOf(owner);

        vm.prank(owner);
        vault.withdrawInvestments(0);

        assertEq(investToken.balanceOf(owner), ownerBalBefore + 100 * 1e18);
    }

    function test_DepositYield() public {
        vm.prank(owner);
```

```
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(1000 * 1e18);

        vm.prank(user1);
        vault.invest(0, 100 * 1e18);

        // Warp past invest window
        vm.warp(block.timestamp + 8 days);

        vm.prank(owner);
        vault.withdrawInvestments(0);

        // Deposit yield (principal + 10% profit)
        vm.prank(owner);
        vault.depositYield(0, 110 * 1e18);

        assertEq(uint(vault.getPeriodState(0)), uint(IInvestmentVault.PeriodState.YIELD));

        InvestmentVault.Period memory period = vault.getPeriod(0);
        assertEq(period.yieldDeposited, 110 * 1e18);
    }

    // ============ Yield Withdrawal Tests ============

    function test_WithdrawYield() public {
        // Full lifecycle test
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0); // 5% fee
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(1000 * 1e18);

        vm.prank(user1);
        vault.invest(0, 100 * 1e18);

        vm.warp(block.timestamp + 8 days);

        vm.prank(owner);
        vault.withdrawInvestments(0);

        // Deposit yield (principal + 10% profit = 110)
        vm.prank(owner);
        vault.depositYield(0, 110 * 1e18);

        uint256 userBalBefore = investToken.balanceOf(user1);

        vm.prank(user1);
        vault.withdrawYield(0);

        // Gross yield = 110, fee = 5.5, net = 104.5
        uint256 expectedNet = 110 * 1e18 - (110 * 1e18 * 500 / 10000);
        assertEq(investToken.balanceOf(user1), userBalBefore + expectedNet);

        // Check claimed
        InvestmentVault.UserInvestment memory userInv = vault.getUserInvestment(user1, 0);
```

```
        assertTrue(userInv.fullyClaimed);
    }

    function test_WithdrawYield_RevertAlreadyClaimed() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);

        vm.warp(block.timestamp + 8 days);
        vm.prank(owner);
        vault.withdrawInvestments(0);
        vm.prank(owner);
        vault.depositYield(0, 110 * 1e18);

        vm.prank(user1);
        vault.withdrawYield(0);

        // Try to claim again
        vm.prank(user1);
        vm.expectRevert(IInvestmentVault.AlreadyClaimed.selector);
        vault.withdrawYield(0);
    }

    // ============ Multi-User Yield Distribution Tests ============

    function test_MultiUserYieldDistribution() public {
        // maxTotalLocked = 3000, fraction = 0.1, so max investable = 300
        vm.prank(owner);
        vault.createPeriod(3000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        // User1 locks and invests 100
        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);

        // User2 locks and invests 200
        vm.prank(user2);
        vault.lock(2000 * 1e18);
        vm.prank(user2);
        vault.invest(0, 200 * 1e18);

        vm.warp(block.timestamp + 8 days);

        vm.prank(owner);
        vault.withdrawInvestments(0);

        // Total invested: 300, deposit yield: 330 (10% profit)
        vm.prank(owner);
        vault.depositYield(0, 330 * 1e18);

        // User1: 100/300 * 330 = 110, fee 5% = 5.5, net = 104.5
        (uint256 gross1, uint256 net1, uint256 fee1) = vault.calculateUserYield(user1, 0);
```

```
        assertEq(gross1, 110 * 1e18);
        assertEq(fee1, 55 * 1e17); // 5.5
        assertEq(net1, 1045 * 1e17); // 104.5

        // User2: 200/300 * 330 = 220, fee 5% = 11, net = 209
        (uint256 gross2, uint256 net2, uint256 fee2) = vault.calculateUserYield(user2, 0);
        assertEq(gross2, 220 * 1e18);
        assertEq(fee2, 11 * 1e18);
        assertEq(net2, 209 * 1e18);
    }

    // ============ Period Finalization Tests ============

    function test_FinalizePeriod() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);

        vm.warp(block.timestamp + 8 days);
        vm.prank(owner);
        vault.withdrawInvestments(0);
        vm.prank(owner);
        vault.depositYield(0, 110 * 1e18);

        vm.prank(owner);
        vault.finalizePeriod(0);

        assertEq(uint(vault.getPeriodState(0)), uint(IInvestmentVault.PeriodState.FINALISED));
    }

    function test_SequentialPeriods() public {
        // Create first period
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        // User makes an investment so we can complete the period
        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);

