Staked Robo-Advisor for Yield

Security Assessment

August 30, 2019

Prepared For:
Tim Ogilvie | *Staked*
togilvie@staked.us

Prepared By:
Eric Rafaloff | *Trail of Bits*
eric.rafaloff@trailofbits.com

Michael Colburn | *Trail of Bits*
michael.colburn@trailofbits.com

Executive Summary

From July 29 through August 9 2019, Staked engaged with Trail of Bits to review the security of their Robo-Advisor for Yield (RAY) smart contracts. Trail of Bits conducted this assessment over the course of four person-weeks with two engineers working from commit 127eb38 of the RAY repository.

Trail of Bits completed the assessment using manual, static, and dynamic analysis techniques. The first week focused on understanding RAY at a high level through code and documentation review, checking for common Solidity flaws and missing best practices. Trail of Bits also reviewed proposed fixes for TOB-RAY-01 and TOB-RAY-02, and began to address several design-related concerns that Staked had related to rounding, gas consumption of loops, and contract complexity (e.g., heavy use of settings).

The second week focused on continuing these efforts and digging deeper into the codebase, with the goal of uncovering more in-depth security issues. At Staked's request, Trail of Bits continued to help review fixes on a best-effort basis.

The assessment identified 17 issues in the RAY smart contracts ranging in severity from high to informational. Two such issues, TOB-RAY-01 and TOB-RAY-10, could lead to an attacker forcing a victim into performing undesired actions (e.g., making a repeated purchase of tokens). Though not directly part of RAY, a relevant design flaw of the ERC20 standard enabling token theft was also identified. Other issues include missing security controls, such as data validation, access controls, event logging, patching, and secure configuration. Broader issues related to code quality are detailed in Appendix B. Design-related issues Staked requested guidance on are discussed in Appendix C.

Going forward, it is recommended that the missing security controls identified in this report be applied consistently throughout the codebase, and that the necessary changes be made to address the code quality and design-related issues discussed in Appendix B and Appendix C. In addition, further property testing is recommended using tools such as Echidna and Manticore, which excel at verifying the correctness of specific functionality rather than identifying common vulnerabilities. Appendix D and Appendix E contain example scripts for testing the RAY codebase using Echidna and Manticore, respectively.

Appendix F and Appendix G contain follow-up guidance requested by Staked.

Appendix H contains a fix log for a limited, four-hour follow-up assessment that Staked asked Trail of Bits to perform on August 30, 2019.

Project Dashboard

Application Summary

Name	Staked Robo-Advisor for Yield (RAY)
Version	Git commit 127eb38
Type	Ethereum Smart Contract
Platforms	Solidity

Engagement Summary

Dates	July 29, 2019 - August 9, 2019
Method	Whitebox
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	1	■
Total Medium-Severity Issues	2	■■
Total Low-Severity Issues	9	■■■■■■■■■
Total Informational-Severity Issues	4	■■■■
Total Undetermined-Severity Issues	1	■
Total	17	

Category Breakdown

Cryptography	2	■■
Undefined Behavior	1	■
Access Controls	3	■■■
Configuration	2	■■
Patching	2	■■
Auditing and Logging	1	■
Timing	2	■■

Data Validation	2	■■
Numerics	2	■■
Denial of Service	2	■■
Total	19	

Engagement Goals

The goal of the engagement was to evaluate the security of Staked's Robo-Advisor for Yield (RAY) smart contracts. Aside from checking for common Solidity flaws and missing best practices, Trail of Bits investigated the following questions:

- Is it possible for an attacker to steal or trap tokens?
- Is administration functionality secure and reliable?
- Do RAY's dependencies on other contracts pose a security risk?
- How does rounding of opportunity tokens affect the integrity of price per shares?
- Is it possible that the "for" loops used during user withdrawals will cause transactions to run out of gas?
- What is the safest way to manage RAY's reliance on complex settings?

Coverage

This review included all of the Staked-controlled contracts which make up the RAY system. The specific commit hash of the codebase used for the assessment was `127eb385998c2927c75a3fdcc1553d4c4a4e0b8c` of the `init-repo` branch in the Stakedllc `ray-smart-contracts` Github repository.

The following were areas were prioritized for review based on observations made by Trail of Bits and discussions with Staked:

Arithmetic. RAY makes extensive use of rounding and other mathematical operations. Trail of Bits reviewed arithmetic for potential issues such as overflows and rounding errors, and makes several recommendations in this report regarding the use of safer arithmetic.

Access Controls. Many parts of the system expose privileged functionality, such as adding support for new assets or pausing the contracts. These functions were reviewed to ensure they can be used only by the intended actors and have the desired effect.

Cryptography. RAY supports Staked transmitting signed transactions on behalf of users so that users are not responsible for paying gas. Trail of Bits reviewed this signature handling code for issues such as malleability and replay protection.

In addition to manual analysis, all smart contracts were evaluated using the Slither static analyzer with custom detectors. The WETH9 contract was tested for desired ERC20 security properties using Echidna. Due to time restrictions, Trail of Bits prioritized a holistic review of RAY rather than formally testing individual properties. To supplement this, further property testing is recommended using tools such as Echidna and Manticore.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Secure cryptographic signatures against replay attacks.** When using cryptographic signatures, a nonce (one-time use) token should always be included in signed messages, and properly validated along with its associated signature. See TOB-RAY-01 and TOB-RAY-02 for additional information.

❑ **Consistently enforce access control in the form of function modifiers.** Add the `notPaused` modifier to each function identified in TOB-RAY-03.

❑ **Ensure that Ether cannot be locked accidentally in a contract.** Remove all of the unnecessary fallback functions identified in TOB-RAY-04.

❑ **Remove hard-coded credentials from the codebase.** Replace the Infura API keys referenced in TOB-RAY-05 with new ones and securely store them outside of the codebase.

❑ **Avoid using outdated, insecure versions of the Solidity compiler.** Update the pragma declarations listed in TOB-RAY-06 so that they use the same version as is used in the rest of the codebase.

❑ **Avoid using Solidity compiler optimizations when possible.** Measure gas savings from optimizations, and carefully weigh them against the possibility of optimization-related bugs. See TOB-RAY-07 for additional information.

❑ **Avoid using experimental, insecure features of the Solidity compiler when possible.** Remove the use of `ABIEncoderV2`. See TOB-RAY-08 for additional information.

❑ **Ensure that all sensitive functionality has adequate event logging.** See TOB-RAY-09 for additional information.

❑ **Consistently enforce authorization checks.** Consider complex forms of user interaction, such as the one described in TOB-RAY-10 that spans multiple transactions.

❑ **Perform consistent input validation.** Update all contract constructors to perform adequate input validation, and consistently check the return values of called functions to ensure that operations were successful, as described in TOB-RAY-12.

❑ **Mitigate rounding errors using banker's rounding.** Use the "round half to even" strategy, also called "banker's rounding," as described in TOB-RAY-13.

❑ **Avoid unsafe arithmetic.** Use the SafeMath library to avoid issues such as integer overflows, as recommended in TOB-RAY-14.

❑ **Ensure a loss recovery plan is well documented and tested.** In the event that a capital loss occurs, the fund should have a predefined way to recover. See TOB-RAY-15 for additional information.

❑ **Remove the potential reentrancy in `Entrypoint.purchaseSPRToken`** and be aware that calls to `verifyValue` may result in unintentionally violating the checks-effects-interactions pattern. See TOB-RAY-16 for additional information.

