Protocol Audit Report

Prepared by: Bizarro

Table of Contents

Protocol Summary

The Curves protocol, an extension of friend.tech, introduces several innovative features. For context on friend.tech, consider this insightful article: Friend Tech Smart Contract Breakdown. Key enhancements in the Curves protocol include:

Token Export to ERC20: This pivotal feature allows users to transfer their tokens from the Curves protocol to the ERC20 format. Such interoperability significantly expands usability across various platforms. Within Curves, tokens lack decimal places, but when converted to ERC20, they adopt a standard 18-decimal format. Importantly, users can seamlessly reintegrate their ERC20 tokens into the Curves ecosystem, albeit only as whole, integer units.

Referral Fee Implementation: Curves empowers protocols built upon its framework by enabling them to earn a percentage of all user transaction fees. This incentive mechanism benefits both the base protocol and its derivative platforms.

Presale Feature: Learning from the pitfalls of friend.tech, particularly issues with frontrunners during token launches, Curves incorporates a presale phase. This allows creators to manage and stabilize their tokens prior to public trading, ensuring a more controlled and equitable distribution.

Token Holder Fee: To encourage long-term holding over short-term trading, Curves introduces a fee distribution model that rewards token holders. This fee is proportionally divided among all token holders, incentivizing sustained investment in the ecosystem.

These additions by Curves not only enhance functionality but also foster a more robust and inclusive financial ecosystem.

Disclaimer

Bizarro found as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of 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.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Scope

```
The audit will encompass the following files, each integral to the
functioning of the protocol:

Curves.sol: This is the primary file of the Curves protocol. It contains
the core logic and functions that define the overall behavior and rules of
the system. This file started as a fork of friend.tech
FriendtechSharesV1.sol

CurvesERC20.sol: This file defines the ERC20 token that will be created
upon exporting a Curve token. It outlines the token's properties and
behaviors consistent with the ERC20 standard.

CurvesERC20Factory.sol: Serving as an ERC20 factory, this file abstracts
the logic for ERC20 token creation. Its primary purpose is to streamline
the token creation process and reduce the overall footprint of the
protocol, ensuring efficiency and scalability.

FeeSplitter.sol: This script manages the distribution of fees. It is
responsible for the fair and accurate division of transaction fees amongst
token holders, in line with the protocol's incentive structure.

Security.sol: This file standardizes the security criteria for the
protocol. It includes protocols and measures designed to safeguard the
system against vulnerabilities and ensure compliance with established
```

```
security standards.

Each of these files plays a crucial role in the protocol's architecture
and functionality. The audit will methodically evaluate them for security,
efficiency, and adherence to best practices in smart contract development.
```

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	0
Low	0
Info	0
Gas	0
Total	3

Findings

High

[H-1] `Curves::_addOwnedCurvesTokenSubject` is susceptible to denial of service error.

Description: In `Curves::_addOwnedCurvesTokenSubject` to add a tokenSubject address to the `ownedCurvesTokenSubjects` the function iterates over the whole array of subject addresses to check if the user already owns the tokenSubject's tokens, here If the `ownedCurvesTokenSubjects[owner]` is too large then the function will revert causing the protocol to deny the service.

Impact: A whitelisted account can be subjected to a denial-of-service (DoS) attack by a malicious actor during the presale, preventing the account holder from purchasing curveTokens. As a result, the affected user may miss the presale opportunity and would only be able to acquire curveTokens during the open sale, potentially requiring the use of a different account.

Proof of Concept: When a whitelisted account purchases curveTokens during the presale, the user calls the `Curves::buyCurvesTokenWhitelisted()`, internally this function verifies the provided proof and if valid, calls the `Curves::buyCurvesToken` where if the account is purchasing `curveTokens` of the `curveTokenSubject` for the first time (which it does, because it the first purchase during the presale), it calls the `Curves::_addOwnedCurvesTokenSubject()`, and in this function, it will load all the `curveTokens` owned by the account `ownedCurvesTokenSubjects`, then it will iterate over this array, if the address of the `curvesTokenSubject` is already in the user's array (which is not, because the account

is buying `curveTokens` of this `curvesTokenSubject` for the first time), the function just returns, if the `curvesTokenSubject` is not in the array, then the `curvesTokenSubject` is added to the array.

1. This causes the malicious parties to DoS whitelisted accounts from buying during a presale it is the fact that the `ownedCurvesTokenSubjects` is loaded from storage, and the variable that is iterated on the for loop inside the `Curves::_addOwnedCurvesTokenSubject() function` is loaded from storage, thus, it will consume more gas, thus, less iteration will be able to do.

2. As array only grows, even though the account dumps/sells/transfers/withdraw `curveTokens` from different `curveTokenSubjects`, the array `ownedCurvesTokenSubjects` never decreases, once it has added a registry, it will be there forever. The only way to decrease the length of an array is by poping values from it, but, right now, it is not implemented anywhere that if the user stops owning `curveTokens` of a `curveTokenSubject` the address of the `curveTokenSubject` is poped from the `ownedCurvesTokenSubjects` array.