        // Create second period (in INIT)
        vm.prank(owner);
        vault.createPeriod(2000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);

        // Cannot start second period while first is active
        vm.prank(owner);
        vm.expectRevert(IInvestmentVault.PreviousPeriodNotFinalised.selector);
        vault.startPeriod(1);

        // Complete first period
        vm.warp(block.timestamp + 8 days);
        vm.prank(owner);
```

```
        vault.withdrawInvestments(0);
        vm.prank(owner);
        vault.depositYield(0, 100 * 1e18); // Return principal
        vm.prank(owner);
        vault.finalizePeriod(0);

        // Now can start second period
        vm.prank(owner);
        vault.startPeriod(1);

        assertEq(uint(vault.getPeriodState(1)), uint(IInvestmentVault.PeriodState.INVEST));
    }

    // ============ Fee Withdrawal Tests ============

    function test_WithdrawFees() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);

        vm.warp(block.timestamp + 8 days);
        vm.prank(owner);
        vault.withdrawInvestments(0);
        vm.prank(owner);
        vault.depositYield(0, 110 * 1e18);

        vm.prank(user1);
        vault.withdrawYield(0);

        // Check collected fees
        uint256 fees = vault.collectedFees(address(investToken));
        assertEq(fees, 55 * 1e17); // 5.5

        address feeRecipient = address(100);
        vm.prank(owner);
        vault.withdrawFees(address(investToken), feeRecipient, fees);

        assertEq(investToken.balanceOf(feeRecipient), fees);
        assertEq(vault.collectedFees(address(investToken)), 0);
    }

    // ============ Pause Tests ============

    function test_Pause() public {
        vm.prank(owner);
        vault.pause();

        vm.prank(user1);
        vm.expectRevert();
        vault.lock(100 * 1e18);
    }

    function test_Unpause() public {
        vm.prank(owner);
        vault.pause();
```

```
        vm.prank(owner);
        vault.unpause();

        vm.prank(user1);
        vault.lock(100 * 1e18);
        assertEq(vault.getUserLocked(user1), 100 * 1e18);
    }

    // ============ View Function Tests ============

    function test_GetUnlockableBalance() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(1000 * 1e18);

        // Before investment, all is unlockable
        assertEq(vault.getUnlockableBalance(user1), 1000 * 1e18);

        // After investing 50 (requires 500 locked)
        vm.prank(user1);
        vault.invest(0, 50 * 1e18);

        assertEq(vault.getUnlockableBalance(user1), 500 * 1e18);
        assertEq(vault.getRequiredLocked(user1), 500 * 1e18);
    }

    function test_DepositYield_MultipleCalls() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);
        vm.warp(block.timestamp + 8 days);

        vm.prank(owner);
        vault.withdrawInvestments(0);

        vm.prank(owner);
        vault.depositYield(0, 110 * 1e18);

        vm.prank(owner);
        vm.expectRevert(IInvestmentVault.InvalidState.selector);
        vault.depositYield(0, 120 * 1e18);
    }

    function test_WithdrawInvestments_MultipleCalls_Revert() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
```

```
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);
        vm.warp(block.timestamp + 8 days);

        vm.prank(owner);
        vault.withdrawInvestments(0);
        vm.prank(owner);
        vm.expectRevert(IInvestmentVault.InvalidState.selector);
        vault.withdrawInvestments(0);
    }

    function test_WithdrawInvestments_MultipleCalls_WithExtraBalance_Revert() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 500,0,0);
        vm.prank(owner);
        vault.startPeriod(0);

        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);
        vm.warp(block.timestamp + 8 days);
        investToken.mint(address(vault), 100 * 1e18);

        uint256 ownerBalBefore = investToken.balanceOf(owner);

        vm.prank(owner);
        vault.withdrawInvestments(0);
        assertEq(investToken.balanceOf(owner), ownerBalBefore + 100 * 1e18);

        vm.prank(owner);
        vm.expectRevert(IInvestmentVault.InvalidState.selector);
        vault.withdrawInvestments(0);
    }

    function test_WithdrawInvestments_MultipleCalls_SameToken_Revert() public {
        // use lockToken as investToken
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(lockToken), 500, 0, 0);
        vm.prank(owner);
        vault.startPeriod(0);
        // user1 lock and invest
        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);
        // skip to withdraw time
        vm.warp(block.timestamp + 8 days);
        // mint invest tokens to the vault
        lockToken.mint(address(vault), 100 * 1e18);
        uint256 ownerBalBefore = lockToken.balanceOf(owner);
        // first withdraw
        vm.prank(owner);
        vault.withdrawInvestments(0);
        assertEq(lockToken.balanceOf(owner), ownerBalBefore + 100 * 1e18);
        // second call should revert
        vm.prank(owner);
        vm.expectRevert(IInvestmentVault.InvalidState.selector);
        vault.withdrawInvestments(0);
    }
```

