Zellic

Ante

Smart Contract Security Assessment

May 22, 2022

Prepared for:

Ante Labs

Prepared by:

Aaron Esau and Ayaz Mammadov

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 Introduction

1.1 About Ante

Ante is a chain-agnostic, non-custodial, incentivized testing protocol that allows anyone to tokenize trust in target protocols. Ante is building the Schelling point for decentralized trust by providing a common standard for developers to write incentivized on-chain smart contract tests.

Ante uses a smart contract to verify the correctness of market claims by other contracts by hedging stakers and challengers against each other. Ante test pay stakers a portion of challenger funds over time. However, in the case that an Ante test's claim fails, the challengers are paid out all of the staker's funds.

1.2 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, etc. 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, etc.

Complex integration risks. Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its inter-

action 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, etc.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines, or code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, etc.

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 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, project timelines, etc. We aim to provide useful and actionable advice to our partners that consider their long-term goals, rather than simply a list of security issues at present.

1.3 Scope

The engagement involved a review of the following targets:

Ante Contracts

Repository	https://github.com/antefinance/ante-v0-core
Versions	`fdd0d8d68a5697415cde511aa5dc98c469871bb7`
Programs	AntePool, AntePoolFactory
Type	Solidity
Platform	EVM-compatible

1.4 Project Overview

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

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:

Aaron Esau, Engineer
aaron@zellic.io

Ayaz Mammadov, Engineer
ayaz@zellic.io

1.5 Project Timeline

The key dates of the engagement are detailed below.

May 10, 2022	Kick-off call
May 11, 2022	Start of primary review period
May 13, 2022	End of primary review period
May 20, 2022	Closing call

1.6 Disclaimer

This assessment does not provide any warranties on finding all possible issues within its scope; i.e., the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees on any additional code added to the assessed project after our assessment has concluded. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program. Finally, this assessment report should not be considered financial or investment advice.

2 Executive Summary

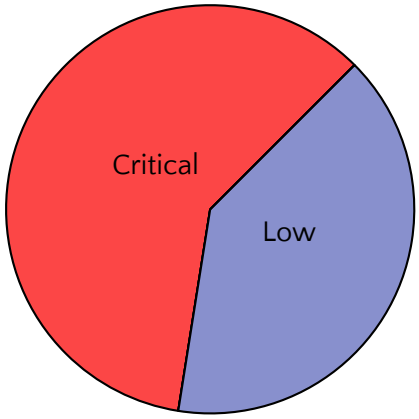
Zellic conducted an audit for Ante Labs from May 11th to May 13th, 2022 on the scoped contracts and discovered 5 findings. Of the 5 findings, 1 was of critical severity, 1 was of high severity, 1 was of medium severity, and the remaining 2 were of low severity.

Zellic thoroughly reviewed the Ante codebase to find protocol-breaking bugs as defined by the documentation, or any technical issues outlined in the Methodology section of this document. Specifically, taking into account Ante's threat model, we focused heavily on bugs resulting in theft or freezing of user funds, incorrect calculation of rewards and decay payments, and false positive triggering of test failure.

Our general overview of the code is that it was well-organized and structured. The code coverage is high and tests are included for the majority of the functions. The documentation was thorough and very high quality. The code was easy to comprehend and very intuitive.

Breakdown of Finding Impacts

Impact Level	Count
Critical	3
High	0
Medium	0
Low	2
Informational	0



3 Detailed Findings

3.1 Ability to force tests to fail with gas limit

- **Target**: AntePool
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Critical
- **Impact**: Critical

Description

It is possible for attackers to force tests to fails by setting the gas limit to a very specific value to where:

- it is low enough that the inner call to `checkTestPasses` runs out of gas, but
- it is high enough that the outer `checkTest`/`checkTestNoRevert` functions finish executing.

This is possible because of a feature in Solidity where try/catch statements revert before the last 1/64th of the transaction gas limit is consumed (Source):

> The caller always retains at least 1/64th of the gas in a call and thus even if the called contract goes out of gas, the caller still has some gas left.

So, if 1/64th of the maximum gas value that causes the test to revert is enough to execute the remainder of `checkTest`, it is possible to force a test to fail.

