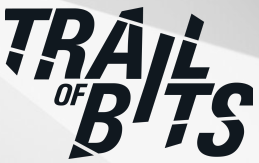

Paraspace

Security Assessment

December 7, 2022

Prepared for:
Cheng Jiang
Ivan Solomonoff
Paraspace

Prepared by: **Tjaden Hess**

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 100+ 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

Paraspace engaged Trail of Bits to review the security of its decentralized lending protocol. From November 28 to December 2, 2022, one consultant conducted a security review of the client-provided source code, with five person-days of effort. This review is intended to cover changes that were made to the Paraspace source code after the culmination of our initial review of the protocol, which began on October 24, 2022, and took seven person-weeks of effort. Details of the timeline, test targets, and coverage of this additional week of review 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 system, with access to both the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes. The scope of this review included the changes made to the Paraspace source code, especially changes to BAYC/APE staking and liquidation logic.

Summary of Findings

The audit uncovered minor code quality issues and gas inefficiencies. While no significant flaws were uncovered, the high complexity of the Paraspace codebase warrants significant caution and more thorough negative unit testing. A summary of the findings is provided below.

EXPOSURE ANALYSIS

Severity	Count
High	0
Medium	0
Low	0
Informational	2
Undetermined	0

CATEGORY BREAKDOWN

Category	Count
Configuration	1
Testing	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineer was associated with this project:

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Project Timeline

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

Date	Event
December 2, 2022	Delivery of report draft
December 5, 2022	Final report readout meeting
December 7, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide an extension to a previously completed security review, focusing on new changes to the codebase. In particular, we sought to answer the following questions:

- Could staked ApeCoins be removed in such a way that causes users to become undercollateralized?
- Are access controls in place to prevent unauthorized users from unstaking ApeCoins or taking other users' tokens?
- Does Paraspace's test corpus ensure that access control and collateralization checks remain in place even during active development?
- Are the current reentrancy protections sufficient to prevent attacks?
- Could NFTs be liquidated at unfair prices? Could users liquidate NFTs without going through the Dutch auction procedure?

Project Targets

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

“Para-Space NFT Money Market”

Repository	https://github.com/para-space/paraspace-core
Versions	dad2e84a668a3c0a772c72ae8e01fb5015d48589
Type	Solidity
Platform	Ethereum

Project Coverage

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

- Manual analysis of the source code, documentation, and test cases
- Static analysis of the source code with Slither and triaging of the results

This audit focused primarily on the APE staking and NFT liquidation functionality. In particular, the following contracts were reviewed:

- `NTokenBAYC`
- `NTokenMAYC`
- `NTokenApeStaking`
- `ApeStakingLogic`
- `PoolApeStaking`
- `PTokenSApe`
- `LiquidationLogic`

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- The upgradeability and safety of the pool proxy contract
- Opportunities to manipulate prices
- Rebasing tokens
- Interactions with NFT marketplaces

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	We found no issues related to arithmetic.	**Satisfactory**
Auditing	The reviewed logic emits appropriate events when transfers and administrative actions are executed.	**Satisfactory**
Authentication / Access Controls	We found no issues in the reviewed components related to authentication and access controls.	**Satisfactory**
Complexity Management	Validation assertions are spread throughout the code, and unit tests of some of these assertions are missing, as described in TOB-PARASPACE-1. This makes understanding and validating interactions between components difficult.	**Moderate**
Cryptography and Key Management	We found no issues with cryptography and key management.	**Satisfactory**
Decentralization	The code features several “administrative” roles (e.g., “pool admin,” “emergency admin,” and “risk admin”). The contracts are upgradeable via a proxy mechanism, which allows the Paraspace team to halt or change the behavior of the contracts at any time. Centralized off-chain price oracles are used; a compromised oracle could allow attackers to drain funds by taking out undercollateralized loans.	**Weak**