3. the `ownedCurvesTokenSubjects` array can be manipulated at will by external parties, not only by the account itself. When any account does a transfer of `curveTokens` to another account, either by calling the `Curves::transferCurvesToken() function` or the `Curves::transferAllCurvesTokens() function`, any of the two functions will internally call the `Curves::_transfer() function`, which it will call the `Curves::_addOwnedCurvesTokenSubject() function` and update the `ownedCurvesTokenSubjects` array of the `to` address.

Attack vector

A `subjectToken` launches a presale and whitelists X numbers of accounts to allow them to participate in the presale, A malicious user creates a new `curveTokens` using random accounts, then using a single account buys 1 `curveTokens` of all the random `tokenSubjects`, then proceeds to call the `Curves::transferAllCurvesTokens()` as a result of this, in a single call, the malicious user will inflate the `ownedCurvesTokenSubjects` array of the whitelisted account, this will cause that when the whitelisted account attempts to purchase the `curveTokens` of the real `subjectToken` during the presale, his transaction will be reverted because his `ownedCurvesTokenSubjects` array was filled with registries of nobodies `subjectTokens`, the array was inflated so much till the point that it can't be iterated and will fail with a gas exception. The whitelisted account has no means to pop any of those registries from his `ownedCurvesTokenSubjects` array, and as a result of this, the user who owns the whitelisted account was forcefully dosed during the presale, now, the only way for him to acquire `curveTokens` of that `tokenSubject` will be by using a different account during the open-sale.

Recommended Mitigation:

The most straightforward mitigation to prevent the permanent DoS is to implement logic that allows accounts to get rid of `curveTokens` from `tokenSubjects` they don't want to own.

Make sure to implement the `pop()` functionality to the `ownedCurvesTokenSubjects` array when the account doesn't own any `curveToken` of a `tokenSubject`.

- This should be implemented in the functions `Curves::sellCurvesToken() function` & `Curves::_transfer() function`. Whenever the account's `curvesTokenBalance` is updated on any of these two functions, make sure to validate if the post balance is 0, if so, pop the `subjectToken`'s address from the `ownedCurvesTokenSubjects` array of the account.

By allowing users to have control over their `ownedCurvesTokenSubjects` array, there won't be any incentive from third parties to attempt to cause a DoS by inflating the users' `ownedCurvesTokenSubjects` array, now, each user will be able to clean up their array as they please.

A more elaborated mitigation that will require more changes across the codebase is to use `EnumerableSets` instead of arrays, and make sure to implement correctly the functions offered by the `EnumerableSets`, such as `.contain()`, `.delete()` and `.add()`.

- But in the end, the objective must be the same, allow users to have control over their `ownedCurvesTokenSubjects`, if they stop owning a `curveToken` of a certain `tokenSubject`, remove that address from the `ownedCurvesTokenSubjects` variable.

[H-2] Unrestricted claiming of fees due to missing balance updates in `FeeSliptter`

Description: The `FeeSplitter` contract is responsible for distributing fees among token holders, it employs an accumulator pattern to distribute rewards over time among users who do not sell or withdraw their tokens as ERC20s. The pattern changes the total reward overtime, and updates it for each user everytime their balance changes.

The current implementation does not update the accumulator associated with each user during token transfers, deposits or withdrawals. The `onBalanceChange()` function, which is responsible for updating the accumulator following changes in the user's balance. Which is only called during buy and sell transactions.

This can be easily exploited to drain the `FeeSplitter` contract of its fees. An attacker could repeatedly transfer the same tokens to new accounts and claim fees everytime. This is possible because `data.userFeeOffset[account]` will be zero for every new account they transfer to, while the claimable rewards are calculated using the current balance returned by the Curves contract.

Since there is no limit to the amount of rewards that may accumulate in the `FeeSplitter` contract, this can be considered loss of matured yield and is hence classified as high severity.

Proof of Concept:

consider the scenario:

1. Alice buys any amount of curve tokens.
2. Alice transfersd her tokens to a new account, Account1, by calling `transferCurvesToken()`
3. The `onBalanceChange()` is not triggered during the transfer, so `data.userFeeOffset[account]` from Account1 is not updated.
4. Account1 can now call `FeeSplitter.claimFees()` and will receive fees as if it had been holding the tokens since the beginning, as `data.userFeeOffset[account]` for this account is 0.
5. Alice creates another new account, Account2, and transfers the tokens from Account1 to Account2.
6. The `onBalanceChange(token, account)` function is not triggered during this transfer, so `data.userFeeOffset[account]` for Account2 is zero.
7. Account2 can now claim fees.
8. Alice can repeat this process, creating new accounts and transferring tokens between them, to drain the contract of its fees.

Recommended Mitigation: To mitigate this issue, the `onBalanceChange(token, account)` function should be triggered during all token transfers, deposits, and withdrawals. For token transfers, it should be triggered for both accounts. This will ensure that `userFeeOffset` is accurately tracked, preventing users from claiming fees multiple times.

[H-3] Any `CurveSubjects` clould be turned to a `HoneyPot` by the creator of `CurveSubject`, which causes that users can only buy but can't sell the curve tokens any more. Then malicious creators can sell their own tokens at a high price to make profit.