❑ **Investigate the gas usage of paths leading to loops containing external calls,** given the current set of supported assets. Begin researching potential architectural alternatives to mitigate TOB-RAY-17.

Long Term

❑ **Ensure that security libraries are up-to-date.** Use the latest version of OpenZeppelin's ECDSA library, which allows signatures to be safely used as nonces, as described in TOB-RAY-01.

❑ **Add additional testing for security-related functionality.** For examples, see TOB-RAY-02, TOB-RAY-03, TOB-RAY-04, and TOB-RAY-12.

❑ **Follow service-provider best practices.** Configure Infura projects to require project secrets and request whitelisting, as described in TOB-RAY-05 and Infura's documentation. Project secrets should also be stored securely outside of the codebase.

❑ **Secure the build environment by removing outdated software.** As described in TOB-RAY-06, remove older versions of the Solidity compiler from the build environment so that it is not possible to accidentally compile the contracts using the wrong version.

❑ **Stay up to date on security changes affecting RAY's software stack.** For example, monitor the development and adoption of Solidity compiler optimizations to assess their maturity. This is related to TOB-RAY-07.

❑ **Integrate static analysis tools like Slither into the CI pipeline.** This will aid in the detection of issues such as the ones described in TOB-RAY-08, TOB-RAY-14, TOB-RAY-16, TOB-RAY-17.

❑ **Use a blockchain monitoring system to track any suspicious behavior in the contracts.** When proper event logging is performed, monitoring can help detect issues such as those described in TOB-RAY-09.

❑ **Eliminate multi-transaction functionality wherever possible.** This will help prevent race conditions, such as the one identified in TOB-RAY-10.

❑ **Consider what risks are acceptable for RAY.** For example, consider whether depending on the WETH9 token, or any other ERC20 token, is an acceptable risk. See TOB-RAY-11 for additional information.

❑ **Eliminate error-prone operations such as rounding wherever possible.** This will aid in mitigating the rounding issue described in TOB-RAY-13.

❑ **Explore automated loss recovery.** As part of remediating TOB-RAY-15, consider automated solutions that would detect a loss and trigger a recovery process. For example, new shares could be minted and sold automatically.

❑ **Wherever possible, favor incremental pull- over push-based logic.** This will help to reduce the chance of introducing a denial of service as described in TOB-RAY-17.

Findings Summary

#	Title	Type	Severity
1	Pre-signed transactions can be replayed	Cryptography	High
2	Contract does not properly handle invalid signatures	Cryptography	Informational
3	Functions are missing notPaused modifiers	Access Controls	Low
4	Ether can be accidentally locked in contracts	Access Controls	Low
5	Hardcoded Infura API keys	Configuration	Low
6	Solidity compiler is outdated	Patching	Low
7	Solidity compiler optimizations can be dangerous	Configuration	Low
8	ABIEncoderV2 is Used	Patching and Undefined Behavior	Undetermined
9	Missing event logging	Auditing and Logging	Low
10	Race condition affecting purchase of SPR tokens	Timing	Medium
11	Dependence on insecure WETH9	Timing and Data Validation	Medium
12	Missing validation	Data Validation	Low
13	Rounding strategy may lead to errors	Numerics	Low

14	Use of unsafe arithmetic	Numerics	Informational
15	Missing loss recovery	Denial of Service	Informational
16	Reentrancy from trusted third-party contracts	Access Controls	Low
17	External calls in loops may result in denial of service	Denial of Service	Low

1. Pre-signed transactions can be replayed

Severity: High
Difficulty: Low
Type: Cryptography
Finding ID: TOB-RAY-01
Target: `/contracts/protocol/impl/secondary/Payer.sol`

Description
The Payer contract allows users to submit pre-signed transactions in which the associated gas costs are paid for by Staked. However, the contract does not prevent a malicious user from observing a pre-signed transaction and replaying it before it expires. This could be exploited by an attacker in order to force a user into performing an intended action more than once.

The pre-signed transaction validation routines that are implemented by the contract are shown in Figure 1.1. Note that although the provided signature, expiration time, and value are validated, there is no nonce that would prevent a replay attack.

It should also be noted that, due to signature malleability, signatures should not be used as nonces when using `ecrecover`. Older versions of OpenZeppelin's ECDSA library, such as the one used by RAY, remain affected by this issue.

```
   bytes32 msgHash = keccak256(abi.encodePacked(
      portfolioKey,
      beneficiary,
      value,
      upgrades,
      funder,
      expiryTimeInSeconds,
      salt
    ));

    msgHash = msgHash.toEthSignedMessageHash();

    // find the messages signer based on the message hash and signature
    address signer = msgHash.recover(signature);

    // require the signer of the transaction is the one paying for it
    require(signer == funder, "#Payer verifyPurchase(): The signer does not equal the
funder");
```
Figure 1.1 - Lines 250-266 of `/contracts/protocol/impl/secondary/Payer.sol`

```
   bytes32 msgHash = keccak256(abi.encodePacked(
      tokenId,
      value,
      funder,
      expiryTimeInSeconds,
      salt
    ));
    msgHash = msgHash.toEthSignedMessageHash();

    // find the messages signer based on the message hash and signature
    address signer = msgHash.recover(signature);

    // require the signer of the transaction is the one paying for it
    require(signer == funder, "#Payer verifyAddCapital(): The signer does not equal the
funder/beneficiary");
```

Figure 1.2 - Lines 285-299 of `/contracts/protocol/impl/secondary/Payer.sol`

Exploit Scenario
An attacker replays a pre-signed transaction multiple times before it expires, thereby forcing the original sender into repeatedly withdrawing a position or purchasing tokens.

Recommendation
Short term, a nonce (one-time use) token should be included in signed messages, and validated along with its associated signature.

Long term, use the latest version of OpenZeppelin's ECDSA library, which allows signatures to be used as nonces safely.

References
- OpenZeppelin's ECDSA library

2. Contract does not properly handle invalid signatures

Severity: Informational
Difficulty: Low
Type: Cryptography
Finding ID: TOB-RAY-02
Target: `/contracts/protocol/impl/secondary/Payer.sol`

Description
The Payer contract validates that pre-signed transactions contain the same funder and signer address. However, the contract does not detect invalid signatures by checking the return value of `msgHash.recover`, which returns all zeros upon failure. It is therefore possible for an attacker to choose a funder address of all zeros and pass the signature check, despite the fact that this is an invalid address.

```
   bytes32 msgHash = keccak256(abi.encodePacked(
      portfolioKey,
      beneficiary,
      value,
      upgrades,
      funder,
      expiryTimeInSeconds,
      salt
    ));

    msgHash = msgHash.toEthSignedMessageHash();

    // find the messages signer based on the message hash and signature
    address signer = msgHash.recover(signature);

    // require the signer of the transaction is the one paying for it
    require(signer == funder, "#Payer verifyPurchase(): The signer does not equal the
funder");
```

Figure 2.1 - Lines 250-266 of `/contracts/protocol/impl/secondary/Payer.sol`

```
   bytes32 msgHash = keccak256(abi.encodePacked(
      tokenId,
      value,
      funder,
      expiryTimeInSeconds,
      salt
    ));

    msgHash = msgHash.toEthSignedMessageHash();

    // find the messages signer based on the message hash and signature
    address signer = msgHash.recover(signature);

    // require the signer of the transaction is the one paying for it
    require(signer == funder, "#Payer verifyAddCapital(): The signer does not equal the
funder/beneficiary");
```

Figure 2.2 - Lines 285-299 of `/contracts/protocol/impl/secondary/Payer.sol`

Exploit Scenario
An attacker exploits this lack of validation to put the contract into an invalid state.