```
    // ============ Emergency Cancel & Withdraw Tests ============

    function test_CancelPeriod_EmergencyWithdraw() public {
        vm.prank(owner);
        vault.createPeriod(1000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 0, 0, 0);
        vm.prank(owner);
        vault.startPeriod(0);
        vm.prank(user1);
        vault.lock(1000 * 1e18);
        vm.prank(user1);
        vault.invest(0, 100 * 1e18);
        vm.prank(owner);
        vault.cancelPeriod(0);
        uint256 userBalBefore = investToken.balanceOf(user1);
        vm.prank(user1);
        vault.emergencyWithdrawInvestment(0);
        assertEq(investToken.balanceOf(user1), userBalBefore + 100 * 1e18);
        // emergencyWithdrawInvestment again shoule be reverteds
        vm.prank(user1);
        vm.expectRevert(IInvestmentVault.AlreadyClaimed.selector);
        vault.emergencyWithdrawInvestment(0);
    }


    function test_MaxInvestPerWallet_Bypass_Issue() public {
        // pre period with no maxInvestPerWallet
        vm.prank(owner);
        vault.createPeriod(10000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 0, 0, 0); // no cap
        vm.prank(owner);
        vault.startPeriod(0);
        vm.prank(user1);
        vault.lock(5000 * 1e18);
        // Invest 1 token to ensure amount > 0
        vm.prank(user1);
        vault.invest(0, 1);
        // Move to next period
        vm.warp(block.timestamp + 8 days);
        vm.prank(owner);
        vault.withdrawInvestments(0);
        vm.prank(owner);
        vault.depositYield(0, 1); // deposit minimal yield
        vm.prank(owner);
        vault.finalizePeriod(0);
        // This period has a smaller maxInvestPerWallet
        vm.prank(owner);
        vault.createPeriod(10000 * 1e18, 30 days, 7 days, 1e17, address(investToken), 0, 0, 100 * 1e18);
        vm.prank(owner);
        vault.startPeriod(1);
        // Users do not need to lock again, they can invest directly
        vm.prank(user1);
        vault.invest(1, 100 * 1e18); // normal
        // Attempt to invest over the cap, should revert
        vm.prank(user1);
        vm.expectRevert(IInvestmentVault.ExceedsWalletInvestCap.selector);
        vault.invest(1, 200 * 1e18);
    }

