Memebox by Solidly Labs
Audit Report

Prepared by Cyfrin

Version 2.2

Lead Auditors

Giovanni Di Siena

Assisting Auditors

Hans

May 4, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Solidly V2 Memecore is a new smart contract developed by Solidly Labs and in use on the Memebox platform. It is a renewed interpretation and reimplementation of the `xy >= k` AMM written in Solidity version 0.8.19, specifically tailored to cater to memecoins.

Key features include:

- LP fee abstraction and configurable fee structure.
- Vertically-integrated locking mechanism & native memecoin functionality (e.g. locks, burns, split, extends, transfers).
- The ability to earn fees in perpetuity even after liquidity is burned.
- Lockers/burners can access their fees at any time.
- Laying the groundwork for yield tokenization by making the yield receipt fungible.
- Optional “Copilot” role for pool fees, allowing projects to manage their own pool fee as a replacement for fee-on-transfer tokens.

5 Audit Scope

Cyfrin conducted an audit of Solidly V2 Memecore based on the code present in the repository commit hash 757b18a, with remediations applied to the final commit hash 14533e7.

All contracts within the flattened `SolidlyV2-memecore.sol` file were included in the scope of the audit, including:

- Math
- SolidlyV2Accounting
- SolidlyV2LockBox

- SolidlyV2Pair
- SolidlyV2ERC42069
- SolidlyV2Factory

6 Executive Summary

Over the course of 6 days, the Cyfrin team conducted an audit on the Solidly V2 Memecore smart contracts provided by Solidly Labs. In this period, a total of 10 issues were found.

This review of the Solidly V2 Memecore contracts uncovered two instances of potentially undefined behavior that should be addressed, either in the form of remediation or documentation, regarding the LP locking functionality and setting of pool fees.

The inline documentation of the codebase is minimal, with some areas also remaining untested. As such, the test suite coverage could be improved to cover all core functionalities of the codebase. However, this is overall a well-architected and clean codebase that is easy to read and understand. Additionally, a number of black-box integration tests and both differential/invariant fuzz tests were written as part of the audit process to provide greater assurances as to the correctness of the code.

Summary

Project Name	Memebox by Solidly Labs
Repository	v2-core
Commit	757b18ad0578...
Audit Timeline	Apr 22nd - Apr 29th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	3
Informational	5
Gas Optimizations	1
Total Issues	10

Summary of Findings

[M-1] Locks can be created with expiry in the past	Resolved
[L-1] Accounts blocked by the token cannot claim their fees	Acknowledged

[L-2] Consider a two-step ownership transfer	Acknowledged
[L-3] `SolidlyV2Pair::setPoolFee` fails to adequately consider fee setters defined by `SolidlyV2Factory`	Acknowledged
[I-1] Flash loans can be taken for free	Acknowledged
[I-2] Lack of events emitted for state changes	Acknowledged
[I-3] Secondary markets for LP tokens are problematic	Acknowledged
[I-4] Users can hide liquidity in low-duration locks	Acknowledged
[I-5] Lock creation can be front-run	Acknowledged
[G-1] Cache domain separator and only recompute if the chain ID has changed	Acknowledged

7 Findings

7.1 Medium Risk

7.1.1 Locks can be created with expiry in the past

Description: Due to a silent overflow in `SolidityV2ERC42069::lockToken`, a sufficiently large duration will cause the unlock date to be in the past. This could allow the caller to create a fraudulent lock, advertising that they have locked for the maximum duration but which can actually be withdrawn immediately. However, the impact is somewhat limited as this will be visible to anyone who calls `SolidityV2ERC42069::getLock(s)` with the owner's address (perhaps in a UI). From the attacker's perspective, the extension functionality could ideally be used to wrap around at will, hiding this malicious intent; however, this is not possible due to the checked math revert when incrementing the date.

Impact: This bug has a high likelihood of being abused with a more limited impact; therefore, it is categorized as a medium-severity finding.