Documentation	The project has reasonable documentation describing its goals and philosophy. However, the project would benefit from additional documentation describing its internals. For example, the constant indicated in TOB-PARASPACE-2 does not correspond to any publicly documented value.	**Moderate**
Front-Running Resistance	We found no issues related to front-running.	**Satisfactory**
Low-Level Manipulation	We found no issues related to low-level manipulation.	**Satisfactory**
Testing and Verification	Test coverage is generally high, but some critical functionality has no corresponding negative testing, as described in TOB-PARASPACE-1.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Missing negative tests for several assertions	Testing	**Informational**
2	Use of a magic constant with unclear meaning for the sAPE unstaking incentive	Configuration	**Informational**

Detailed Findings

1. Missing negative tests for several assertions	
Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-PARASPACE-1
Target: `test-suites`	

Description

The Paraspace protocol consists of numerous interacting components, and each operation is validated by checks that are widely dispersed throughout the codebase. Therefore, a robust suite of negative test cases is necessary to prevent vulnerabilities from being introduced if developers unwittingly remove or alter checks during development.

However, a number of checks are present in the codebase without corresponding test cases. For example, the health factor check in the `FlashClaimLogic` contract is required in order to prevent users from extracting collateralized value from NFTs during flash claims, but there is no unit test to ensure this behavior. Commenting out the lines in figure 1.1 does not cause any test to fail.

```
86    require(
87       healthFactor > DataTypes.HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
88       Errors.HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD
89    );
```

Figure 1.1: FlashClaimLogic.sol#86–89

A test that captures the desired behavior could, for example, initiate a flash claim of a BAYC NFT that is tied to collateralized staked APE (sAPE) and then withdraw the APE directly from the `ApeCoinStaking` contract, causing the account's health factor to fall below 1.

As another example, removing the following lines from the `withdrawApeCoin` function in the `PoolApeStaking` contract demonstrates that no negative test validates this function's logic.

```
73    require(
74       nToken.ownerOf(_nfts[index].tokenId) == msg.sender,
75       Errors.NOT_THE_OWNER
76    );
```

Figure 1.2: PoolApeStaking.sol#73–76

Exploit Scenario

Alice, a Paraspace developer, refactors the `FlashClaimLogic` contract and mistakenly omits the health factor check. Expecting the test suite to catch such errors, she commits the code, and the new version of the Paraspace contracts becomes vulnerable to undercollateralization attacks.

Recommendations

Short term, for each `require` statement in the codebase, ensure that at least one unit test fails when the assertion is removed.

Long term, consider requiring that Paraspace developers ensure a minimum amount of unit test code coverage when they submit new pull requests to the Paraspace contracts, and that they provide justification for uncovered conditions.

2. Use of a magic constant with unclear meaning for the sAPE unstaking incentive

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-PARASPACE-2
Target: `contracts/protocol/tokenization/NTokenApeStaking.sol`	

Description

If a Paraspace user has a low health factor and is eligible for liquidation, other users may forcibly unstake any sAPE owned by the undercollateralized user and claim a fraction of the unstaked coins as a reward. The percentage of unstaked claims that is awarded is defined inline without a comment, as shown in figure 2.1:

```
130    function initializeStakingData() internal {
131       ApeStakingLogic.APEStakingParameter
132           storage dataStorage = apeStakingDataStorage();
133       ApeStakingLogic.executeSetUnstakeApeIncentive(dataStorage, 30);
134    }
```

Figure 2.1: NTokenApeStaking.sol#130–134

Because this value is not marked explicitly as a constant in a prominent location and does not have a comment explaining its meaning, users may misunderstand its nature.

Exploit Scenario

Bob, a Paraspace user, reads the `NTokenApeStaking` contract and sees that the unstaking incentive is set to 30. He misinterprets the value, believing it represents a percentage rather than 1/100th of a percent, and spends undue resources liquidating small sAPE positions.

Recommendations

Short term, move the constant to the top of the file and add an explanatory comment.

Long term, add user-facing documentation detailing all current incentive rates and similar parameters.

Summary of Recommendations

The Paraspace decentralized lending protocol is a work in progress with multiple planned iterations. Trail of Bits recommends that Paraspace address the findings detailed in this report and take the following additional steps prior to deployment:

- Adopt a testing policy that ensures full code coverage, including negative testing coverage of all `require` statements, in order to prevent the introduction of new bugs (TOB-PARASPACE-1).
- Expand the project's documentation to include descriptions of the project's internals and to comprehensively document protocol parameters (TOB-PARASPACE-2).

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. Non-Security-Related Findings

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- The `DefaultReserveInterestRateStrategy` contract defines both a `borrowUsageRatio` and a `supplyUsageRatio` variable. These variables contain the same value, so they could be consolidated:

```
vars.borrowUsageRatio = vars.totalDebt.rayDiv(

    vars.availableLiquidityPlusDebt

);

vars.supplyUsageRatio = vars.totalDebt.rayDiv(

    vars.availableLiquidityPlusDebt

);
```

- In `WETHGateway`, `withdrawETHWithPermit` interprets a value of `-1` as an indication that it should withdraw the user's full balance. However, when the value is not `-1`, the contract still makes an unnecessary external call to fetch the user's balance. Consider moving the balance call under the conditional to prevent this unnecessary external call.

```
uint256 userBalance = pWETH.balanceOf(msg.sender);

uint256 amountToWithdraw = amount;

// if amount is equal to uint(-1), the user wants to redeem everything

if (amount == type(uint256).max) {

    amountToWithdraw = userBalance;

}
```

- The following comment describing the purposes of the bits in `ReserveConfigurationMap.data` is incorrect:

```
struct ReserveConfigurationMap {
    ...
    //bit 152-167 liquidation protocol fee
    //bit 168-175 eMode category
    //bit 176-211 unbacked mint cap in whole tokens, unbackedMintCap == 0 =>
minting disabled
```

```
    //bit 212-251 debt ceiling for isolation mode with
(ReserveConfiguration::DEBT_CEILING_DECIMALS) decimals
    //bit 252-255 unused

    uint256 data;
}
```

As can be seen from the following definitions, the four bits starting at position 168 hold the asset type:

```
uint256 internal constant ASSET_TYPE_START_BIT_POSITION = 168;
uint256 internal constant IS_DYNAMIC_CONFIGS_START_BIT_POSITION = 172;
```

- The following comment in `LiquidationLogic.sol` is incorrect:

```
* @notice Function to liquidate an ERC721 of a position if its Health Factor
drops below 1. The caller (liquidator)
* covers `liquidationAmount` amount of debt of the user getting liquidated, and
receives
* a proportional tokenId of the `collateralAsset` minus a bonus to cover market
risk
```

The comment should say something like the following:

```
* @notice Function to liquidate an ERC721 of a position if its Health Factor
drops below 1. The caller (liquidator)
* covers `liquidationAmount` amount of debt of the user getting liquidated
minus a bonus to cover market risk, and
* receives the tokenId of the `collateralAsset`
```

- It appears that `IncentivesController` in the Aave codebase was renamed to `RewardController` in the Paraspace codebase. However, the renaming appears to be incomplete, as shown in the following example:

```
/**
 * @notice Returns the address of the Incentives Controller contract
 * @return The address of the Incentives Controller
 **/
function getIncentivesController()
    external
    view
    virtual
    returns (IRewardController)
{
    return _rewardController;
}
```

Also, note that both `RewardController` (singular) and `RewardsController` (plural) are used in the codebase (though the former seems to be used more).

- PTokens allow self-liquidation, but NTokens do not:

```
require(
    params.liquidator != params.borrower,
    Errors.LIQUIDATOR_CAN_NOT_BE_SELF
);
```

We could find no vulnerabilities associated with allowing self-liquidation. Nonetheless, Paraspace should consider disabling self-liquidation for PTokens for consistency.