Zellic wrote a proof of concept exploit to verify the exploitability of this issue.

Impact

An attacker could force certain pools to fail and claim their rewards. Note that as of the time of this writing, no community-written, deployed tests are vulnerable.

Recommendations

It is not currently possible to directly detect an out-of-gas error in a try/catch. Zellic and Ante Labs determined that the best solution is to implement magic return values so that pools can distinguish between a "false" returned by an out-of-gas reversion and a test failure (indicated by returning false or manual reversion).

Remediation

Ante Labs acknowledged this finding and plans to implement a fix—most likely using the magic return value method described in the Recommendations section.

In the meantime, Ante Labs plans to provide analysis tools to community test writers to lower the likelihood of a vulnerable test being deployed. Note that no community-written, deployed tests are vulnerable as of the time of this writing.

3.2 Number of challengers is constrained by block gas limit

- **Target**: AntePool
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: High
- **Impact**: Critical

Description

An attacker can freeze funds for a low cost by causing the `_calculateChallengerEligibility` function to hit the block gas limit. Since the loop iterates over every challenger in storage, if enough challengers are registered, the `checkTest` function will not be callable when the test fails.

Impact

- Front-running bots may be able to claim the majority of rewards by exploiting the block gas limit issue using the following steps:
  1. Upon detecting a failed check, depositing a large amount of capital as challenger.
  2. Locking `checkTest` by registering many challengers.
  3. Twelve blocks later, unlocking `checkTest` by removing them.
  4. Calling `checkTest` to claim rewards and 5% bounty.
- Stakers could prevent `checkTest` from running until their funds are unstaked after realizing a test is going to fail.
- An attacker could perform griefing attacks to prevent payouts from failed checks.

Note that this vulnerability can be chained with the `MIN_CHALLENGER_STAKE` bypass vulnerability to significantly lower the attack cost. We determined that in practice, exploiting these two vulnerabilities together to lock funds would cost approximately $60,000 USD due to block gas as of the time of this writing. An attack is especially likely if the profit of delaying `checkTest` exceeds the cost of the attack.

Recommendations

We recommend dynamically calculating the `MIN_CHALLENGER_STAKE` so that it is economically impractical to perform this attack.

For recommendations on mitigating the minimum challenger stake bypass vulnerability, see the finding in section 3.3.

Remediation

Ante Labs acknowledged this finding and implemented a fix in commit `a9490290d23191d2bbcc2acfce5c901aed1bb5d2`.

3.3 Bypassable minimum challenger stake

- **Target**: AntePool
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Low
- **Impact**: Low

Description

It is possible to bypass the following check in the `stake` function. This would allow malicious challengers to stake less than the minimum of `MIN_CHALLENGER_STAKE` (default `1e16` or `0.01 ether`) ether:

```
require(amount ≥ MIN_CHALLENGER_STAKE, “ANTE: Challenger must stake more
    than 0.01 ETH”);
```

To bypass the `MIN_CHALLENGER_STAKE`, challengers can

1. Call the `stake` function to stake `MIN_CHALLENGER_STAKE`
2. In the same transaction, call the `unstake` (internally `_unstake`) function to unstake `MIN_CHALLENGER_STAKE - 1`

Now, the challenger is still registered while only costing 1 base unit (`0.00000001 ether`) and block gas fees.

Impact

Front-running bots could register a challenger on every test for a very low cost to steal the 5% bounty when a test fails.

Recommendations

If challengers wish to withdraw enough challenger stake that their total staked amount becomes less than `MIN_CHALLENGER_STAKE`, require that *all* of their stake be removed:

```
function _unstake(
    uint256 amount,
    bool isChallenger,
    PoolSideInfo storage side,
    UserInfo storage user
) internal {
```

```
    // Calculate how much the user has available to unstake, including the
    // effects of any previously accrued decay.
    // prevAmount = startAmount * decayMultiplier / startDecayMultiplier
    uint256 prevAmount = _storedBalance(user, side);

    if (prevAmount == amount) {
        user.startAmount = 0;
        user.startDecayMultiplier = 0;
        side.numUsers = side.numUsers.sub(1);

        // Remove from set of existing challengers
        if (isChallenger) challengers.remove(msg.sender);
    } else {
        require(amount ≤ prevAmount, “ANTE: Withdraw request exceeds
    balance.”);
        require(!isChallenger
                || prevAmount.sub(amount) > MIN_CHALLENGER_STAKE,
                “ANTE: must withdraw at least MIN_CHALLENGER_STAKE”);
        user.startAmount = prevAmount.sub(amount);
        // Reset the startDecayMultiplier for this user, since we've
    updated
        // the startAmount to include any already-accrued decay.
        user.startDecayMultiplier = side.decayMultiplier;
    }
    side.totalAmount = side.totalAmount.sub(amount);

    emit Unstake(msg.sender, amount, isChallenger);
}
```

For recommendations on mitigating the maximum challengers limit due to block gas limit vulnerability, see the finding in section 3.2.

Remediation

Ante Labs acknowledged this finding and implemented a fix in commit `8e4db312c7046db3f76146080f166baeab025acb`.

3.4 Reentrant `checkTest` allows pool draining

- **Target**: AntePool
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Critical

Description

Because `checkTest` allows reentrancy, in specific cases, an attacker may be able to drain AntePool by

1. Calling `checkTest` on a test that returns `false` (not one that reverts). The test must be written in a way that causes the contract to make an external call to an attacker contract. The attacker contract repeats step 1 as many times as desired.
2. After entering the if condition, the `_verifier` is changed to the current caller.
3. The current caller calls `claim` after `checkTest` returns.

Steps 2–3 repeat for each reentrant call to `checkTest`, causing the 5% bounty to be claimed multiple times.

For this to be exploitable, a test must

- be able to return `false` without reverting.
- not have a `checkTestPasses` function that is `view` or `pure`.
- call a function on the tested contract that internally makes an external call (e.g. to `fallback` or `receive`) to an attacker-controlled contract, for whatever reason.

Impact

If a test fails on a contract matching certain requirements, an attacker could drain the majority of the pool by repeatedly changing the verifier and claiming bounties.

Recommendations

We recommend using the `nonReentrant` modifier or otherwise preventing the `checkTest` function from allowing reentrancy.

Remediation

Ante Labs acknowledged this finding and implemented a fix in commit `8448a63d3c7f7303e35cfc63807cdad540d3aa85`.

3.5 Late challengers receive no payout

- **Target**: AntePool
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

Challengers who stake funds within the 12-block delay period before a check fails do not receive any payout because the `stake` function only tracks the block number of the most recent staking:

```
function stake(bool isChallenger) external payable override
    testNotFailed {
     ...
    if (isChallenger) {
         ...

        // Record challenger info for future use
        // Challengers are not eligible for rewards if challenging within
    12 block window of test failure
        challengers.insert(msg.sender);
        eligibilityInfo.lastStakedBlock[msg.sender] = block.number;
    } else {
        side = stakingInfo;
    }

     ...
}
```

Impact

If a long-time challenger happens to stake additional funds within the last 12 blocks before a test fails, they will receive no payout—even for their older stake.

Recommendations

Fix the logic to appropriately pay challengers based on when they chose to stake. The purpose of the 12-block window is to make front running more difficult. Only new challengers should be penalized for challenging within the 12-block window.

Remediation

Ante Labs noted that the security benefits of having the 12-block period strongly outweigh the risk presented by this issue.

4 Discussion

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

4.1 Ante test vulnerabilities

An interesting fact about Ante we wanted to note is that tests themselves may be vulnerable. Tests must be written with care to ensure they do not fail (revert or return `false`) for any reason other than the tested invariants failing.

4.2 Known issues

As of the time of this writing, Ante lists the following known issues on their documentation website:

- Challenger decay calculation is inaccurate and slightly overestimates the decay paid by challengers (overall error is < 1%/year even in the worst case scenario). Calculation is more accurate the more often `updateDecay()` is called.
- Staker and challenger balances are slightly underestimated due to rounding issues in intermediate calculations; overall loss is small relative to total pool balance flux (< 0.1%).
- Test verification can be front run by challengers who stake small amounts of ether in every pool.
- `checkTest` gas usage can be unbounded as it scales linearly with number of unique challengers.