Because no such circumstances were identified in the current codebase, this issue is considered informational.

Recommendation
Short term, validate the return values of all calls to `recover`, and reject errors as indicated by an address of all zeros.

Long term, add additional tests to the codebase that validate the proper handling of invalid signatures.

3. Functions are missing notPaused modifiers

Severity: Low
Difficulty: Low
Type: Access Controls
Finding ID: TOB-RAY-03
Target: Multiple contract functions

Description
The `notPaused` modifier, which is responsible for checking if a function has been “paused” and cannot be interacted with, is missing from the public `buyERC20Position` functions in the following files:

- `/contracts/protocol/impl/opportunities/BzxImpl.sol`
- `/contracts/protocol/impl/opportunities/CompoundImpl.sol`
- `/contracts/protocol/impl/opportunities/DydxImpl.sol`

Exploit Scenario
An attacker exploits the missing `notPaused` modifiers to interact with sensitive functionality even after it is paused.

Recommendation
Short term, add the `notPaused` modifier to each identified function.

Long term, add additional tests to the codebase which validate that paused functions cannot be interacted with unintentionally.

4. Ether can be accidentally locked in contracts

Severity: Low
Type: Access Controls
Target: Multiple contracts

Difficulty: High
Finding ID: TOB-RAY-04

Description
Several contracts have empty fallback functions. If a user accidentally sends Ether to one of these contracts, it will be locked up forever with no way for the sender to get it back.

```
function() public payable {

}
```

Figure 4.1 - Lines 115-117 of `/contracts/protocol/impl/Oracle.sol`

The following is a list of all affected contracts:

- `/contracts/protocol/impl/secondary/Entrypoint.sol`
- `/contracts/protocol/impl/opportunities/DydxImpl.sol`
- `/contracts/protocol/impl/opportunities/CompoundImpl.sol`
- `/contracts/protocol/impl/opportunities/BzxImpl.sol`
- `/contracts/protocol/impl/Router.sol`
- `/contracts/protocol/impl/Oracle.sol`
- `/contracts/protocol/NCController.sol`

Exploit Scenario
A user accidentally sends Ether to one of the affected contracts, thereby locking their Ether up forever with no way to get it back.

Recommendation
Short term, remove all unnecessary fallback functions. For instances where contracts need to send Ether to each other, enforce address whitelisting to prevent users from accidentally sending Ether to the wrong contract.

Long term, add additional tests to the codebase which validate that Ether cannot be unintentionally accepted by contracts without a corresponding withdrawal function.

5. Hardcoded Infura API keys

Severity: Low
Type: Configuration
Target: `truffle.js`

Difficulty: High
Finding ID: TOB-RAY-05

Description
The codebase contains hardcoded Infura API keys for each environment.

The severity of this issue is considered low because Infura does not sign transactions. An attacker is currently limited in what they may do with a stolen API key, although this may change in the future.

```
let kovan_infura_apikey = "https://kovan.infura.io/v3/[REDACTED]";
let rinkeby_infura_apikey = "https://rinkeby.infura.io/v3/[REDACTED]";
let main_infura_apikey = "https://mainnet.infura.io/v3/[REDACTED]";
```

Figure 5.1 - Lines 2-4 of `truffle.js`

Exploit Scenario
An attacker who has access to the codebase reuses the stolen API keys to intentionally cause legitimate Infura API requests to become ratelimited.

Recommendation
Short term, replace the referenced Infura API keys with new ones and securely store them outside of the codebase.

Long term, configure Infura projects to require project secrets and request whitelisting, as described in Infura's documentation. Project secrets should also be stored securely outside of the codebase.

References
- Infura Documentation for API Authentication

6. Solidity compiler is outdated

Severity: Low
Type: Patching
Target: Multiple contracts

Difficulty: Low
Finding ID: TOB-RAY-06

Description
While version 0.4.25 of the Solidity compiler is intended to be used by RAY, the following files use version 0.4.24 instead:

- `/contracts/external/WETH9.sol`
- `/contracts/Migrations.sol`

Version 0.4.24 of the Solidity compiler is known to be affected by several bugs, one of which is rated medium/high in severity. When using the ** operator, an unexpected value can result when the exponent is shorter than 256 bits. While none of the above files were found to contain vulnerable code, it is recommended that the version of the targeted Solidity compiler be updated to avoid potential issues in the future.

Exploit Scenario
A code change in the future exposes the affected contracts to a known exploitable bug.

Recommendation
Short term, update the pragma declaration of the affected files listed above so that they use the same version as is used in the rest of the codebase.

Long term, remove older versions of the Solidity compiler from the build environment so that it is not possible to accidentally compile the contracts using the wrong version.

References
- List of known Solidity compiler bugs

7. Solidity compiler optimizations can be dangerous

Severity: Low
Type: Configuration
Target: `truffle.js`

Difficulty: Low
Finding ID: TOB-RAY-07

Description
Staked has enabled optional compiler optimizations in Solidity.

There have been several bugs with security implications related to optimizations. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default. It is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6.

A compiler audit of Solidity from November, 2018 concluded that the optional optimizations may not be safe. Moreover, the Common Subexpression Elimination (CSE) optimization procedure is "implemented in a very fragile manner, with manual access to indexes, multiple structures with almost identical behavior, and up to four levels of conditional nesting in the same function." Similar code in other large projects have resulted in bugs.

There are likely latent bugs related to optimization, and/or new bugs that will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the RAY contracts.

Recommendation
In the short term, measure the gas savings from optimizations, and carefully weigh that against the possibility of an optimization-related bug.

In the long term, monitor the development and adoption of Solidity compiler optimizations to assess its maturity.

8. ABIEncoderV2 is Used

Severity: Undetermined
Type: Patching and Undefined Behavior
Target: Multiple contracts

Difficulty: Low
Finding ID: TOB-RAY-08

Description
ABIEnncoderV2 is used by the following files. This feature is still experimental and is not ready for production use.

- `/contracts/protocol/impl/opportunities/DydxImpl.sol`
- `/contracts/external/dydx/Types.sol`
- `/contracts/external/dydx/SoloMargin.sol`
- `/contracts/external/dydx/Operation.sol`
- `/contracts/external/dydx/Getters.sol`
- `/contracts/external/dydx/Actions.sol`
- `/contracts/external/dydx/Account.sol`

Over three percent of all GitHub issues for the Solidity compiler are related to experimental features, with `ABIEncoderV2` constituting the vast majority of these. Several issues and bug reports are still open and unresolved. `ABIEncoderV2` has been associated with over twenty high-severity bugs over the past year, with some so recent that they have not yet been included in a Solidity release. For example, earlier this year a severe bug was found in the encoder and was introduced in Solidity 0.5.5.

Exploit Scenario
Staked deploys its contracts. After the deployment, a bug is found in the encoder. As a result, the contracts are broken and are exploited to steal tokens.

Recommendation
Short term, use neither `ABIEncoderV2` nor any other experimental Solidity features. Refactor the code to alleviate the need to pass or return arrays of strings to and from functions.

Long term, integrate static analysis tools like Slither into your CI pipeline to detect unsafe pragmas.

9. Missing event logging

Severity: Low
Difficulty: Low
Type: Auditing and Logging
Finding ID: TOB-RAY-09
Target: Multiple contracts

Description
Sensitive functionality is missing adequate event logging. For example, all administrative functions in `/contracts/protocol/impl/governance/Admin.sol` are missing event logging, except for `addOpportunities`.

