Zellic

CrocSwap

Smart Contract Security Assessment

August 24, 2022

Prepared for:

Doug Colkitt

Crocodile Labs

Prepared by:

Ayaz Mammadov and Katerina Belotskaia

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Executive Summary

Zellic conducted an audit for Crocodile Labs from July 25th to August 13th, 2022.

Our general overview of the code is that it was very featured and comprehensive, containing a variety of features that each provided their own mechanisms that had to be considered uniquely. The code was very logic heavy although still very comprehendible.

We applaud Crocodile Labs for their attention to detail and diligence in maintaining incredibly high code quality standards in the development of CrocSwap; this is especially impressive due to the sheer size of the project. Furthermore, Crocodile Labs provided amazing documentation containing images of the plethora of mechanisms implemented.

Zellic thoroughly reviewed the CrocSwap codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

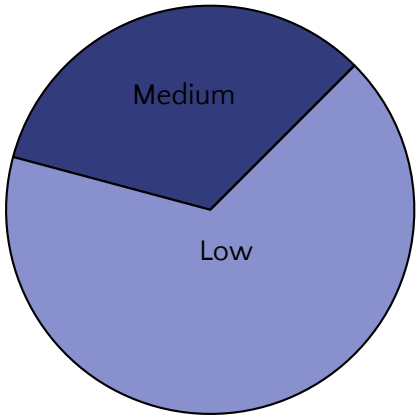
Specifically, taking into account CrocSwap's threat model, we focused heavily on issues that would break core invariants such as keeping the balance of the prices in the ticks, operations that could break the agent masking layer resulting in either false impersonation or double spends, and the credit flow system which should settle flows across a wide variety of calls/operations.

During our assessment on the scoped CrocSwap contracts, we discovered three findings. Fortunately, no critical issues were found. Of the three findings, one was of medium severity, and the remaining findings were of low severity in nature.

Additionally, Zellic recorded its notes and observations from the audit for Crocodile Labs's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	2
Informational	0



2 Introduction

2.1 About CrocSwap

CrocSwap is a decentralized exchange (DEX) protocol that allows for two-sided AMMs combining concentrated and ambient constant-product liquidity on any arbitrary pair of blockchain assets.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of open-source tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these shallow bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so forth as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, and so forth.

Complex integration risks. Several high-profile exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

Code maturity. We review for possible improvements in the codebase in general. We

look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so forth.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their importance may differ. This varies based on numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

CrocSwap Contracts

Repository	https://github.com/CrocSwap
Versions	`ce05a3cdb4d28b4d399d13e2ce9d64fade78f62f`
Programs	• CrocSwapDex
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of five person-weeks. The assessment was conducted over the course of two calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Stephen Tong, Co-founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Ayaz Mammadov, Engineer
ayaz@zellic.io

Katerina Belotskaia, Engineer
kate@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

July 25, 2022	Kick-off call
July 25, 2022	Start of primary review period
August 13, 2022	End of primary review period
TBD	Closing call

3 Detailed Findings

3.1 `initPool` handles fees incorrectly

- **Target**: PoolRegistry.sol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Medium
- **Impact**: Low

Description

The function `initPool` allows to create a new pool for tokens pair from a preset template, which is defined by poolIdx. `registerPool` is called during the pool initialization process and gets a preset template, which is defined by poolIdx and saves new pool in storage. Also, there is the `protocolTake_` value, which is set by the global `protocolTakeRate_` value. But this happens after the new pool is saved in storage, and setting this value will not be applied to the new pool.

```
function registerPool (address base, address quote, uint256 poolIdx)
    internal returns (PoolSpecs.PoolCursor memory, uint128) {
    assertPoolFresh(base, quote, poolIdx);
    PoolSpecs.Pool memory template = queryTemplate(poolIdx);
    PoolSpecs.writePool(pools_, base, quote, poolIdx, template);
    template.protocolTake_ = protocolTakeRate_;
    return (queryPool(base, quote, poolIdx), newPoolLiq_);
    }
```

Impact

All pulls will be created with a zero `protocolTake_` value. And until this value is set by a separate transaction, the protocol fee for the swap will not be charged.

```
// uint128 protoFee is protocolTake_
    function vigOverFlow (uint128 flow, uint16 feeRate, uint8 protoProp)
    private pure returns (uint128 liqFee, uint128 protoFee) {
        unchecked {
            uint256 FEE_BP_MULT = 1000000;
            protoFee = uint128(totalFee * protoProp / 256);
            liqFee = uint128(totalFee) - protoFee;
```

```
        }
    }
```

Recommendations

Should change the order of operations and set the value before saving pool.

```
function registerPool (address base, address quote, uint256 poolIdx)
    internal returns (PoolSpecs.PoolCursor memory, uint128) {
    assertPoolFresh(base, quote, poolIdx);
    PoolSpecs.Pool memory template = queryTemplate(poolIdx);
    template.protocolTake_ = protocolTakeRate_;
    PoolSpecs.writePool(pools_, base, quote, poolIdx, template);
    return (queryPool(base, quote, poolIdx), newPoolLiq_);
    }
```

Remediation

TBD

3.2 The function `safeTransfer` can fail silently

- **Target**: TransferHelper.sol
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The functions `safeTransfer` and `safeTransferFrom` use low-level function `call` for tokens transferring, which return `true` value in case of calling non-existent contract.

```
function safeTransfer( address token, address to, uint256 value)
    internal {
        (bool success, bytes memory data) =
            token.call(abi.encodeWithSelector(IERC20Minimal.transfer.
    selector, to, value));
        require(success && (data.length == 0 || abi.decode(data, (bool)))
    , “TF”);
    }
```

Impact

Since there is no verification of the existence of the contract being called, in the case described above, the transaction will be counted as successful despite the fact that the tokens will not be sent.