Proof of Concept: Append this test to `MultiTest.js`:

```
it("lock can be created in the past", async function () {
  const { user1, test0, test1, router, pair } = await loadFixture(deploySolidlyV2Fixture);

  let token0 = test0;
  let token1 = test1;

  // Approve tokens for liquidity provision
  await token0.connect(user1).approve(router.address, ethers.constants.MaxUint256);
  await token1.connect(user1).approve(router.address, ethers.constants.MaxUint256);

  // Provide liquidity
  await router.connect(user1).addLiquidity(
    token0.address,
    token1.address,
    ethers.utils.parseUnits("100", 18),
    ethers.utils.parseUnits("100", 18),
    0,
    0,
    user1.address,
    ethers.constants.MaxUint256
  );

  const liquidityBalance = await pair.balanceOf(user1.address);

  let blockTimestamp = (await ethers.provider.getBlock('latest')).timestamp;

  let maxUint128 = ethers.BigNumber.from("340282366920938463463374607431768211455");

  // Lock LP tokens
  await pair.connect(user1).lockToken(user1.address, liquidityBalance, maxUint128.sub(blockTimestamp));


  let ret = await pair.getLock(user1.address, 0);
  expect(ret.date).to.be.eq(0);
});
```

Recommended Mitigation: Cast the timestamp to `uint128` prior to performing the addition rather than unsafely downcasting the result of the addition:

```
locks[from].push(LockData({
    amount: uint128(amount - fee),
-     date: uint128(block.timestamp + duration)
```

```
+     date: uint128(block.timestamp) + duration
}));
```

Solidly Labs: Fixed in commit 14533e7.

Cyfrin: Verified. The timestamp is first cast to `uint128` prior to performing the addition.

7.2 Low Risk

7.2.1 Accounts blocked by the token cannot claim their fees

Description: In `SolidlyV2`, liquidity providers accrue fees from trades made in the pool. These fees are bound to the account holding the liquidity position at the time of trading as the fees are synced on transfers.

When the liquidity provider wants to claim their fees they call `SolidlyV2Pair::claimFees`:

```
if (feeState.feeOwed0 > 1) {
    amount0 = feeState.feeOwed0 - 1;
    _safeTransfer(token0, msg.sender, uint256(amount0));
    feesClaimedLP0 += amount0;
    feesClaimedTotal0 += amount0;
    feeState.feeOwed0 = 1;
}
```

Here the tokens are transferred to `msg.sender`.

The issue is that some tokens, such as `USDC`, have block lists where some accounts are blocked from receiving or doing transfers. Were this to happen to `msg.sender`, they would never be able to claim their fees, and these funds would be locked in the pool indefinitely.

Impact: If an account is blocked by, for example, `USDC`, the user will not be able to claim their fees.

Recommended Mitigation: Add a parameter `address to` to `SolidlyV2Pair::claimFees` so that the fees can be transferred to a different account.

Solidly Labs: Acknowledged.

Cyfrin: Acknowledged.

7.2.2 Consider a two-step ownership transfer

Description: In the `SolidtlyV2-memecore` contracts the owner is set as `msg.sender` in `SolidlyV2Factory` when deployed.

It can also be changed using `SolidlyV2Factory::setOwner`:

```
    function setOwner(address _newOwner) external {
        require(msg.sender == owner);
        owner = _newOwner;
    }
```

Here there's a risk that the owner is lost if the new address not correct. This would result in some functionality not being available, like assigning new copilots and adjusting fees.

Impact: The `owner` can mistakenly be given to an account that is out of the protocol's control.

Recommended Mitigation: Consider implementing a two-step ownership transfer where `owner` first sets a `pendingOwner` then the `pendingOwner` can accept the new ownership , like OpenZeppelin `Ownable2Step`

Solidly Labs: Acknowledged.

Cyfrin: Acknowledged.

7.2.3 `SolidlyV2Pair::setPoolFee` fails to adequately consider fee setters defined by `SolidlyV2Factory`

Description: If a new `feeSetter` is registered by the owner of `SolidlyV2Factory`, then it is possible for the pool fee to be set below the minimum specified by the factory. This breaks the invariant that the pool fee should always be between the defined min/max values since it is only possible for the protocol to influence this value by explicitly

modifying it on the factory itself, whereas a fee setter can modify it on the pool directly to become out of sync with the factory.

```
function setPoolFee(uint16 _poolFee) external {
    require(ISolidlyV2Factory(factory).isFeeSetter(msg.sender) || msg.sender == copilot, 'UA');
    if (msg.sender == copilot) {
        require(_poolFee >= ISolidlyV2Factory(factory).minFee() && !copilotRevoked); // minimum fee
        ↪  enforced for copilot
    } else {
        require(!protocolRevoked);
    }
    require(_poolFee <= 1000); // pool fee capped at 10%
    uint16 feeOld = poolFee;
    poolFee = _poolFee;
    emit SetPoolFee(feeOld, _poolFee);
}
```

Additionally, if the owner of `SolidlyV2Factory` calls `SolidlyV2Pair::revokeFeeRole`, then registered fee setters are no longer able to call `SolidlyV2Pair::setPoolFee` despite no explicit consideration of fee setters.

Impact:

- The pool fee can become out of sync with `SolidlyV2Factory::minFee`.
- Fee setters cannot set fees once the protocol has revoked its fee role.

Proof of Concept:

```
it("fee below min", async function () {
  const { user1, factory, pair } = await loadFixture(deploySolidlyV2Fixture);

  await factory.setFeeSetter(user1.address, true);
  pair.connect(user1).setPoolFee(0);
  await factory.minFee().then(minFee => console.log(`SolidlyV2Factory::minFee: ${minFee}`));
  await pair.poolFee().then(poolFee => console.log(`SolidlyV2Pair::poolFee: ${poolFee}`));

  await pair.revokeFeeRole();
  const minFee = await factory.minFee();
  await expect(pair.connect(user1).setPoolFee(minFee)).to.revertedWithoutReason();
});
```

Recommended Mitigation: Explicitly handle calls from fee setters as distinct from the factory owner in `SolidlyV2Pair::setPoolFee`, also enforcing that the pool fee cannot be set below the defined global minimum.

Solidly Labs: Acknowledged. Both these things are like this by design – `feeSetters` are not constrained by `minFee`, and both protocol and `feeSetter` are considered the same for revoke.

Cyfrin: Acknowledged.

7.3 Informational

7.3.1 Flash loans can be taken for free

Description: Unlike other AMM implementations, `SolidlyV2Pair` calculates fees independently for each underlying pool token. Showing just the `amount0Out` calculation:

```
if (amount0Out > 0) {
    uint256 amount1In = balance1 - (_reserve1 - amount1Out);
    uint256 fee1 = _updateFees(1, amount1In, _poolFee, _protocolRatio);
    _k(balance0, balance1 - fee1, uint256(_reserve0), uint256(_reserve1));
    _updateReserves(balance0, (balance1 - fee1));
    _emitSwap(0, amount1In, amount0Out, amount1Out, to);
}
```

Since it is only possible to swap a single token, say `token0`, only `token0Out` will be non-zero. Since `amount1Out` is `0` the calculation `uint256 amount1In = balance1 - (_reserve1 - amount1Out);` will yield `amount1In = 0`. Hence, no fee will be charged.

Impact: This has a high likelihood, but it is up to the project to determine the impact of being able to take flash swaps for free.

Proof of Concept: Add this test to `MultiTest.js`:

```
it("charges no fee for flashloans", async function () {
const { user1, test0, test1, router, pair } = await loadFixture(deploySolidlyV2Fixture);

let token0 = test0;
let token1 = test1;


// Approve tokens for liquidity provision
await token0.connect(user1).approve(router.address, ethers.constants.MaxUint256);
await token1.connect(user1).approve(router.address, ethers.constants.MaxUint256);

// Provide liquidity
await router.connect(user1).addLiquidity(
  token0.address,
  token1.address,
  ethers.utils.parseUnits("100", 18),
  ethers.utils.parseUnits("100", 18),
  0,
  0,
  user1.address,
  ethers.constants.MaxUint256
);

const FlashRecipient = await ethers.getContractFactory("FlashRecipient");
const flashRecipient = await FlashRecipient.deploy(token0.address, pair.address);

// doesn't revert
await flashRecipient.takeFlashloan();
});
```

with this file in `contracts/test/FlashRecipient.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

import {SolidlyV2Pair} from "../SolidlyV2Pair.sol";
import {IERC20} from "./IERC20.sol";
```

```
contract FlashRecipient {

    IERC20 public token0;
    SolidlyV2Pair public pair;

    constructor(address _token0, address _pair) {
        token0 = IERC20(_token0);
        pair = SolidlyV2Pair(_pair);
    }

    function takeFlashloan() external {
        pair.swap(1e18, 0, address(this), "data");
    }

    function solidlyV2Call(address , uint256 amount0Out, uint256, bytes memory) external returns (bool)
    ↪ {
        // no fee being paid
        token0.transfer(msg.sender, amount0Out);
        return true;
    }
}
```

Recommended Mitigation: Calculate the fee for both sides, but only call `_updateFees()` if they have changed.

Solidly Labs: Acknowledged. Flashloans are not a big market, especially with memecoins, we see more potential benefits in leaving them free of charge.

Cyfrin: Acknowledged.

7.3.2 Lack of events emitted for state changes

Description: Events are useful to track contract changes off-chain. Hence emitting events for state changes is very valuable for off-chain monitoring and tracking of on-chain state.

Impact: Important state changes can be missed in monitoring and off chain tracing.

Recommended Mitigation: Consider emitting events for:

- `SolidlyV2Factory::setOwner`
- `SolidlyV2Factory::setFeeCollector`
- `SolidlyV2Factory::setFeeSetter`
- `SolidlyV2Pair::setCopilot`
- `SolidlyV2Pair::revokeFeeRole`

Also consider emitting a special event when tokens are burnt (but kept tracked for fee accrual):

- `SolidlyV2ERC42069::transferZero`
- `SolidlyV2ERC42069::transferZeroFrom`

Solidly Labs: Partially fixed. Most important events are covered, added `OwnerChanged` on factory. We're very constrained by bytecode space.

Cyfrin: Acknowledged.

7.3.3 Secondary markets for LP tokens are problematic

Description: Unlike other AMM implementations, where fees are accrued in the LP token, Solidly V2 implements an accrual system where the fees are tracked in `feeGrowthGlobal0/1` per LP token. Each liquidity provider can

call `SolidlyV2Pair::claimFees` to collect their accrued fees, with the benefit being that a liquidity provider does not have to burn their position to access the fees they have accrued.

Fee accrual is synced with every action that changes the balance: when claiming fees, mints, locks, and, most importantly, on any transfer of LP tokens or locked or burnt positions. This means that the fees accrued are bound to the account holding the liquidity position at that time.

Therefore, if these tokens are traded or used as collateral on secondary markets, the fees accrued while in escrow would be lost as these contracts cannot, without modification and knowledge of the Solidly V2 Memecore system, claim fees.

Impact: Using Solidly V2 Memecore LP tokens in AMMs, such as within Solidly V2 itself, or different escrow/collateral style contracts will cause fees to be lost.

Recommended Mitigation: Be clear about this behavior in the documentation. For the LP tokens to be safely used in a pool of any sort, these would need an ERC-4626-style wrapper vault to be developed. The locked/burnt positions are not so concerning as they would need a custom-built implementation to be traded anyway, which would cater to this in its development..

Solidly Labs: Acknowledged. Secondary markets for memecoin LPs are close to non-existent anyway. The most important functions with meme LP tokens are directly available in the core contracts.

Cyfrin: Acknowledged.

7.3.4 Users can hide liquidity in low-duration locks

Description: One of the main features of SolidlyV2-memecore is the ability to lock liquidity for a period of time. This will transfer the liquidity to a `LockBox` contract, where it will be locked until the expiry (which is specified when locking).

This liquidity still accrues fees for the holder but is technically held by the `LockBox` contract. Thus `balanceOf` will show `0` for the account having the lock.

This could be used to "hide" liquidity as a user could have a lock with a `0` duration and then, when needing the liquidity, just withdraw the lock.

Impact: A user could use locks to "hide" liquidity since a normal ERC20 `balanceOf` will not show it. However, there appears to be no clear way of abusing this, and hence, this is only for your information.

Recommended Mitigation: If this is a concern, consider including locked liquidity as part of the `balanceOf`. This would break the ERC20 accounting, though, as the tokens would be double-counted on both the `LockBox` balance and the account balance.

Solidly Labs: Acknowledged. Users can check that it's expired or close to expiry.

Cyfrin: Acknowledged.

7.3.5 Lock creation can be front-run

Description: `SolidlyV2ERC42069::lockToken` can be front-run with the transfer of a dust lock. This would cause the index at which the lock is created to be different from what the sender expected when sending the transaction.

In the worst case, this could cause the wrong lock to be withdrawn, extended, or split. If the sender is attentive, this can be easily remedied, but it could waste the gas cost for one TX.

Impact: The index at which a lock is created/split/transferred can be something other than expected.

Recommended Mitigation: Consider mentioning in the documentation that the user needs to use the events emitted to verify at which index their lock was created.

Solidly Labs: Acknowledged.

Cyfrin: Acknowledged.

7.4 Gas Optimization

7.4.1 Cache domain separator and only recompute if the chain ID has changed

For greater gas efficiency, it is recommended that the current chain ID be cached on contract creation and that the domain separator be recomputed only if a change of chain ID is detected (i.e., `block.chainid` != cached chain ID). An example can be seen in the implementation of `Solmate::ERC20`.

Solidly Labs: Acknowledged.

Cyfrin: Acknowledged.