}
```

2. UnitTestOutput

```
Ran 46 tests for test/InvestmentVault.t.sol:InvestmentVaultTest
[PASS] test_CancelPeriod_EmergencyWithdraw() (gas: 529852)
[PASS] test_Constructor() (gas: 21405)
[PASS] test_Constructor_RevertZeroAddress() (gas: 96282)
[PASS] test_CreatePeriod() (gas: 257700)
[PASS] test_CreatePeriod_RevertInvalidParameters() (gas: 52553)
[PASS] test_CreatePeriod_RevertNonOwner() (gas: 21467)
[PASS] test_DepositYield() (gas: 595566)
[PASS] test_DepositYield_MultipleCalls() (gas: 589506)
[PASS] test_DepositYield_RevertIfLessThanPrincipal() (gas: 503961)
[PASS] test_DepositYield_RevertIfNotWithdrawn() (gas: 492414)
[PASS] test_DepositYield_RevertIfZeroAmount() (gas: 501888)
[PASS] test_FinalizePeriod() (gas: 592042)
[PASS] test_GetPeriodState_AutoTransition() (gas: 303412)
[PASS] test_GetUnlockableBalance() (gas: 525588)
[PASS] test_Invest() (gas: 521070)
[PASS] test_Invest_LastMoment_UnlockAllAfterInvestWindow() (gas: 467132)
[PASS] test_Invest_Multiple() (gas: 535791)
[PASS] test_Invest_RevertAfterInvestWindow() (gas: 411826)
[PASS] test_Invest_RevertIfCollateralJustEnough() (gas: 394630)
[PASS] test_Invest_RevertInsufficientCollateral() (gas: 415058)
[PASS] test_Lock() (gas: 123693)
[PASS] test_Lock_RevertZeroAmount() (gas: 23238)
[PASS] test_MEV_TemporaryCollateral() (gas: 447386)
[PASS] test_MaxInvestPerWallet_Bypass_Issue() (gas: 929873)
[PASS] test_MultiUserYieldDistribution() (gas: 706506)
[PASS] test_MultiUser_ImbalancedInvestments() (gas: 682990)
[PASS] test_Pause() (gas: 53454)
[PASS] test_Pause_AllUserOpsBlocked() (gas: 86746)
[PASS] test_SequentialPeriods() (gas: 877745)
[PASS] test_StartPeriod() (gas: 313556)
[PASS] test_StartPeriod_RevertNotInit() (gas: 298793)
[PASS] test_Unlock() (gas: 137458)
[PASS] test_Unlock_PartialWithActiveInvestment() (gas: 529192)
[PASS] test_Unlock_RevertIfCollateralRequired() (gas: 496893)
[PASS] test_Unlock_RevertInsufficientBalance() (gas: 121539)
[PASS] test_Unlock_RevertWithActiveInvestment() (gas: 516133)
[PASS] test_Unpause() (gas: 130808)
[PASS] test_WithdrawFees() (gas: 661200)
[PASS] test_WithdrawInvestments() (gas: 517971)
[PASS] test_WithdrawInvestments_MultipleCalls_Revert() (gas: 524997)
[PASS] test_WithdrawInvestments_MultipleCalls_SameToken_Revert() (gas: 548724)
[PASS] test_WithdrawInvestments_MultipleCalls_WithExtraBalance_Revert() (gas: 562736)
[PASS] test_WithdrawYield() (gas: 660534)
[PASS] test_WithdrawYield_RevertAlreadyClaimed() (gas: 663560)
[PASS] test_WithdrawYield_RevertIfNoDepositYield() (gas: 505170)
[PASS] test_WithdrawYield_RevertTwice() (gas: 594089)
Suite result: ok. 46 passed; 0 failed; 0 skipped; finished in 3.75ms (13.44ms CPU time)

Ran 1 test suite in 130.41ms (3.75ms CPU time): 46 tests passed, 0 failed, 0 skipped (46 total tests)
```

11.2 External Functions Check Points

1. InvestmentVault.sol_check_point.md

File: src/InvestmentVault.sol

contract: InvestmentVault is IInvestmentVault, Ownable2Step, ReentrancyGuard, Pausable

(Empty fields in the table represent things that are not required or relevant)

Index	Function	StateMutability	Modifier	Param Check	IsUserInterface	Unit Test	Miscellaneous
1	createPeriod(uint256,uint256,uint256,uint256,address,uint256,uint256,uint256)		`onlyOwner`			Passed	
2	startPeriod(uint256)		`onlyOwner`			Passed	
3	withdrawInvestments(uint256)		`onlyOwner`, `nonReentrant`			Passed	
4	depositYield(uint256,uint256)		`onlyOwner`, `nonReentrant`			Passed	
5	finalizePeriod(uint256)		`onlyOwner`			Passed	
6	withdrawFees(address,address,uint256)		`onlyOwner`, `nonReentrant`			Passed	
7	pause()		`onlyOwner`			Passed	
8	unpause()		`onlyOwner`			Passed	
9	cancelPeriod(uint256)		`onlyOwner`, `nonReentrant`			Passed	
10	lock(uint256)		`nonReentrant`, `whenNotPaused`		Yes	Passed	
11	unlock(uint256)		`nonReentrant`, `whenNotPaused`		Yes	Passed	
12	invest(uint256,uint256)		`nonReentrant`, `whenNotPaused`		Yes	Passed	
13	withdrawYield(uint256)		`nonReentrant`, `whenNotPaused`		Yes	Passed	
14	emergencyWithdrawInvestment(uint256)		`nonReentrant`, `whenNotPaused`		Yes	Passed	
15	getPeriodState(uint256)	view				Passed	
16	getPeriod(uint256)	view				Passed	
17	getUserInvestment(address,uint256)	view				Passed	
18	getUserLocked(address)	view				Passed	
19	getUnlockableBalance(address)	view				Passed	
20	getRequiredLocked(address)	view				Passed	
21	calculateUserYield(address,uint256)	view				Passed	

FAIRYPROOF

https://medium.com/@FairyproofT

https://twitter.com/FairyproofT

https://www.linkedin.com/company/fairyproof-tech

https://t.me/Fairyproof_tech

Reddit: https://www.reddit.com/user/FairyproofTech