This issue is pervasive throughout the codebase, affecting both user and administrative functionality.

Exploit Scenario
A forensic analysis of an incident becomes more challenging due to a lack of adequate event logging.

Recommendation
Short term, ensure that all sensitive functionality has adequate event logging.

Long term, consider using a blockchain monitoring system to track any suspicious behavior in the contracts.

10. Race condition affecting purchase of SPR tokens

Severity: Medium
Difficulty: Medium
Type: Timing
Finding ID: TOB-RAY-10
Target: `/contracts/protocol/impl/secondary/Entrypoint.sol`

Description

A race condition affects the `purchaseSPRToken` function of the `Entrypoint` contract. If a victim gives an allowance to the contract so that `transferFrom` will succeed inside of the `verifyValue` function, an attacker can race the victim to call `purchaseSPRToken`. By setting the `beneficiary` call parameter to his address, it is possible to force the victim into entering a position that he did not intend to enter.

```
function purchaseSPRToken(
   bytes32 key,
   address beneficiary,
   uint value,
   bytes32[] memory upgrades
 )
   public
   payable
   returns(bytes32)
 {

   // verify the signed message etc. since we'll be sending this

   address rayContract = _storage.getVerifier(key);
   uint payableValue = verifyValue(key, beneficiary, value, rayContract);
```

Figure 10.1 - Lines 152-166 of /contracts/protocol/impl/secondary/Entrypoint.sol

```
function verifyValue(
   bytes32 key,
   address funder,
   uint inputValue,
   address rayContract
 )
   internal
   returns(uint)
 {

   address principalAddress = _storage.getPrincipalAddress(key);

   if (_storage.getIsERC20(principalAddress)) {

     require(IntERC20(principalAddress).transferFrom(funder, address(this), inputValue),
"#RAY verifyValue: Transfer of ERC20 Token failed");
     IntERC20(principalAddress).approve(rayContract, inputValue); // should one time max
approve the ray contract (or everytime it upgrades and changes addresses)
     return 0;
```

Figure 10.2 - Lines 316-332 of /contracts/protocol/impl/secondary/Entrypoint.sol

Exploit Scenario
An attacker succeeds in racing a victim to call `purchaseSPRToken`, and forces him to purchase SPR tokens for an opportunity he did not intend to invest in. In order to exit his position, the victim would need to call `withdrawPosition` and incur a fee.

Recommendation
Short term, validate that `msg.sender` is equal to the owner of the token.

Long term, eliminate the potential for race conditions to occur whenever possible by ensuring that all sensitive functionality, such as purchases and withdrawals, only take place within a single transaction.

11. Dependence on insecure WETH9

Severity: Medium
Type: Timing and Data Validation
Target: `/contracts/external/WETH9.sol`

Difficulty: High
Finding ID: TOB-RAY-11

Description
RAY depends on the WETH9 token, which is affected by a known race condition in the ERC20 standard's `approve` function. In addition, WETH9 is known to not fully conform to the ERC20 standard's `transferFrom` function. While these issues exist in third-party code, RAY's dependency on the token can lead to the theft of users' WETH9 tokens.

The ERC20 standard describes how to create generic token contracts. Among others, a ERC20 contract defines the following two functions:

- `transferFrom(from, to, value)`
- `approve(spender, value)`

These functions give permission to a third party to spend tokens. Once the function `approve(spender, value)` has been called by a user, spender can spend up to value tokens of the user's by calling `transferFrom(user, to, value)`.

This schema is vulnerable to a race condition when the user calls `approve` a second time on a spender that has already been allowed. If the spender sees the transaction containing the call before it has been mined, then the spender can call `transferFrom` to transfer the previous value and still receive the authorization to transfer the new value.

```
function approve(address guy, uint wad) public returns (bool) {
     allowance[msg.sender][guy] = wad;
     emit Approval(msg.sender, guy, wad);
     return true;
 }
```

Figure 11.1 - Lines 49-53 of /contracts/external/WETH9.sol

In addition, WETH9's `transferFrom` does not work as expected. If a user performs a withdrawal using her own address, the following code does not check her allowance. While this was not found to currently impact RAY, this behavior is unexpected and developers should therefore use caution when using the `transferFrom` function.

```
function transferFrom(address src, address dst, uint wad)
     public
     returns (bool)
```

```
    {
        require(balanceOf[src] >= wad);

        if (src != msg.sender && allowance[src][msg.sender] != uint(-1)) {
            require(allowance[src][msg.sender] >= wad);
            allowance[src][msg.sender] -= wad;
        }

        balanceOf[src] -= wad;
        balanceOf[dst] += wad;

        emit Transfer(src, dst, wad);

        return true;
    }
```
Figure 11.2 - Lines 59-79 of /contracts/external/WETH9.sol

Exploit Scenario

1. Alice calls `approve(Bob, 1000)`. This allows Bob to spend 1,000 tokens.
2. Alice changes her mind and calls `approve(Bob, 500).` Once mined, this will decrease to 500 the number of tokens that Bob can spend.
3. Bob sees the transaction and calls `transferFrom(Alice, X, 1000)` before `approve(Bob, 500)` has been mined.
4. If Bob's transaction is mined before Alice's, 1,000 tokens will be transferred by Bob. But once Alice's transaction is mined, Bob can call `transferFrom(Alice, X, 500)`. Bob has transferred 1500 tokens even though this was not Alice's intention.

Recommendation

While this issue is known and can have a severe impact, there is no straightforward solution. This issue is a flaw in the ERC20 design. Consider whether it is an acceptable risk for RAY to depend on the WETH9 token, or on any other ERC20 token.

12. Missing validation

Severity: Low
Type: Data Validation
Target: Multiple Contracts

Difficulty: Medium
Finding ID: TOB-RAY-12

Description
The following forms of missing validation were discovered within the codebase:

- Constructors were found to be missing any form of validation (e.g. checking that an address isn't equal to zero).
- Token upgrades in the `Entrypoint` contract aren't checked to ensure that they are associated with valid keys.
- Return values from external calls are not routinely checked.

While no instances were found to be directly exploitable, it is highly recommended that RAY's contracts consistently validate input and check return values to prevent bugs in the future.

```
constructor(
   address _stakedWalletAddress,
   address _weth,
   address _dai
 )
   public
 {

   stakedWalletAddress = _stakedWalletAddress;

   contracts[0x01] = _weth;
   contracts[0x02] = _dai;
   contracts[0x03] = msg.sender; // need to deploy Admin and then forgo this

   storageWrappers[msg.sender] = true;

 }
```

Figure 12.1 - Lines 109-125 /contracts/protocol/impl/Storage.sol

```
function purchaseSPRToken(
   bytes32 key,
   address beneficiary,
   uint value,
   bytes32[] memory upgrades
 )
   public
   payable
   returns(bytes32)
```

```
  {

    // verify the signed message etc. since we'll be sending this

    address rayContract = _storage.getVerifier(key);
    uint payableValue = verifyValue(key, beneficiary, value, rayContract);

    // we're the beneficiary/owner of this RAY (the point of this contract) if they wish to
    // have the option of upgrading automatically
    bytes32 tokenId = IRAYProtocol(rayContract).purchaseSPRToken.value(payableValue)(key,
address(this), value); // payable value could be 0

    RAYToken memory rayToken = RAYToken(beneficiary, upgrades);
    rayTokens[tokenId] = rayToken;
```

Figure 12.2 - Lines 152-173 of /contracts/protocol/impl/secondary/Entrypoint.sol. Note that the user-supplied array of updates is not validated.

```
    ●   /contracts/protocol/impl/secondary/Entrypoint.sol:331
    ●   /contracts/protocol/impl/opportunities/BzxImpl.sol:316
    ●   /contracts/protocol/impl/opportunities/BzxImpl.sol:379
    ●   /contracts/protocol/impl/opportunities/BzxImpl.sol:690
    ●   /contracts/protocol/impl/opportunities/DydxImpl.sol:397
    ●   /contracts/protocol/impl/opportunities/DydxImpl.sol:700
    ●   /contracts/protocol/impl/opportunities/DydxImpl.sol:883
    ●   /contracts/protocol/impl/secondary/Payer.sol:326
    ●   /contracts/protocol/impl/opportunities/CompoundImpl.sol:372
    ●   /contracts/protocol/impl/opportunities/CompoundImpl.sol:661
    ●   /contracts/protocol/impl/Router.sol:318
```

Figure 12.3 - A list of missing return value checks for external contract calls.

Exploit Scenario

A developer accidentally initializes a contract with incorrect data, which is accepted due to a lack of validation.

An ERC20 token transfer is rejected. This goes unnoticed for lack of a return value check.

Recommendation

Short term, update all contract constructors to perform adequate input validation, and consistently check the return values of called functions to ensure that operations were successful.

Long term, implement additional testing to ensure that contracts cannot be initialized with invalid data, and that public functions consistently validate input.

13. Rounding strategy may lead to errors

Severity: Low
Difficulty: High
Type: Numerics
Finding ID: TOB-RAY-13
Target: Multiple Contracts

Description
Rounding strategies are used that may lead to errors and impact the fund's value. For example, the following scenario can occur:

- A user wants to exit their position. This is handled by the `Router` contract's `withdrawPosition` function, which accepts an amount of Ether that the user would like to withdraw.
- This requires the `TokenHandler` contract to calculate how many tokens the withdrawn Ether is worth, which is implemented in its `calculatePercentage` function. The number of tokens calculated are to be burned.
- Rounding occurs when the value of shares is not divisible by the amount of Ether a user wants to withdraw. An on-chain precision of 18 decimal places is used to compensate.
- Rounding up means more tokens are burned and the value of the entire top-level fund is decreased. This results in the bottom-level fund (opportunity) increasing in value.
- Rounding down means fewer tokens are burned and the value of the entire top-level fund increases. This results in the bottom-level fund (opportunity) increasing in value.

Currently, the `calculatePercentage` function always rounds up. Other areas of the codebase, such as cumulative rate calculation, deal with potential rounding errors by using Solidity's default strategy of rounding down.

While it may be possible for an attacker to intentionally trigger rounding, this would be difficult to exploit due to the high precision used. Even so, it should be considered that over time an undesired amount of value may be lost due to rounding.

Exploit Scenario
An attacker exploits error-prone rounding by making repeated transactions that would intentionally incur rounding. This increases the amount of value lost due to rounding errors.

Recommendation

Short term, use the “round half to even” strategy, also known as “banker’s rounding.” This approach rounds halfway numbers up or down using a simple, dynamic policy which helps eliminate rounding bias. However, this approach assumes an even distribution of numbers that will be rounded up and down.

Long term, eliminate rounding wherever possible. For example, users currently request a specific amount of Ether to withdraw, which then requires the contract to calculate the number of corresponding tokens to burn. Instead, users can request a whole number of tokens to sell, and the amount of Ether they would get back would be the product of the tokens they want to sell and the value of a single token.

14. Use of unsafe arithmetic

Severity: Informational
Type: Numerics
Target: Multiple Contracts

Difficulty: Undetermined
Finding ID: TOB-RAY-14

Description

Multiple areas of the codebase are missing the SafeMath library, which provides a way to perform arithmetic that is safe from overflows and underflows. While none of the identified instances were discovered to be exploitable, it is recommended that RAY consistently use the SafeMath library to prevent issues arising in the future.

The following code snippets highlight examples where unsafe arithmetic is used, and should not be considered an exhaustive list.

```
 function updateCumulativeRate(bytes32 key) public notDeprecated returns(uint) {

      // this can never underflow, a miner can't change the timestamp so that
      // it is less than it's previous block. If multiple tx's come in on the same
      // block, timeSinceLastUpdate will be 0 as it should be.
      uint timeSinceLastUpdate = now - _storage.getLastUpdatedRate(key);

      // TODO: should add if (timeSinceLastUpdate > 0) to save on unneccessary gas from
manip. storage
      // 0 is possible since multiple transactions could come in the same block for the same
portfolio

      address principalAddress = _storage.getPrincipalAddress(key);

      uint rateToAdd = ((timeSinceLastUpdate * (_storage.getRate(principalAddress) *
ON_CHAIN_PRECISION / secondsPerYear)) / ON_CHAIN_PRECISION); // leave it scaled up 1e14

      uint cumulativeRate = _storage.getCumulativeRate(key) + rateToAdd;

StorageWrapperTwo(_storage.getContractAddress(getStorageWrapperTwoContract)).setCumulativeRa
te(key, cumulativeRate);

StorageWrapperTwo(_storage.getContractAddress(getStorageWrapperTwoContract)).setLastUpdatedR
ate(key, now);

      return cumulativeRate;

  }
```

Figure 14.1 - Lines 259-279 of /contracts/protocol/impl/FeeModel.sol

```
 function increaseTokenCapital(
      bytes32 opportunityKey,
      bytes32 tokenId,
      uint pricePerShare,
      uint value
  )
      public
      notDeprecated
      verifyContract(opportunityKey)
      verifyAmount(opportunityKey, value)
  {

      uint numOfShares = value * ON_CHAIN_PRECISION / pricePerShare;
      // I've used a BASE_PRICE_IN_WEI == 10000 and ON_CHAIN_PRECISION == 1e14 which is why
we have this multiplication
      // even though BASE_PRICE_IN_WEI == 1 right now. Experimenting with what works best to
mitigate dmg from the spam deposit/withdrawal attack.
      uint premiumValue = (numOfShares * (pricePerShare - (BASE_PRICE_IN_WEI *
ON_CHAIN_PRECISION))) / ON_CHAIN_PRECISION;

      uint newHistoricalYield = _storage.getHistoricalYield(opportunityKey) + premiumValue;

StorageWrapper(_storage.getContractAddress(getStorageWrapperContract)).setHistoricalYield(op
portunityKey, newHistoricalYield);

      uint tokenCapital = _storage.getTokenCapital(opportunityKey, tokenId);
      uint newTokenCapital = tokenCapital + value;

StorageWrapper(_storage.getContractAddress(getStorageWrapperContract)).setTokenCapital(oppor
tunityKey, tokenId, newTokenCapital);

      increaseTokenCapital2(opportunityKey, tokenId, numOfShares); // fix stack to deep
error

    }
```
Figure 14.2 - Lines 176-202 of /contracts/protocol/impl/TokenHandler.sol

Exploit Scenario
A change to the codebase introduces the potential for an integer overflow or underflow to be triggered by an attacker, thereby making it possible to steal Ether.

Recommendation
Short term, consistently use the SafeMath library for all addition, subtraction, multiplication, and division operations.

Long term, integrate static analysis tools like Slither into your CI pipeline to detect unsafe arithmetic.

15. Missing loss recovery

Severity: Informational
Difficulty: High
Type: Denial of Service
Finding ID: TOB-RAY-15
Target: Multiple Contracts

Description
In the event of a loss of capital, RAY has no defined loss recovery strategy. As is, a loss can cause the fund to become temporarily locked, and operators would need to manually deposit additional Ether for users to withdraw their original funds.

Exploit Scenario
An opportunity defaults and the RAY contracts no longer work. Users' funds are temporarily locked until Staked performs a recovery action.

Recommendation
Short term, thoroughly document and test all loss-recovery procedures.

Long term, consider automated solutions that would detect a loss and trigger a recovery process. For example, new shares could be minted and sold automatically.

16. Reentrancy from trusted third-party contracts

Severity: Informational
Difficulty: High
Type: Access Controls
Finding ID: TOB-RAY-16
Target: `/contracts/protocol/impl/secondary/Entrypoint.sol`

Description

The `purchaseSPRToken` function contains a potential reentrancy issue. Slither classifies this reentrancy as benign, as it would currently have the same effect as executing the function consecutively.

```
function purchaseSPRToken(
    bytes32 key,
    address beneficiary,
    uint value,
    bytes32[] memory upgrades
  )
    public
    payable
    returns(bytes32)
  {

    address rayContract = _storage.getVerifier(key);
    uint payableValue = verifyValue(key, beneficiary, value, rayContract);

    bytes32 tokenId = IRAYProtocol(rayContract).purchaseSPRToken.value(payableValue)(key,
address(this), value); // payable value could be 0

    RAYToken memory rayToken = RAYToken(beneficiary, upgrades);
    rayTokens[tokenId] = rayToken;
```

Figure 16.1 - Lines 152-173 of /contracts/impl/secondary/Entrypoint.sol

Note that while this function appears to adhere to the checks-effects-interactions pattern used to avoid reentrancy, before writing state variables, `purchaseSPRToken` makes an internal call to `verifyValue` which calls out to an external contract in order to transfer ERC20 tokens.

```
function verifyValue(
    bytes32 key,
    address funder,
    uint inputValue,
    address rayContract
  )
    internal
    returns(uint)
  {

    address principalAddress = _storage.getPrincipalAddress(key);

    if (_storage.getIsERC20(principalAddress)) {

      require(IntERC20(principalAddress).transferFrom(funder, address(this), inputValue),
"#RAY verifyValue: Transfer of ERC20 Token failed");
      IntERC20(principalAddress).approve(rayContract, inputValue); // should one time max
approve the ray contract (or everytime it upgrades and changes addresses)
```

Figure 16.2 - Lines 316-331 of /contracts/impl/secondary/Entrypoint.sol

Exploit Scenario
Staked unintentionally adds support for a malicious token. A user interacts with RAY and triggers the reentrancy. This has unintended side effects on the system.

Recommendations
Short term, remove the potential reentrancy and be aware that calls to `verifyValue` may result in unintentionally violating the checks-effects-interactions pattern.

Long term, integrate static analysis tools like Slither into your CI pipeline to detect reentrancy vulnerabilities.

17. External calls in loops may result in denial of service

Severity: Low
Difficulty: High
Type: Denial of Service
Finding ID: TOB-RAY-17
Target: `/contracts/protocol/impl/secondary/{NavCalulator, Oracle}.sol`

Description
Slither identified several functions which contain loops that make calls to other RAY contracts. Some of these loops scale according to the number of assets being held or supported. The following functions are at risk of causing a denial of service on the system, should the number of supported assets reach a threshold where these calls reach the block gas limit:

- `NavCalculator.getLentYield`
- `Oracle.withdrawFromProtocols`

Code comments and discussions with the Staked team indicate they are aware of this issue and are investigating the likelihood of encountering this issue and potential workarounds.

Exploit Scenario
Staked adds support to RAY for a variety of new assets. A user holds positions in many different Opportunities. As a result, when they attempt to withdraw from their position, they reach the block gas limit and are unable to withdraw their funds from the system.

Recommendation
Short term, investigate the gas usage of paths leading to these loops given the current set of supported assets and begin researching potential architectural alternatives to mitigate this issue.

Long term, favor incremental pull- over push-based logic wherever possible. Integrate static analysis tools like Slither into your CI pipeline to detect these types of calls in loops.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Numerics	Related to numeric calculations
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important

Medium	Individual user’s information is at risk, exploitation would be bad for client’s reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client’s reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

B. Code Quality

Code-quality findings are included to detail findings which are not immediate security concerns, but could lead to the introduction of vulnerabilities in the future.

- There are many functions which could be declared `external` instead of `public` for potential gas savings.
    - For a full list, run Slither with the `external-function` detector.
- There are several state variables which are currently unused. Consider removing these for deployment gas savings.
    - For a full list, run Slither with the `unused-state` detector.
- Replace the integer literal used in the `currCompoundBalance` calculation in `CompoundImpl.sol` with scientific notation or a constant variable for improved clarity.

C. Design-Related Issues

Staked requested that Trail of Bits address a list of design-related issues pertaining to the RAY smart contracts. This included rounding operations having an adverse affect on price per share, user withdrawals hitting gas limits due to the use of `for` loops, and the complexity associated with relying on many contract settings. This section of the document addresses each of these three concerns.

Rounding of Opportunity Token Affects PPS's Integrity

This issue was addressed in TOB-RAY-13, which concluded that while it may be possible for an attacker to intentionally trigger rounding, this would be difficult to exploit due to the high precision math involved.

In order to mitigate the residual risk associated with an undesired amount of value being lost to rounding over a long period of time:

1. Use the “round half to even” strategy, also known as “banker’s rounding.”
2. Eliminate rounding entirely wherever possible (e.g. by requiring users to only deal in denominations of tokens rather than arbitrary amounts of Ether).

For Loops Used for User Withdraws

In addition to the loops identified in TOB-RAY-17, Slither also identified several other instances where external calls were being made to other RAY contracts inside of a loop. Unlike the functions in TOB-RAY-17, however, these functions may work around an out-of-gas exception by being re-run on a subset of the input. The relevant functions are as follows:

- `Oracle.rebalance`
- `Admin.deprecateRAYController`
- `Admin.addOpportunities`
- `Admin.setContractsAddres`
- `Admin.setImplsPrincipalToken`
- `Admin.setCoinSettings`
- `Admin.setVerifierTODO`

Reliance on "Settings"

While this was not deemed to be a security issue, the complexity associated with having many contract settings should be noted. Trail of Bits recommends that Staked:

- Thoroughly document and test each setting, in order to prevent contracts from entering a bad state.
- Thoroughly document and test the upgradability of each contract, in order to ensure that recovery is possible in the event of a bug.
- Whenever possible, prefer automated deployment processes over manual deployment processes to eliminate the potential for human error.
- Ensure that contract deprecation checks are enforced by consistently using the `notDeprecated` modifier.

D. Property Testing with Echidna

The following is a list of Echidna tests for validating the WETH9 token, as used by RAY, against relevant ERC20 properties. While no issues resulted from these checks, this has been included as an example at the request of Staked.

```
import "./WETH9.sol";


contract CryticInterface{
    address internal crytic_owner = address(0x41414141);
    address internal crytic_user = address(0x42424242);
    address internal crytic_attacker = address(0x43434343);

    uint internal initialTotalSupply;
    uint internal initialBalance_owner;
    uint internal initialBalance_user;
    uint internal initialBalance_attacker;

    uint initialAllowance_user_attacker;
    uint initialAllowance_attacker_user;
    uint initialAllowance_attacker_attacker;
}

contract TEST is CryticInterface, WETH9 {

    constructor() public payable {
        initialTotalSupply = totalSupply();
        balanceOf[crytic_owner] = 0;
        balanceOf[crytic_user] = initialTotalSupply/2;
        initialBalance_user = initialTotalSupply/2;
        balanceOf[crytic_attacker] = initialTotalSupply/2;
        initialBalance_attacker = initialTotalSupply/2;
    }

    /*
    Type: Code quality
    Return: Success
    */
    function crytic_zero_always_empty_ERC20Properties() public returns(bool){
        return balanceOf[address(0x0)] == 0;
    }

    /*
    Type: Code Quality
    Return:
    */
    function crytic_approve_overwrites() public returns (bool) {
        bool approve_return;
        approve_return = approve(crytic_user, 10);
        require(approve_return);
        approve_return = approve(crytic_user, 20);
        require(approve_return);
        return allowance[msg.sender][crytic_user] == 20;
    }
    /*
    Type: Undetermined severity
    Return: Success
    */
    function crytic_less_than_total_ERC20Properties() public returns(bool){
        return balanceOf[msg.sender] <= totalSupply();
    }

    /*
    Type: Low severity
    Return: Success
    */
    function crytic_totalSupply_consistant_ERC20Properties() public returns(bool){
        return balanceOf[crytic_owner] + balanceOf[crytic_user] + balanceOf[crytic_attacker]
<= totalSupply();
    }

    /*
    Properties: Transferable
    */

    /*
    Type: Code Quality
    Return: Fail or Throw
    */
    function crytic_revert_transfer_to_zero_ERC20PropertiesTransferable() public returns
(bool) {
        if (balanceOf[msg.sender] == 0)
          revert();
        return transfer(address(0x0), balanceOf[msg.sender]);
    }

    /*
    Type: Code Quality
    Return: Fail or Throw
    */
    function crytic_revert_transferFrom_to_zero_ERC20PropertiesTransferable() public returns
(bool) {
        uint balance = balanceOf[msg.sender];
        bool approve_return = approve(msg.sender, balance);
        return transferFrom(msg.sender, address(0x0), balanceOf[msg.sender]);
    }

    /*
    Type: ERC20 Standard
    Fire: Transfer(msg.sender, msg.sender, balanceOf[msg.sender])
    Return: Success
    */
    function crytic_self_transferFrom_ERC20PropertiesTransferable() public returns(bool){
        uint balance = balanceOf[msg.sender];
        bool approve_return = approve(msg.sender, balance);
        bool transfer_return = transferFrom(msg.sender, msg.sender, balance);
        return (balanceOf[msg.sender] == balance) && approve_return && transfer_return;
    }


    /*
    Type: ERC20 Standard
    Return: Success
    */
    function crytic_self_transferFrom_to_other_ERC20PropertiesTransferable() public
returns(bool){
        uint balance = balanceOf[msg.sender];
        bool approve_return = approve(msg.sender, balance);
        bool transfer_return = transferFrom(msg.sender, crytic_owner, balance);
        return (balanceOf[msg.sender] == 0) && approve_return && transfer_return;
    }

    /*
    Type: ERC20 Standard
    Fire: Transfer(msg.sender, msg.sender, balanceOf[msg.sender])
    Return: Success
    */
    function crytic_self_transfer_ERC20PropertiesTransferable() public returns(bool){
        uint balance = balanceOf[msg.sender];
        bool transfer_return = transfer(msg.sender, balance);
        return (balanceOf[msg.sender] == balance) && transfer_return;
    }

    /*
    Type: ERC20 Standard
    Fire: Transfer(msg.sender, other, 1)
    Return: Success
    */
    function crytic_transfer_to_other_ERC20PropertiesTransferable() public returns(bool){
        uint balance = balanceOf[msg.sender];
        address other = crytic_user;
        if (other == msg.sender) {
           other = crytic_owner;
        }
        if (balance >= 1) {
           bool transfer_other = transfer(other, 1);
           return (balanceOf[msg.sender] == balance-1) && (balanceOf[other] >= 1) &&
transfer_other;
        }
        return true;
    }

    /*
    Type: ERC20 Standard
    Fire: Transfer(msg.sender, user, balance+1)
    Return: Fail or Throw
    */
    function crytic_revert_transfer_to_user_ERC20PropertiesTransferable() public
returns(bool){
        uint balance = balanceOf[msg.sender];
        if (balance == (2 ** 256 - 1))
            return true;
        bool transfer_other = transfer(crytic_user, balance+1);
        return true;
    }


    /*
    Properties: Not Burnable
    */

    /*
    Type: Undetermined severity
    Return: Success
    */
    function crytic_supply_constant_ERC20PropertiesNotBurnable() public returns(bool){
        return initialTotalSupply <= totalSupply();
    }


    /*
    Properties: Not Mintable, Not Burnable
    */

    /*
    Type: Undetermined severity
    Return: Success
    */
    function crytic_supply_constant_ERC20PropertiesNotMintableNotBurnable() public
returns(bool){
        return initialTotalSupply == totalSupply();
    }



    /*
    Properties: Not Mintable
    */

    /*
    Type: Undetermined severity
    Return: Success
    */
    function crytic_supply_constant_ERC20PropertiesNotMintable() public returns(bool){
        return initialTotalSupply >= totalSupply();
    }

}
```

Figure D.1 - An Echidna test contract for validating ERC20 properties.

```
seqLen: 50
testLimit: 20000
prefix: "crytic_"
deployer: "0x41414141"
sender: ["0x42424242", "0x43434343"]
dashboard: false
balanceAddr: 1000000000000000000000
balanceContract: 1000000000000000000000
```

Figure D.2 - An associated Echidna configuration file.

E. Property Testing with Manticore

The following is an example of how Manticore could be used to identify bugs in the `calculatePercentage` function. While no issues resulted from these checks, this has been included as an example at the request of Staked.

```
from manticore.ethereum import ManticoreEVM, ABI
from manticore.core.smtlib import Operators, solver

src = """
pragma solidity 0.4.25;

contract Test {
    uint internal constant ON_CHAIN_PRECISION = 1000000000000000000;

    function calculatePercentage(
      uint valueToWithdraw,
      uint shares,
      uint pricePerShare
    )
      pure
      returns (uint)
    {
      uint percentageOfValue = ceil(((valueToWithdraw * (ON_CHAIN_PRECISION * 10)) /
((shares * pricePerShare) / ON_CHAIN_PRECISION)), 10) / 10;
      return (shares * percentageOfValue / ON_CHAIN_PRECISION); // find shares to burn by
the % of the capital credited since ratio for shares:capital is constant
    }

    function ceil(uint a, uint m) internal pure returns (uint) {
       return ((a + m - 1) / m) * m;
   }
}
"""

m = ManticoreEVM()

withdraw = m.make_symbolic_value()
shares = 5
pps = 1000000000000000000 # 1 ETH

account = m.create_account(balance=1)
contract = m.solidity_create_contract(src, owner=account)

contract.calculatePercentage(withdraw, shares, pps)

for state in m.ready_states:
    burned = state.platform.transactions[-1].return_data
    burned = ABI.deserialize("uint", burned)

    condition = <constraint to test>

    if m.generate_testcase(state, name="BugFound", only_if=condition):
      print("Constraint violated! See {}".format(m.workspace))
```

F. Follow-Up Questions and Answers

This appendix aims to answer a list of follow-up questions provided by Staked on August 8, 2019.

1. A walkthrough of a few of the tests written that use Echidna, Manticore, etc.

- Please see Appendix D and Appendix E.
- One of our engineers recently gave a workshop at Trufflecon 2019. Their training material can be found here.

2. Best practices to automate checking if an external contract matches a certain interface we want them to implement, then checking if it has any malicious (or just wrong) behavior in the implementation. In our use-case, this would be the Opportunity interface. Since we send value to these contracts and require them to send us the value back when we request it, it's important to make sure the implementation has strict rules in place (set by us) that are followed. If not automated, then best practice manual screening process.

- We would recommend property testing for the invariants you want enforced. Both Echidna and Manticore can do this. For a list of ERC20 properties, please see Appendix D.

3. Fallbacks to employ inside the smart contracts to recover in case of a logical bug. (while considering we wish this to be trust-less). Are there measures you suggest? Ex. A tokens value calculation somehow is corrupted and gives them an excessive balance. What do we do at the point it's already happened? What do we do to help prevent this from ever occurring? Example: Speedbumps on certain actions (take 7 days to withdraw a token that exceeds a 25% return), take the 7 days to 'review' the transaction and decide if it's valid (through governance)

- Before launch, thoroughly document and test a recovery plan. This plan should identify what can go wrong (including items unrelated to smart contract bugs, such as wallet compromise), and specific actions to follow (e.g. pause the system, inform users, etc.). For additional information, please see our deployment best practices below.
- Ensure that event logging is routinely performed, as recommended in TOB-RAY-09. Then, implement a monitoring solution. Many open source and commercial solutions can be configured to watch for certain events and then trigger an action such as a Slack notification.

4. How to handle ongoing changes in a manner that preserves already audited contracts and logic (unless a logic upgrade is required). Ex. We want to change NCController.sol to add some functionality that has no effect on the general system of tracking value. How to introduce this change without requiring the overall system be re-audited?

- We would recommend identifying a list of invariants and performing appropriate property testing using Echidna and Manticore. This will help ensure that even in the event of a code change, the identified invariants still hold true.

5. Deployment best practices (Specific questions unknown at this time)

- Ensure that event logging is routinely performed, as recommended in TOB-RAY-09.
- The upgrade and migration strategy should be thoroughly documented and tested before deployment. If you have to react under stress, you will benefit from a clear procedure. The procedure must include components such as:
    - What are the functions to call to properly initialize the new contracts?
    - Who on the team is responsible for deploying upgrades? Who can act if they are unavailable?
    - Where are keys stored? Who has access to them?
    - What post-deployment script should be run to ensure the correct deployment?
    - How are users communicated with? What is the official procedure for making announcements?
- Do at least one test of a full upgrade/migration before deploying the initial version. It will allow you to confirm that your procedure contains all the required information.
- For information regarding secure key handling, please see Appendix G.
- For additional information on contract upgrade strategies, please see the following resources:
    - Contract Upgrade Anti-Patterns
    - How Contract Migration Works

And a few questions that can probably be answered off the call:

6. Is it possible to check the revert reasons in production as it is when developing locally and how to do it? (Assuming a require() caused the revert). If not, is it an ability coming in the future? Or do we need to adopt the approach of catching errors but not reverting, instead have an enum or something similar of errors - to associate them with different numbers, and returning the appropriate number/error instead? Compound takes this approach, but it seems safer to force reverts when undesirable states occur. It would make debugging errors in prod. easier naturally.

- We discourage manual error handling, as it is error-prone itself. Instead, we recommend continuing to use `require` and allowing reverts to be triggered.
- For quickly obtaining the reason for a failed transaction, see eth-reveal. For the general debugging of transactions, see debug_tracetransaction.

7. Can one change interfaces visibility modifiers for functions from public to external? Or would that change the ABI in a way in certain scenarios that makes it incompatible and introduces bugs with the actual implementation? Ex. ERC20 uses a public visibility modifier on setBalance(), if I changed that to external in the interface I'm inheriting from but the ERC20 token function setBalance() I'm trying to call at runtime uses public visibility and not external would that have any security issues? I understand the difference between the two (reading from calldata vs memory) but not sure if it may introduce issues in some edge case.

- We understand this question to be asking if changing a function's visibility from public to external results in an ABI change, and if so is it unsafe? This change would not change the ABI and would therefore be safe to use.

8. Best sources to watch for the latest security issues. Ex. Trail of Bits blog

- The Trail of Bits blog
- The Ethereum blog
- New releases of Solidity and associated notes
- Blogs/communities for projects you rely on or interact with (e.g., OpenZeppelin, MakerDAO, etc.) in order to stay on top of any issues in RAY's dependencies.
- General Ethereum newsletters such as the Week In Ethereum or EthHub Weekly will also cover Ethereum-wide security issues worth knowing about.

G. Key Handling Guidance

This appendix covers concerns surrounding key material handling, as requested by Staked.

The safety of key material is important in any system, but particularly so in Ethereum; keys dictate access to money and resources. Theft of keys could mean a complete loss of funds or trust in the market. The current configuration uses an environment variable in production to relay key material to applications that use these keys for interacting with on-chain components. However, attackers with local access to the machine may be able to extract these environment variables and steal key material, even without privileged positions. Therefore, we recommend the following:

- Move key material from environment variables to a dedicated secret management system with trusted computing capabilities. The two best options for this are Google Cloud Key Management System (GCKMS) or Hashicorp Vault with Hardware Security Module (HSM) backing.
- Restrict access to GCKMS or Hashicorp Vault to only those applications and administrators that must have access to the credential store.
- Local key material, such as keys used by fund administrators, may be stored in local HSMs, such as YubiHSM2.
- Limit the number of staff members and applications with access to this machine.
- Segment the machine away from all other hosts on the network.
- Ensure strict host logging, patching, and auditing policies are in place for any machine or application that handles said material.
- Determine the business risk of a lost or stolen key, and what the Disaster Recovery and Business Continuity (DR/BC) policies are in the event of a stolen or lost key.

H. Fix Log for August 30, 2019

Staked requested that Trail of Bits perform a limited follow-up assessment over the course of four hours, during which the following items were prioritized:

- Validating constructors
- Verifying return values
- Using banker's rounding
- Unsafe arithmetic affecting withdrawals

No new issues were discovered. The section below details Trail of Bits' validation of Staked's remediation efforts.

Validating Constructors

This issue is related to TOB-RAY-12 and pertains to missing validation affecting multiple contract constructors. Trail of Bits found that this issue was not remediated and remains open. New contracts (e.g., FeeModel) and old contracts both contain constructors with missing validation.

Verifying Return Values

This issue is related to TOB-RAY-12 and pertains to missing return value checks from external function calls. Trail of Bits found that this issue was not remediated and remains open. New contracts (e.g., PortfolioManager) and old contracts both contain external function calls that are missing return value checks.

Using Banker's Rounding

This issue is related to TOB-RAY-13 and pertains to error-prone rounding that can impact the fund's value. Staked communicated that due to its multi-layer design, rounding is always necessary and Trail of Bits' long-term recommendation was not feasible to implement. Instead, Trail of Bits' short-term recommendation was implemented and banker's rounding is now used for both RAY and Opportunities.

While this issue is considered closed, Trail of Bits recommends that Staked continue to explore ways to simplify value calculations and reduce both rounding errors and complexity wherever possible in its codebase.

Unsafe arithmetic affecting withdrawals

This issue is related to TOB-RAY-14 and pertains to unsafe arithmetic operations that are susceptible to integer underflows and overflows. Trail of Bits found that this issue was not remediated and remains open. While basic validations are performed in several areas of the codebase, the SafeMath library remains unused for a majority of arithmetic operations, including those related to withdrawal functionality.