Although, when initializing the pool, it is checked that the contract balance has been increased by the expected value of liquidity, which makes unpossible the creation of a pool for non-existent tokens. But they will be checked only in case `newPoolLiq_` was set, otherwise pool will be created without initial liquidity.

Recommendations

Explicitly check the existence of contracts before transferring tokens.

Remediation

TBD

3.3 Limit prices handled incorrectly

- **Target**: TradeMatcher.sol
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

Limit prices are used to stop operations when a certain price is reached; in the main core loop of the function `sweepSwapLiq`, a require statement is used to verify that the current price is appropriate in respect to the limits that have been set. However, due to the implementation of this require, when a user is executing a sell operation in the case that the `current price == limit price`, 1 tick of trade will still go through.

```
function sweepSwapLiq ( ... ) internal {
    require(swap.isBuy_ == (curve.priceRoot_ < swap.limitPrice_), “SD”);
     ...
}
```

Impact

Limit prices are not respected in the proposed circumstances, and the user will trade at a worse price than they wished for.

Recommendations

Use two separate conditions for each type of operation (`isBuy` and `!isBuy`), or if this is the intended behavior, document it.

Remediation

TBD

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 The function `resetNonce` allows to set any value of nonce

The function `userCmdRelayer` allows calling an arbitrary command on behalf of another user who has signed an off-chain transaction. In addition to other values, the user signs a `nonce` value for each approved command to prevent the replay-attack. According to documentation, a `nonce` cannot be evaluated until prior orders at lower nonces have been successfully evaluated. But there is the function `resetNonce` that allows users to set an arbitrary `nonce` value, including a large one from the current one.

The user's ability to jump over the `nonce` value for operations that have already been approved but not executed should be limited.

4.2 It is not possible to approve a transaction for swap if `feeRate_` is zero

The permission oracle should return a positive value if the operation is approved. But if the `feeRate_` value is zero, the transaction will still be canceled due to overflow.

```
function verifyPermitSwap (PoolSpecs.PoolCursor memory
    pool, address base, address quote, bool isBuy, bool inBaseQty,
    uint128 qty) internal {
        if (pool.oracle_ != address(0)) {
            uint16 discount = ICrocPermitOracle(pool.oracle_)
            .checkApprovedForCrocSwap(lockHolder_, msg.sender, base,
    quote, isBuy, inBaseQty, qty, pool.head_.feeRate_);
            require(discount > 0, "Z");
            pool.head_.feeRate_ -= discount;
        }
    }
```

The same for `verifyPermit` function:

```
function verifyPermit ( ... ) internal {
        if (pool.oracle_ != address(0)) {
            uint16 discount = ICrocPermitOracle(pool.oracle_)
                .checkApprovedForCrocPool(lockHolder_, msg.sender, base,
    quote, ambient,
                                          swap, concs, pool.head_.
    feeRate_);
            require(discount > 0, "Z");
            pool.head_.feeRate_ -= discount;
        }
    }
```

The confirmation result and the discount value should be returned explicitly.

4.3 Documentation for `timeUint32`

We just wanted to note that the documentation provided for the function `timeUint32` is incorrect. It was assumed that the function would return the current time until `2038`. However, this would only be true if the function used a `int32`; the function returns a `uint32`, which returns the current time until `2106`.

4.4 Fuzzer Dynamic Analysis

As part of our audit, we implemented several end-to-end fuzzing tests, the code of which was provided to the client. These Echidna tests are simple yet effective.

We have written tests for minting and burning ambient and concentrate liquidity.

In our tests, to maximize coverage, we added multiple effects functions so that the input data is in the allowed range and to avoid a large number of reverted transactions:

- Add a random but valid amount of base and quote ambient liquidity to the pool.
- Remove a random but valid amount of ambient liquidity to the pool.
- Add a random but valid amount of base and quote concentrate liquidity and in a random but valid range of ticks from the pool.
- Remove a random but valid amount of ambient concentrate and in a random but valid range of ticks from the pool.

We focused on checking the invariant of the token balance and liquidity in the pool. Fortunately, we did not find any problems related to abnormal balance and liquidity values.

4.5 Setting up the Fuzzer

1. Clone the CrocSwap Dex repository
2. Add/Overwrite the relevant files in the contract folders:
    - fuzzing.zip/contracts/CrocSwap.sol to repository/contracts
    - fuzzing.zip/contracts/callpath/* to repository/contracts/callpaths
    - fuzzing.zip/contracts/lens/CrocQuery.sol to repository/contracts/lens
    - fuzzing.zip/tests/* to repository/tests
    - fuzzing.zip/config.yaml to repository/
3. In the repository/ folder run this command

```
echidna --config config.yaml --contract tests/tests.sol
```

5 Audit Results

At the time of our audit, the code was not deployed to mainnet evm.

During our audit, we discovered three findings. Of these, one was medium risk and two were low risk. Crocodile Labs acknowledged all findings and implemented fixes.

5.1 Disclaimers

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any additional code added to the assessed project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.