CURVE STABLECOIN (CRVUSD) SECURITY AUDIT REPORT

June 5, 2023

MixBytes()

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only. The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the Client. If you are not the intended recipient(s) of this document, please note that any disclosure, copying or dissemination of its content is strictly forbidden.

1.2 Security Assessment Methodology

A group of auditors are involved in the work on the audit. The security engineers check the provided source code independently of each other in accordance with the methodology described below:

1. Project architecture review:

- Project documentation review.
- General code review.
- Reverse research and study of the project architecture on the source code alone.

Stage goals
- Build an independent view of the project's architecture.
- Identifying logical flaws.

2. Checking the code in accordance with the vulnerabilities checklist:

- Manual code check for vulnerabilities listed on the Contractor's internal checklist. The Contractor's checklist is constantly updated based on the analysis of hacks, research, and audit of the clients' codes.
- Code check with the use of static analyzers (i.e Slither, Mythril, etc).

Stage goal

Eliminate typical vulnerabilities (e.g. reentrancy, gas limit, flash loan attacks etc.).

3. Checking the code for compliance with the desired security model:

- Detailed study of the project documentation.
- Examination of contracts tests.
- Examination of comments in code.
- Comparison of the desired model obtained during the study with the reversed view obtained during the blind audit.
- Exploits PoC development with the use of such programs as Brownie and Hardhat.

Stage goal

Detect inconsistencies with the desired model.

4. Consolidation of the auditors' interim reports into one:

- Cross check: each auditor reviews the reports of the others.
- Discussion of the issues found by the auditors.
- Issuance of an interim audit report.

Stage goals

- Double-check all the found issues to make sure they are relevant and the determined threat level is correct.
- Provide the Client with an interim report.

5. Bug fixing & re-audit:

- The Client either fixes the issues or provides comments on the issues found by the auditors. Feedback from the Customer must be received on every issue/bug so that the Contractor can assign them a status (either "fixed" or "acknowledged").
- Upon completion of the bug fixing, the auditors double-check each fix and assign it a specific status, providing a proof link to the fix.
- A re-audited report is issued.

Stage goals

- Verify the fixed code version with all the recommendations and its statuses.
- Provide the Client with a re-audited report.

6. Final code verification and issuance of a public audit report:

- The Customer deploys the re-audited source code on the mainnet.
- The Contractor verifies the deployed code with the re-audited version and checks them for compliance.
- If the versions of the code match, the Contractor issues a public audit report.

Stage goals

- Conduct the final check of the code deployed on the mainnet.
- Provide the Customer with a public audit report.

Finding Severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss funds.
Low	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Customer regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Customer is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

1.3 Project Overview

The core idea of the stablecoin design is Lending-Liquidating AMM Algorithm. The idea is that it converts between collateral (for example, ETH) and the stablecoin (let's call it USD here). If the price of collateral is high - a user has deposits all in ETH, but as it goes lower, it converts to USD.

1.4 Project Dashboard

Project Summary

Title	Description
Client	Curve Finance
Project name	Curve Stablecoin (crvUSD)
Timeline	April 10 2023 - June 01 2023
Number of Auditors	3

Project Log

Date	Commit Hash	Note
10.04.2023	0d9265cc2dbd221b0f27f880fac1c590e1f12d28	Commit for the audit

Date	Commit Hash	Note
30.05.2023	c5169a7eb687a9878b989696a5c813dfc737e377	Commit for the reaudit

Project Scope

The audit covered the following files:

File name	Link
AggMonetaryPolicy.vy	AggMonetaryPolicy.vy
AggregateStablePrice.vy	AggregateStablePrice.vy
CryptoWithStablePrice.vy	CryptoWithStablePrice.vy
EmaPriceOracle.vy	EmaPriceOracle.vy
PegKeeper.vy	PegKeeper.vy
AMM.vy	AMM.vy
Controller.vy	Controller.vy
ControllerFactory.vy	ControllerFactory.vy
Stablecoin.vy	Stablecoin.vy

Deployments

File name	Contract deployed on mainnet	Comment
Stablecoin	0xf939E0A03FB07F59A73314E73794Be0E57ac1b4E	
ControllerFactory	0xC9332fdCB1C491Dcc683bAe86Fe3cb70360738BC	

File name	Contract deployed on mainnet	Comment
AMM	0x136e783846ef68C8Bd00a3369F787dF8d683a696	
Controller	0x8472A9A7632b173c8Cf3a86D3afec50c35548e76	
AggMonetaryPolicy	0xc684432FD6322c6D58b6bC5d28B18569aA0AD0A1	
AggregateStablePrice	0xe5Afcf332a5457E8FafCD668BcE3dF953762Dfe7	
PegKeeper	0xaA346781dDD7009caa644A4980f044C50cD2ae22	USDC
PegKeeper	0xE7cd2b4EB1d98CD6a4A48B6071D46401Ac7DC5C8	USDT
PegKeeper	0x6B765d07cf966c745B340AdCa67749fE75B5c345	USDP
PegKeeper	0x1ef89Ed0eDd93D1EC09E4c07373f69C49f4dcCae	TUSD

1.5 Summary of findings

Severity	# of Findings
Critical	2
High	2
Medium	4
Low	6

ID	Name	Severity	Status
C-1	Withdrawal of tokens from AMM	Critical	Fixed

C-2	Inflation attack on empty ticks	Critical	Fixed
H-1	Withdrawal of tokens with debt as zero	High	Fixed
H-2	`AMM.exchange()` produces negative fees	High	Fixed
M-1	Raw calls on ETH transfers allow reentrancy	Medium	Fixed
M-2	Not formalized stablecoin minting to addresses	Medium	Acknowledged
M-3	Early liquidations via `repay()` front-running	Medium	Fixed
M-4	`AggregateStablePrice` can be manipulated	Medium	Acknowledged
L-1	No checks of the result of `AMM.withdraw`	Low	Acknowledged
L-2	`AMM.exchange()` has no deadline	Low	Acknowledged
L-3	Not compatible with rebase/deflationary/hookable tokens and tokens with fees	Low	Acknowledged
L-4	Key functions do not have return values	Low	Acknowledged
L-5	ControllerFactory has unnecessary calculation operations	Low	Acknowledged
L-6	ControllerFactory.set_admin() does not have two-step ownership transferring	Low	Acknowledged

1.6 Conclusion

During the audit process 2 CRITICAL, 2 HIGH, 4 MEDIUM, and 6 LOW severity findings were spotted. After working through the reported findings, all of them were acknowledged or fixed by the client.

2.FINDINGS REPORT

2.1 Critical

C-1	Withdrawal of tokens from AMM
Severity	Critical
Status	Fixed in c5169a7e

Description

`Controller` allows to call `AMM` via callback:

```
withdraw_sig = get_method_id("withdraw(address,uint256)") # AMM

controller.liquidate_extended(user, 0, frac, True,
    market_amm.address, withdraw_sig, [])
```

AMM.vy#L728

```
def withdraw(user: address, frac: uint256) -> uint256[2]:
```

This method is sufficient to fulfill all the necessary conditions for a callback (Controller.vy#L525).

Using `liquidate_extended`, a hacker has the ability to withdraw any amount from `AMM`. It is also possible to make a complete liquidation through a partial one.

The test script was handed over to the customer.

Recommendation

It is recommended to use a specific signature for calling a callback.

Client's commentary

> Used specific callbacks now (such as callback_liquidate() etc).

C-2	Inflation attack on empty ticks
Severity	Critical
Status	Fixed in c5169a7e

Description

Each AMM tick represents an empty vault, where shares are issued for collateral. A hacker can manipulate a tick so that it contains just 1 wei share and any amount of collateral. For example, suppose the hacker initially inflates the tick so that it contains 1 wei share and 1 ETH. Next, the hacker sees a victim's transaction in the mempool, which is going to deposit 20 ETH into this tick. The hacker then needs to inflate the tick to contain 1 wei share and 10 ETH + 1 wei right before the victim's transaction.

How many shares will the victim receive in this tick? The victim receives 1 wei share due to a rounding error:

```
1 wei share * 20 ETH / (10 ETH + 1 wei) = 1 wei share
```

Now there are 2 wei shares in total in this tick, one for the victim and one for the hacker.

The hacker then self-liquidates and receives 50% of the ether from the entire tick, which is 15 ETH, even though they initially invested 10 ETH. The profit is +5 ETH.

How does the hacker inflate the collateral in the tick?

Step 1. Before the frontrunning, the hacker ensures that the tick contains 1 share and 100+ wei ETH. They can do it using the AMM fee, performing `exchange()` back and forth. This is a heavy operation, plus there may be fees if there are other positions before the hacker's ticks. Therefore, this must be done in advance. After that the hacker self-liquidates 99% of their position, leaving one share. If there are no other positions before the hacker, they only spend money on gas.

Step 2. Next, the hacker attacks themselves using an inflation attack. To do this, they perform `create_loan()+repay()` 100 times from another account. Each pass inflates the collateral by 1.5 times.

How it works:

Suppose the tick currently contains 1 wei share and 106 wei collateral. The hacker deposits (106 * 2 - 1) wei collateral into the tick. How many shares will be minted?

```
1 wei share * (106 * 2 - 1) / 106 = 1 wei share
```

Only 1 wei share was minted due to a rounding error.

Now the tick has 2 wei shares and approximately `106 * 3` wei collateral.

What happens when the hacker performs a full repay? They get back approximately

```
106 * 3 / 2 wei
```

The tick remains with approximately the same amount: 1 wei share and `106 * 3 / 2` wei collateral.

It can be seen that one pass of `create_loan()+repay()` does not change the number of shares but increases the collateral by 1.5 times. 100 rounds can inflate the collateral from 106 wei to 42 ETH.

The test script was handed over to the customer.

Recommendation

There are different approaches on how to solve the Inflation Attack problem. Some of the approaches along with their pros and cons, can be found in the OpenZeppelin github issue: https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3706.

One way to resolve the problem is to use virtual dead shares, as implemented in the latest OpenZeppelin ERC-4626 vault:

- ERC4626.sol#L200
- ERC4626.sol#L207

In case this particular fix is chosen, it is recommended to use a virtual offset of 1000 (UniswapV2Pair.sol#L120), as this will make the residual possibility of griefing practically unattainable.

Client's commentary

> Excellent finding. Used the OpenZeppelin method to fix.

2.2 High

H-1	Withdrawal of tokens with debt as zero
Severity	High
Status	Fixed in c5169a7e

Description

This code allows you not to spend debt in liquidation (Controller.vy#L990):

```
debt = unsafe_div(debt * frac, 10**18)
```

If `debt * frac` is less than 10**18, then you don't have to pay for the liquidation part. We especially have the ability to eliminate the entire collateral by passing frac as 1. Example:

```
l_amount = 1

collateral_token._mint_for_testing(user, c_amount)
market_controller.create_loan(c_amount, l_amount, n)
market_controller.liquidate_extended(user, 0, 10 ** 18 - 1,
    True, ZERO_ADDRESS, [])
# d_debt = 0
# xy[0] = 0
# xy[1] 1000
# PROFIT stablecoin.balanceOf(user) +0
# PROFIT collateral_token.balanceOf(user) +1000
```

In this case, the attack is disadvantageous due to the gas.

Recommendation

It is recommended to add an additional check that `debt != 0`.

Client's commentary

> Fixed

H-2	`AMM.exchange()` produces negative fees
Severity	High
Status	Fixed in c5169a7e

Description

Using `exchange()` back-and-forth can generate profit without losing stablecoins. This occurs due to inaccuracies in calculating the invariant when AMM fees are low.

The profit generated from each tick is small (approximately 0.000000000136353050%), but under certain circumstances it can be significant enough for a hacker to execute an attack.

For example, let's assume that there are 100 billion USD spread across 1000 ticks and both the AMM fee and AMM admin fee are equal to zero. In this case, during a single back-and-forth pass, the hacker would earn 0.000000000136353050% from each tick and their total profit from the 100 billion would be approximately 0.136 USD.

If we consider that such a back-and-forth pass would take 833,333 gas, then at the current gas prices on the Ethereum mainnet, such an attack would not be economically feasible.

However, the attack may become more relevant in the future if it is executed on a sidechain with low gas fees. For example, at a gas price of 130 gwei on the Polygon MATIC network (1 MATIC = $0.93), the hacker would spend approximately $0.10 on the attack but earn $0.136. The hacker can repeat the back-and-forth passes in a cycle within a single transaction 1000 times in a cycle and will get a net profit of $36 from a single transaction. By repeating this attack over and over again, they would be able to drain a significant portion of the funds available in the AMM.

The attack works with an AMM fee ranging from 0 to 650,000. Starting from an AMM fee of 1,000,000, the attack fails.

Recommendation

It is recommended to set a limit on the minimum possible AMM fee of no less than 1,000,000.

Client's commentary

> Fee limited

2.3 Medium

M-1	Raw calls on ETH transfers allow reentrancy
Severity	Medium
Status	Fixed in c5169a7e

Description

If WETH is used as collateral, users can choose to receive native ETH when it is sent to users. It happens in function `_withdraw_collateral()`.

- Controller.vy#L514-L521

It breaks Checks-Effects-Interactions pattern. So, a call to an arbitrary address can be made in the middle of functions `repay()` and `_liquidate()`.

- Controller.vy#L731
- Controller.vy#L795
- Controller.vy#L1054
- Controller.vy#L1069

Thus, an attacker can reenter some other smart contract (excluding this Controller). For example, this attacker can call `rate_write()` in `AggMonetaryPolicy`, and the rate will be updated using old `total_debt` (not affected by ongoing `repay` or `liquidate`).

Recommendation

We recommend limiting gas on native ETH transfers.

M-2	Not formalized stablecoin minting to addresses
Severity	Medium
Status	Acknowledged

Description

Admin of `ControllerFactory` can mint any amount of stablecoin to any address calling `set_debt_ceiling`. It is designed to mint tokens to Controllers, but the function does not check that a receiver is among Controllers.

- ControllerFactory.vy#L306-L313

Moreover, this function is used to mint tokens to PegKeepers, and there are no checks that a receiver is among PegKeepers. The whole process is not protected and requires strong admin attention.

Recommendation

We recommend checking that the inputted address in `set_debt_ceiling` is allowed to receive mint stablecoins (is among either Controllers or PegKeepers).

Client's commentary

> That's a good thinking, however we also want to use this to allow to mint for bridged pools (on other chains).

M-3	Early liquidations via `repay()` front-running
Severity	Medium
Status	Fixed in c5169a7e

Description

If a user's position is underwater, a partial `repay()` does not move the borrower's bands but merely reduces the `initial_debt`:

```
else:  # partial repay

if ns[0] > active_band:
    # Not in liquidation - can move bands
    ...

else:
    # Underwater - cannot move band but can avoid a bad liquidation
    ... # do nothing

self.loan[_for] = Loan({initial_debt: debt, rate_mul: rate_mul})
```

- Controller.vy#L777-L781

A hacker can sandwich the victim's transaction:

1. The hacker uses `exchange()` to move the `active_band` forward so the user's position becomes underwater.
2. Then the user performs a partial `repay()` which now will not move the user's bands.
3. The hacker returns their funds (minus fee) using `exchange()` back.

In the above example, if the collateral price goes lower, the user's position will be subject to liquidation much earlier.

Currently, a user cannot protect themselves from such griefing. If the user calls `repay(0)` to move their bands according to the actual debt, nothing will happen:

```
def repay(_d_debt: ...):
...
if _d_debt == 0:
    return
```

- Controller.vy#L731

Recommendation

It is recommended to allow the user to move their bands by calling `repay(0)`.

Client's commentary

> Good point. This is not dangerous because even "liquidation" is not scary at all, and in addition, attacker pays some fees to manipulate.

M-4	`AggregateStablePrice` can be manipulated
Severity	Medium
Status	Acknowledged

Description

If there is not enough liquidity in the pools or there are no pools, then `10**18` is returned as the price in `AggregateStablePrice`.

- AggregateStablePrice.vy#L151

```
if Dsum == 0:
    return 10**18
```

It is supposed to be used to manipulate the price. At an early stage of the project, this can be significant.

Recommendation

We recommend taking these conditions into account when deploying contracts.

Client's commentary

> Absolutely. That's why we fill the pegkeeper pools right away after deploying

2.4 Low

L-1	No checks of the result of `AMM.withdraw`
Severity	Low
Status	Acknowledged

Description

After `withdraw` there is no check that this method did not return any amount.

Controller.vy#L770

There will be either emptiness or dust, but it is worth checking or providing for such a possibility in a comment.

Recommendation

It is recommended that you further check `xy[0]`.

L-2	`AMM.exchange()` has no deadline
Severity	Low
Status	Acknowledged

Description

The `AMM.exchange()` and `AMM.exchange_dy()` functions have slippage checks for output tokens; however, they do not have a deadline check for the transaction:

```
def exchange(i, j, in_amount, min_amount, _for)
...
def exchange_dy(i, j, out_amount, max_amount, _for)
```

- AMM.vy#L1284-L1312

A realistic scenario is possible where, due to high gas prices, an `exchange()` transaction will remain in the mempool for a significant amount of time, and the prices in the AMM may change significantly. This would allow MEV-bots to steal the user's positive slippage (unrealized profit).

Recommendation

It is recommended to add a `deadline` parameter in exchange functions.

Client's commentary

> Deadline are bad practice, in our opinion. Just min _amount is better

L-3	Not compatible with rebase/deflationary/hookable tokens and tokens with fees
Severity	Low
Status	Acknowledged

Description

Smart contracts do not check token balances, and it only works for normal tokens without unexpected balance behavior. In addition, tokens with additional hooks can open additional attack vectors.

Recommendation

This design is acceptable and even has some security advantages - if the project does not have plans to have AMMs with such tokens.

Client's commentary

> Yes - only wrapped

L-4	Key functions do not have return values
Severity	Low
Status	Acknowledged

Description

Many key functions do not have return values but they imply different behavior in case of non-revert.

- Controller.vy#L586
- Controller.vy#L600
- Controller.vy#L671
- Controller.vy#L685
- Controller.vy#L700
- Controller.vy#L731
- Controller.vy#L795
- Controller.vy#L1054
- Controller.vy#L1069

It can add more complexity on integration with other smart contracts.

Recommendation

We recommend returning key values that can explain the results of call executions.

L-5	ControllerFactory has unnecessary calculation operations
Severity	Low
Status	Acknowledged

Description

ControllerFactory stores the number of collaterals as a unit starting from `2**128`.

- ControllerFactory.vy#L215

Then it reads the number by subtracting`2**128`.

- ControllerFactory.vy#L249
- ControllerFactory.vy#L260

Calculations with `2**128` can likely be dropped.

Recommendation

We recommend removing calculations with `2**128`.

Client's commentary

> They cannot be because default value is 0, and indexes also start with 0. To distinguish between 0 and None, we add a constant offset (which could be anything).

L-6	ControllerFactory.set_admin() does not have two-step ownership transferring
Severity	Low
Status	Acknowledged

Description

Admin can transfer ownership to anyone, even if it is a wrong address or zero-value address.

- ControllerFactory.vy#L281-L288

Recommendation

We recommend following a two-step procedure of ownership transferring when a new owner has to accept ownership.

Client's commentary

> That's why it is imperative that only DAO (which can only vote for calls) owns the factory, not an EOA or a multisig

3. ABOUT MIXBYTES

MixBytes is a team of blockchain developers, auditors and analysts keen on decentralized systems. We build opensource solutions, smart contracts and blockchain protocols, perform security audits, work on benchmarking and software testing solutions, do research and tech consultancy.

Contacts

https://github.com/mixbytes/audits_public

https://mixbytes.io/

hello@mixbytes.io

https://twitter.com/mixbytes