Description:

Impact:

Proof of Concept:

```
File: contracts\Curves.sol
218:     function _transferFees(
...
224:     ) internal {
225:         (uint256 protocolFee, uint256 subjectFee, uint256
referralFee, uint256 holderFee, ) = getFees(price);
226:         {
227:             bool referralDefined =
referralFeeDestination[curvesTokenSubject] != address(0);
228:             {
229:                 address firstDestination = isBuy ?
feesEconomics.protocolFeeDestination : msg.sender;
230:                 uint256 buyValue = referralDefined ? protocolFee :
protocolFee + referralFee;
231:                 uint256 sellValue = price - protocolFee - subjectFee
- referralFee - holderFee;
232:                 (bool success1, ) = firstDestination.call{value:
isBuy ? buyValue : sellValue}("");
233:                 if (!success1) revert CannotSendFunds();
234:             }
235:             {
236:                 (bool success2, ) = curvesTokenSubject.call{value:
subjectFee}("");
237:                 if (!success2) revert CannotSendFunds();
238:             }
239:             {
240:                 (bool success3, ) = referralDefined
                        // @audit always be called even when
referralFee is 0
241:                     ?
referralFeeDestination[curvesTokenSubject].call{value: referralFee}("")
242:                     : (true, bytes(""));
243:                 if (!success3) revert CannotSendFunds();
244:             }
245:
246:             if (feesEconomics.holdersFeePercent > 0 &&
address(feeRedistributor) != address(0)) {
```

```
247:                 feeRedistributor.onBalanceChange(curvesTokenSubject,
msg.sender);
248:                 feeRedistributor.addFees{value: holderFee}
(curvesTokenSubject);
249:             }
250:         }
...
261:     }
```

POC

```
import { expect, use } from "chai";
import { solidity } from "ethereum-waffle";
use(solidity);
import { SignerWithAddress } from "@nomiclabs/hardhat-ethers/signers";
//@ts-ignore
import { ethers } from "hardhat";

import { type Curves } from "../contracts/types";
import { buyToken } from "../tools/test.helpers";
import { deployCurveContracts } from "./test.helpers";

describe("Make Curve Subject To Be A Honey Pot test", () => {
  let testContract;
  let evilReferralFeeReceiver;
  let owner: SignerWithAddress, evilSubjectCreator: SignerWithAddress,
      alice: SignerWithAddress, bob: SignerWithAddress, others:
SignerWithAddress[];
  beforeEach(async () => {
    testContract = await deployCurveContracts();
    [owner, evilSubjectCreator, alice, bob, others] = await
ethers.getSigners();
    const EvilReferralFeeReceiver = await
ethers.getContractFactory("EvilReferralFeeReceiver");
    evilReferralFeeReceiver = await
EvilReferralFeeReceiver.connect(evilSubjectCreator).deploy();
  });

  it("While 'HONEY POT' mode enabled, users can only buy, but can't sell
any Curve token ", async () => {
    // 1. The evil create a subject and set a normal referral fee receiver
    await
testContract.connect(evilSubjectCreator).mint(evilSubjectCreator.address);
    await
testContract.connect(evilSubjectCreator).setReferralFeeDestination(
      evilSubjectCreator.address, evilSubjectCreator.address
    );

    // 2. The evil buy enough tokens at low price
    await buyToken(testContract, evilSubjectCreator, evilSubjectCreator,
1);
    await buyToken(testContract, evilSubjectCreator, evilSubjectCreator,
```

```
10);

    // 3. victims buy or sell tokens normally
    await buyToken(testContract, evilSubjectCreator, alice, 10);
    await
testContract.connect(alice).sellCurvesToken(evilSubjectCreator.address,
5);
    await buyToken(testContract, evilSubjectCreator, bob, 10);
    await
testContract.connect(bob).sellCurvesToken(evilSubjectCreator.address, 5);

    // 4. at some time point, the evil enables 'HONEY POT' mode by
updating the referral fee receiver
    await
testContract.connect(evilSubjectCreator).setReferralFeeDestination(
      evilSubjectCreator.address, evilReferralFeeReceiver.address
    );
    await
evilReferralFeeReceiver.connect(evilSubjectCreator).setCurvesAndSubject(
      testContract.address, evilSubjectCreator.address
    );
    await
evilReferralFeeReceiver.connect(evilSubjectCreator).updateBalances(
      [alice.address, bob.address]
    );


    // 5. now, victims can buy, but can't sell
    await buyToken(testContract, evilSubjectCreator, alice, 1);
    let tx =
testContract.connect(alice).sellCurvesToken(evilSubjectCreator.address,
1);
    expect(tx).to.revertedWith("CannotSendFunds()");

    // 6. but the evil can sell tokens normally, of course at a higher
price than buy and make profit
    await
evilReferralFeeReceiver.connect(evilSubjectCreator).setAllowList(
      evilSubjectCreator.address, true
    );

testContract.connect(evilSubjectCreator).sellCurvesToken(evilSubjectCreato
r.address, 11);
  });

});
```

Recommended Mitigation: Solady's `forceSafeTransferETH()` seems suitable for this case: