Yield V2

Security Assessment

October 19, 2021

Prepared for:

Allan Niemerg
Yield Protocol

Alberto Cuesta Cañada
Yield Protocol

Prepared by:

Natalie Chin and Maximilian Krüger

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

Classification and Copyright

This report has been made public at the request of Yield.

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 mutually 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 such, 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.

Table of Contents

Executive Summary

Overview

Yield engaged Trail of Bits to review the security of its Yield protocol V2 smart contracts. From September 13 to October 1, 2021, a team of two consultants conducted a security review of the client-provided source code, with six person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

We focused our testing efforts on the identification of flaws that could result in a compromise or lapse of confidentiality, integrity, or availability of the target system. We performed automated testing and a manual review of the code.

Summary of Findings

Our review resulted in 18 findings, including 5 of high severity and 4 of medium severity. One of the high-severity issues has a difficulty level of low, which means that an attacker would not have to overcome significant obstacles to exploit it. The most severe issue stems from a lack of access controls and could allow any user to liquidate any vault, draining funds from the protocol for profit.

EXPOSURE ANALYSIS

Severity	*Count*
High	5
Medium	4
Low	1
Informational	6
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Data Validation	8
Patching	4
Configuration	2
Undefined Behavior	2
Timing	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager dan.guido@trailofbits.com	**Sam Greenup**, Project Manager sam.greenup@trailofbits.com

The following engineers were associated with this project:

Maximilian Krüger, Consultant max.kruger@trailofbits.com	**Natalie Chin**, Consultant natalie.chin@trailofbits.com

Project Timeline

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

Date	Event
September 13, 2021	Project pre-kickoff call
September 20, 2021	Status update meeting #1
September 27, 2021	Status update meeting #2
October 4, 2021	Delivery of report draft
October 4, 2021	Report readout meeting
October 18, 2021	Fix Log added (Appendix F)

Project Targets

The engagement involved a review and testing of the targets listed below.

vault-v2

Repository	https://github.com/yieldprotocol/vault-v2/
Versions	`819a713416249da92c44eb629ed26a49425a4656`
	`9b36585830af03e71798fa86ead9ab4d92b6dd7c` (for `Witch.sol`)
Type	Solidity
Platform	Ethereum

yieldspace-v2

Repository	https://github.com/yieldprotocol/yieldspace-v2
Version	`36405150567a247e2819c1ec1d35cf0ab666353a`
Type	Solidity
Platform	Ethereum

yield-utils-v2

Repository	https://github.com/yieldprotocol/yield-utils-v2
Version	`a5cfe0c95e22e136e32ef85e5eef171b0cb18cd6`
Type	Solidity
Platform	Ethereum

strategy-v2

Repository	https://github.com/yieldprotocol/strategy-v2
Version	`fef352a339c19d8a4975de327e65874c5c6b3fa7`
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 and their results include the following:

`Cauldron`. This non-upgradeable contract is the core accounting system of the protocol and keeps track of features including vaults, series, asset whitelists, and collateral whitelists. We checked that the functions are implemented correctly, that proper access controls are in place, and that inputs are validated correctly.

`Ladle`. Users interact with the protocol primarily by calling `Ladle` functions batched together in a "recipe." The `Ladle` serves as a gatekeeper and vault manager, enabling users to create vaults, add liquidity, transfer tokens, and repay debt. We checked the correctness of the implementation, the fund-transfer process, and the data validation.

`Witch`. The `Witch` contract is the liquidation engine of the protocol. It enables users to start Dutch auctions for the collateral of undercollateralized vaults and to then buy some or all of the collateral. We checked the implementation of these Dutch auctions, during which the price of collateral decreases linearly (for the duration of an auction) to a configurable fraction of the initial price. We also checked whether vaults that are not undercollateralized can be liquidated and compared the liquidation engine to the MakerDAO Liquidations 2.0 system, which is a very similar system.

`Pool`. The `Pool` contract facilitates the exchange of base tokens for wrapped fyTokens by maintaining the YieldSpace invariant. We reviewed the flow of funds through the contract and the preconditions of the minting flow and checked whether fyTokens are minted properly.

`Strategy`. This contract allows users to exchange their liquidity provider (LP) tokens for strategy tokens and additional rewards. We checked that strategy tokens can be minted only when the contract is connected to a pool and that the distribution of rewards adheres to the expected schedule.

`Join`. This contract is deployed each time a new asset is added to the protocol and holds the protocol's balance of that asset. We reviewed the correctness of the API's implementation, the fund-transfer process, and the access controls.

`FYToken`. This contract implements the fyToken. This synthetic token can be redeemed at the price of the underlying asset, which is provided by an oracle, upon its maturity date. We manually reviewed the implementation and its access controls.

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
Access Controls and Data Validation	We identified one high-severity issue stemming from a lack of proper access controls, which could allow an attacker to liquidate any vault (TOB-YP2-006). The access controls also lack sufficient tests, which could have caught the high-severity issue. Additionally, many individual functions lack comprehensive access controls and data validation; instead, the access controls and data validation for those functions are implemented only once per call stack (TOB-YP2-017). The protocol's contracts are authorized to call only certain functions on one another, which limits each contract's privileges. However, this makes the correctness of the access controls highly dependent on the correctness of the deployment; it also means that the access controls are difficult to verify from the code alone.	Weak
Arithmetic	Solidity 0.8's `SafeMath` is used throughout the project, and we did not find any arithmetic issues of medium or high severity. However, the arithmetic is not tested through fuzzing or symbolic execution, which would help ensure its correctness. Additionally, the documentation on the arithmetic would benefit from further detail.	Satisfactory
Assembly Use/Low-Level Calls	The contracts use assembly for optimization purposes but lack comments documenting its use. We identified two high-severity issues involving the lack of a contract existence check prior to execution of a `delegatecall` (TOB-YP2-001) and the use of a `delegatecall` in a	Moderate

	payable function, which may cause unexpected behavior (TOB-YP2-002).	
Code Stability	The code underwent frequent changes before and during the audit and will likely continue to evolve.	**Moderate**
Decentralization	A couple of externally owned accounts held by the Yield team have near-total control over the protocol, making it a centralized system that requires trust in a single entity. However, the Yield team intends to transfer control to a governance system operating through a timelock contract, which will reduce the protocol's centralization.	**Weak**
Upgradeability	The core accounting contract, the `Cauldron`, is not upgradeable. Other contracts, such as the `Ladle` and `Witch`, can be replaced with new versions authorized to call the `Cauldron`. We did not find any issues caused by this database pattern of upgradeability. The protocol does not use the complex and error-prone `delegatecall` pattern of upgradeability.	**Satisfactory**
Function Composition	Many of the system's functionalities, especially its data validation functionalities, are broken up into multiple functions in order to save gas. This makes some of the code less readable and more difficult to modify (TOB-YP2-017).	**Moderate**
Front-Running	We found one high-severity issue related to front-running (TOB-YP2-009). However, time constraints prevented us from exhaustively checking the protocol for front-running and unintended arbitrage opportunities.	**Further Investigation Required**
Monitoring	The Yield Protocol's functions emit events for critical operations. Additionally, Yield indicated that it has an incident response plan, and the protocol uses Tenderly to monitor on-chain activity.	**Satisfactory**

Specification	Yield provided its “Syllabus,” “Cookbook,” and “Deployment” documents, as well as the Yield Protocol and YieldSpace whitepapers, as documentation. However, Trail of Bits recommends creating additional technical documentation that details the optimizations in the system and the abilities of privileged users; this documentation should also include architecture diagrams showing the flow of funds through the system.	**Satisfactory**
Testing and Verification	The codebase contains an adequate number of unit tests. However, it lacks tests for simple access controls, which could have caught the high-severity issue outlined in TOB-YP2-006. The test suite also lacks advanced testing methods like fuzzing and symbolic execution, which are required for proper arithmetic testing.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Lack of contract existence check on delegatecall may lead to unexpected behavior	Data Validation	**High**
2	Use of delegatecall in a payable function inside a loop	Data Validation	**High**
3	Lack of two-step process for critical operations	Data Validation	**Medium**
4	Risks associated with use of ABIEncoderV2	Patching	**Undetermined**
5	Project dependencies contain vulnerabilities	Patching	**Medium**
6	Witch's buy and payAll functions allow users to buy collateral from vaults not undergoing auctions	Access Controls	**High**
7	Solidity compiler optimizations can be problematic	Undefined Behavior	**Informational**
8	Risks associated with EIP-2612	Configuration	**Informational**
9	Failure to use the batched transaction flow may enable theft through front-running	Data Validation	**High**
10	Strategy contract's balance-tracking system could facilitate theft	Data Validation	**High**
11	Insufficient protection of sensitive keys	Configuration	**Medium**
12	Lack of limits on the total amount of collateral sold at auction	Data Validation	**Medium**

13	Lack of incentives for calls to Witch.auction	Timing	**Undetermined**
14	Contracts used as dependencies do not track upstream changes	Patching	**Low**
15	Cauldron’s give and tweak functions lack vault existence checks	Data Validation	**Informational**
16	Problematic approach to data validation and access controls	Data Validation	**Informational**
17	isContract may behave unexpectedly	Undefined Behavior	**Informational**
18	Use of multiple repositories	Patching	**Informational**

Detailed Findings

1. Lack of contract existence check on delegatecall may lead to unexpected behavior	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-YP2-001
Target: `vault-v2/contracts/Ladle.sol`	

Description

The `Ladle` contract uses the `delegatecall` proxy pattern. If the implementation contract is incorrectly set or is self-destructed, the contract may not detect failed executions.

The `Ladle` contract implements the `batch` and `moduleCall` functions; users invoke the former to execute batched calls within a single transaction and the latter to make a call to an external module. Neither function performs a contract existence check prior to executing a `delegatecall`. Figure 1.1 shows the `moduleCall` function.

```
/// @dev Allow users to use functionality coded in a module, to be used with batch
/// @notice Modules must not do any changes to the vault (owner, seriesId, ilkId),
/// it would be disastrous in combination with batch vault caching
function moduleCall(address module, bytes calldata data)
    external payable
    returns (bytes memory result)
{
    require (modules[module], "Unregistered module");
    bool success;
    (success, result) = module.delegatecall(data);
    if (!success) revert(RevertMsgExtractor.getRevertMsg(result));
}
```

Figure 1.1: `vault-v2/contracts/Ladle.sol#L186-L197`

An external module’s address must be registered by an administrator before the function calls that module.

```
/// @dev Add or remove a module.
function addModule(address module, bool set)
    external
    auth
{
    modules[module] = set;
    emit ModuleAdded(module, set);
}
```

Figure 1.2: `vault-v2/contracts/Ladle.sol#L143-L150`

If the administrator sets the module to an incorrect address or to the address of a contract that is subsequently destroyed, a `delegatecall` to it will still return success. This means that if one call in a batch does not execute any code, it will still appear to have been successful, rather than causing the entire batch to fail.

The Solidity documentation includes the following warning:

```
The low-level functions call, delegatecall and staticcall return true as their first
return value if the account called is non-existent, as part of the design of the
EVM. Account existence must be checked prior to calling if needed.
```

Figure 1.3: A snippet of the Solidity documentation detailing unexpected behavior related to `delegatecall`

Exploit Scenario

Alice, a privileged member of the Yield team, accidentally sets a module to an incorrect address. Bob, a user, invokes the `moduleCall` method to execute a batch of calls. Despite Alice's mistake, the `delegatecall` returns success without making any state changes or executing any code.

Recommendations

Short term, implement a contract existence check before a `delegatecall`. Document the fact that using `suicide` or `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from introducing these functions.

Long term, carefully review the Solidity documentation, especially the "Warnings" section, and the pitfalls of using the `delegatecall` proxy pattern.

2. Use of delegatecall in a payable function inside a loop

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-YP2-002
Target: `vault-v2/contracts/Ladle.sol`	

Description

The `Ladle` contract uses the `delegatecall` proxy pattern (which takes user-provided call data) in a `payable` function within a loop. This means that each `delegatecall` within the `for` loop will retain the `msg.value` of the transaction:

```
 /// @dev Allows batched call to self (this contract).
 /// @param calls An array of inputs for each call.
 function batch(bytes[] calldata calls) external payable returns(bytes[] memory
results) {
     results = new bytes[](calls.length);
     for (uint256 i; i < calls.length; i++) {
         (bool success, bytes memory result) = address(this).delegatecall(calls[i]);
         if (!success) revert(RevertMsgExtractor.getRevertMsg(result));
         results[i] = result;
     }

     // build would have populated the cache, this deletes it
     cachedVaultId = bytes12(0);
 }
```

Figure 2.1: `vault-v2/contracts/Ladle.sol#L186-L197`

The protocol does not currently use the `msg.value` in any meaningful way. However, if a future version or refactor of the core protocol introduced a more meaningful use of it, it could be exploited to tamper with the system arithmetic.

Exploit Scenario

Alice, a member of the Yield team, adds a new functionality to the core protocol that adjusts users' balances according to the `msg.value`. Eve, an attacker, uses the batching functionality to increase her ETH balance without actually sending funds from her account, thereby stealing funds from the system.

Recommendations

Short term, document the risks associated with the use of `msg.value` and ensure that all developers are aware of this potential attack vector.

Long term, detail the security implications of all functions in both the documentation and the code to ensure that potential attack vectors do not become exploitable when code is refactored or added.

References

- ”Two Rights Might Make a Wrong,” Paradigm

3. Lack of two-step process for critical operations

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-YP2-003
Target: `vault-v2/contracts/Cauldron.sol`	

Description

The `_give` function in the `Cauldron` contract transfers the ownership of a vault in a single step. There is no way to reverse a one-step transfer of ownership to an address without an owner (i.e., an address with a private key not held by any user). This would not be the case if ownership were transferred through a two-step process in which an owner proposed a transfer and the prospective recipient accepted it.

```
 /// @dev Transfer a vault to another user.
function _give(bytes12 vaultId, address receiver)
    internal
    returns(DataTypes.Vault memory vault)
{
    require (vaultId != bytes12(0), "Vault id is zero");
    vault = vaults[vaultId];
    vault.owner = receiver;
    vaults[vaultId] = vault;
    emit VaultGiven(vaultId, receiver);
}
```

Figure 3.1: `vault-v2/contracts/Cauldron.sol#L227-L237`

Exploit Scenario

Alice, a Yield Protocol user, transfers ownership of her vault to her friend Bob. When entering Bob's address, Alice makes a typo. As a result, the vault is transferred to an address with no owner, and Alice's funds are frozen.

Recommendations

Short term, use a two-step process for ownership transfers. Additionally, consider adding a zero-value check of the receiver's address to ensure that vaults cannot be transferred to the zero address.

Long term, use a two-step process for all irrevocable critical operations.

4. Risks associated with use of ABIEncoderV2	
Severity: **Undetermined**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-YP2-004
Target: Throughout the codebase	

Description

The contracts use Solidity's `ABIEncoderV2`, which is enabled by default in Solidity version 0.8. This encoder has caused numerous issues in the past, and its use may still pose risks.

More than 3% of all GitHub issues for the Solidity compiler are related to current or former experimental features, primarily `ABIEncoderV2`, which was long considered experimental. Several issues and bug reports are still open and unresolved. `ABIEncoderV2` has been associated with more than 20 high-severity bugs, some of which are so recent that they have not yet been included in a Solidity release.

For example, in March 2019 a severe bug introduced in Solidity 0.5.5 was found in the encoder.

Exploit Scenario

The Yield Protocol smart contracts are deployed. After the deployment, a bug is found in the encoder, which means that the contracts are broken and can all be exploited in the same way.

Recommendations

Short term, use neither `ABIEncoderV2` nor any experimental Solidity feature. Refactor the code such that structs do not need to be passed to or returned from functions.

Long term, integrate static analysis tools like Slither into the continuous integration pipeline to detect unsafe pragmas.

5. Project dependencies contain vulnerabilities

Severity: **Medium**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-YP2-005
Target: `package.json`	

Description

Although dependency scans did not yield a direct threat to the project under review, `yarn audit` identified dependencies with known vulnerabilities. Due to the sensitivity of the deployment code and its environment, it is important to ensure dependencies are not malicious. Problems with dependencies in the JavaScript community could have a significant effect on the repositories under review. The output below details these issues.

NPM Advisory	Description	Dependency
1674	Arbitrary Code Execution	`underscore`
1755	Regular Expression Denial of Service	`normalize-url`
1770	Arbitrary File Creation/Overwrite due to insufficient absolute path sanitization	`tar`

Figure 5.1: NPM Advisories affecting project dependencies

Exploit Scenario

Alice installs the dependencies of an in-scope repository on a clean machine. Unbeknownst to Alice, a dependency of the project has become malicious or exploitable. Alice subsequently uses the dependency, disclosing sensitive information to an unknown actor.

Recommendations

Short term, ensure dependencies are up to date. Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation.

Long term, consider integrating automated dependency auditing into the development workflow. If a dependency cannot be updated when a vulnerability is disclosed, ensure that the codebase does not use and is not affected by the vulnerable functionality of the dependency.

6. Witch’s buy and payAll functions allow users to buy collateral from vaults not undergoing auctions

Severity: **High**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-YP2-006
Target: `vault-v2/contracts/Witch.sol`	

Description

The `buy` and `payAll` functions in the `Witch` contract enable users to buy collateral at an auction. However, neither function checks whether there is an active auction for the collateral of a vault. As a result, anyone can buy collateral from any vault. This issue also creates an arbitrage opportunity, as the collateral of an overcollateralized vault can be bought at a below-market price. An attacker could drain vaults of their funds and turn a profit through repeated arbitrage.

Exploit Scenario

Alice, a user of the Yield Protocol, opens an overcollateralized vault. Attacker Bob calls `payAll` on Alice’s vault. As a result, Alice’s vault is liquidated, and she loses the excess collateral (the portion that made the vault overcollateralized).

Recommendations

Short term, ensure that `buy` and `payAll` fail if they are called on a vault for which there is no active auction.

Long term, ensure that all functions revert if the system is in a state in which they are not allowed to be called.

7. Solidity compiler optimizations can be problematic	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-YP2-007
Target: `hardhat.config.js`	

Description

The Yield Protocol V2 contracts have enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and 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. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs 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 Yield Protocol V2 contracts.

Recommendations

Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

8. Risks associated with EIP-2612

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-YP2-008
Target: `yield-utils-v2`	

Description

The use of EIP-2612 increases the risk of `permit` function front-running as well as phishing attacks.

EIP-2612 uses signatures as an alternative to the traditional `approve` and `transferFrom` flow. These signatures allow a third party to transfer tokens on behalf of a user, with verification of a signed message.

The use of EIP-2612 makes it possible for an external party to front-run the `permit` function by submitting the signature first. Then, since the signature has already been used and the funds have been transferred, the actual caller's transaction will fail. This could also affect external contracts that rely on a successful `permit()` call for execution.

EIP-2612 also makes it easier for an attacker to steal a user's tokens through phishing by asking for signatures in a context unrelated to the Yield Protocol contracts. The hash message may look benign and random to the user.

Exploit Scenario

Bob has 1,000 iTokens. Eve creates an ERC20 token with a malicious airdrop called `ProofOfSignature`. To claim the tokens, participants must sign a hash. Eve generates a hash to transfer 1,000 iTokens from Bob. Eve asks Bob to sign the hash to get free tokens. Bob signs the hash, and Eve uses it to steal Bob's tokens.

Recommendations

Short term, develop user documentation on edge cases in which the signature-forwarding process can be front-run or an attacker can steal a user's tokens via `phishing`.

Long term, document best practices for Yield Protocol users. In addition to taking other precautions, users must do the following:

- Be extremely careful when signing a message
- Avoid signing messages from suspicious sources

- Always require hashing schemes to be public

References

- EIP-2612 Security Considerations

9. Failure to use the batched transaction flow may enable theft through front-running

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-YP2-009
Target: `strategy-v2/contracts/Strategy.sol`	

Description

The Yield Protocol relies on users interacting with the `Ladle` contract to batch their transactions (e.g., to transfer funds and then mint/burn the corresponding tokens in the same series of transactions). When they deviate from the batched transaction flow, users may lose their funds through front-running.

For example, an attacker could front-run the `startPool()` function to steal the initial mint of strategy tokens. The function relies on liquidity provider (LP) tokens to be transferred to the `Strategy` contract and then used to mint strategy tokens. The first time that strategy tokens are minted, they are minted directly to the caller:

```
 /// @dev Start the strategy investments in the next pool
 /// @notice When calling this function for the first pool, some underlying needs to
be transferred to the strategy first, using a batchable router.
 function startPool()
     external
     poolNotSelected
 {
     [...]

     // Find pool proportion p = tokenReserves/(tokenReserves + fyTokenReserves)
     // Deposit (investment * p) base to borrow (investment * p) fyToken
     //   (investment * p) fyToken + (investment * (1 - p)) base = investment
     //   (investment * p) / ((investment * p) + (investment * (1 - p))) = p
     //   (investment * (1 - p)) / ((investment * p) + (investment * (1 - p))) = 1 -
p

     uint256 baseBalance = base.balanceOf(address(this));
     require(baseBalance > 0, "No funds to start with");

     uint256 baseInPool = base.balanceOf(address(pool_));
     uint256 fyTokenInPool = fyToken_.balanceOf(address(pool_));

     uint256 baseToPool = (baseBalance * baseInPool) / (baseInPool + fyTokenInPool);
// Rounds down
     uint256 fyTokenToPool = baseBalance - baseToPool;        // fyTokenToPool is
rounded up

     // Mint fyToken with underlying
     base.safeTransfer(baseJoin, fyTokenToPool);
     fyToken.mintWithUnderlying(address(pool_), fyTokenToPool);

     // Mint LP tokens with (investment * p) fyToken and (investment * (1 - p)) base
     base.safeTransfer(address(pool_), baseToPool);
     (,, cached) = pool_.mint(address(this), true, 0); // We don't care about
slippage, because the strategy holds to maturity and profits from sandwiching

     if (_totalSupply == 0) _mint(msg.sender, cached); // Initialize the strategy if
needed

     invariants[address(pool_)] = pool_.invariant();   // Cache the invariant to
help the frontend calculate profits
     emit PoolStarted(address(pool_));
 }
```

Figure 9.1: `strategy-v2/contracts/Strategy.sol#L146-L194`

Exploit Scenario

Bob adds underlying tokens to the `Strategy` contract without using the router. Governance calls `setNextPool()` with a new pool address. Eve, an attacker, front-runs the call to the `startPool()` function to secure the strategy tokens initially minted for Bob's underlying tokens.

Recommendations

Short term, to limit the impact of function front-running, avoid minting tokens to the callers of the protocol's functions.

Long term, document the expectations around the use of the router to batch transactions; that way, users will be aware of the front-running risks that arise when it is not used. Additionally, analyze the implications of all uses of `msg.sender` in the system, and ensure that users cannot leverage it to obtain tokens that they do not deserve; otherwise, they could be incentivized to engage in front-running.

10. Strategy contract’s balance-tracking system could facilitate theft

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-YP2-010
Target: `strategy-v2/contracts/Strategy.sol`	

Description

`Strategy` contract functions use the contract’s balance to determine how many liquidity or base tokens to provide to a user minting or burning tokens.

The `Strategy` contract inherits from the `ERC20Rewards` contract, which defines a reward token and a reward distribution schedule. An admin must send reward tokens to the `Strategy` contract to fund its reward payouts. This flow relies on an underlying assumption that the reward token will be different from the base token.

```
/// @dev Set a rewards token.
/// @notice Careful, this can only be done once.
function setRewardsToken(IERC20 rewardsToken_)
    external
    auth
{
    require(rewardsToken == IERC20(address(0)), "Rewards token already set");
    rewardsToken = rewardsToken_;
    emit RewardsTokenSet(rewardsToken_);
}
```

Figure 10.1: `yield-utils-v2/contracts/token/ERC20Rewards.sol#L58-L67`

The `burnForBase()` function tracks the `Strategy` contract’s base token balance. If the base token is used as the reward token, the contract’s base token balance will be inflated to include the reward token balance (and the balance tracked by the function will be incorrect). As a result, when attempting to burn strategy tokens, a user may receive more base tokens than he or she deserves for the number of strategy tokens being burned:

```
 /// @dev Burn strategy tokens to withdraw base tokens. It can be called only when a
pool is not selected.
```

```
 /// @notice The strategy tokens that the user burns need to have been transferred
previously, using a batchable router.
 function burnForBase(address to)
     external
     poolNotSelected
     returns (uint256 withdrawal)
 {
     // strategy * burnt/supply = withdrawal
     uint256 burnt = _balanceOf[address(this)];
     withdrawal = base.balanceOf(address(this)) * burnt / _totalSupply;

     _burn(address(this), burnt);
     base.safeTransfer(to, withdrawal);
 }
```

Figure 10.2: `strategy-v2/contracts/Strategy.sol#L258-L271`

Exploit Scenario

Bob deploys the `Strategy` contract; DAI is set as a base token of that contract and is also defined as the reward token in the `ERC20Rewards` contract. After a pool has officially been closed, Eve uses `burnWithBase()` to swap base tokens for strategy tokens. Because the calculation takes into account the base token's balance, she receives more base tokens than she should.

Recommendations

Short term, add checks to verify that the reward token is not set to the base token, liquidity token, fyToken, or strategy token. These checks will ensure that users cannot leverage contract balances that include reward token balances to turn a profit.

Long term, analyze all token interactions in the contract to ensure they do not introduce unexpected behavior into the system.

11. Insufficient protection of sensitive keys

Severity: **Medium**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-YP2-011
Target: `hardhat.config.js`	

Description

Sensitive information such as Etherscan keys, API keys, and an owner private key used in testing is stored in the process environment. This method of storage could make it easier for an attacker to compromise the keys; compromise of the owner key, for example, could enable an attacker to gain owner privileges and steal funds from the protocol.

The following portion of the `hardhat.config.js` file uses secrets from the process environment:

```
 let mnemonic = process.env.MNEMONIC
 if (!mnemonic) {
   try {
     mnemonic = fs.readFileSync(path.resolve(__dirname,
'.secret')).toString().trim()
   } catch(e){}
 }
 const accounts = mnemonic ? {
   mnemonic,
 }: undefined

 let etherscanKey = process.env.ETHERSCANKEY
 if (!etherscanKey) {
   try {
     etherscanKey = fs.readFileSync(path.resolve(__dirname,
'.etherscanKey')).toString().trim()
   } catch(e){}
 }
```

Figure 11.1: `vault-v2/hardhat.config.ts#L67-L82`

Exploit Scenario

Alice, a member of the Yield team, has secrets stored in the process environment. Eve, an attacker, gains access to Alice's device and extracts the Infura and owner keys from it. Eve then launches a denial-of-service attack against the front end of the system and uses the owner key to steal the funds held by the owner on the mainnet.

Recommendations

Short term, to prevent attackers from accessing system funds, avoid using hard-coded secrets or storing secrets in the process environment.

Long term, use a hardware security module to ensure that keys can never be extracted.

12. Lack of limits on the total amount of collateral sold at auction

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-YP2-012
Target: `Witch.sol`	

Description

MakerDAO's Dutch auction system imposes limits on the amount of collateral that can be auctioned off at once (both the total amount and the amount of each collateral type). If the MakerDAO system experienced a temporary oracle failure, these limits would prevent a catastrophic loss of all collateral. The Yield Protocol auction system is similar to MakerDAO's but lacks such limits, meaning that all of its collateral could be auctioned off for below-market prices.

Exploit Scenario

The oracle price feeds (or other components of the system) experience an attack or another issue. The incident causes a majority of the vaults to become undercollateralized, triggering auctions of those vaults' collateral. The protocol then loses the majority of its collateral, which is auctioned off for below-market prices, and enters an undercollateralized state from which it cannot recover.

Recommendations

Short term, introduce global and type-specific limits on the amount of collateral that can be auctioned off at the same time. Ensure that these limits protect the protocol from total liquidation caused by bugs while providing enough liquidation throughput to accommodate all possible price changes.

Long term, wherever possible, introduce limits for the system's variables to ensure that they remain within the expected ranges. These limits will minimize the impact of bugs or attacks against the system.

13. Lack of incentives for calls to Witch.auction	
Severity: **Undetermined**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-YP2-013
Target: `Witch.sol`	

Description

Users call the `Witch` contract's `auction` function to start auctions for undercollateralized vaults. To reduce the losses incurred by the protocol, this function should be called as soon as possible after a vault has become undercollateralized. However, the Yield Protocol system does not provide users with a direct incentive to call `Witch.auction`. By contrast, the MakerDAO system provides rewards to users who initialize auctions.

Exploit Scenario

A stock market crash triggers a crypto market crash. The numerous corrective arbitrage transactions on the Ethereum network cause it to become congested, and gas prices skyrocket. To keep the Yield Protocol overcollateralized, many undercollateralized vaults must be auctioned off. However, because of the high price of calls to `Witch.auction`, and the lack of incentives for users to call it, too few auctions are timely started. As a result, the system incurs greater losses than it would have if more auctions had been started on time.

Recommendations

Short term, reward those who call `Witch.auction` to incentivize users to call the function (and to do so as soon as possible).

Long term, ensure that users are properly incentivized to perform all important operations in the protocol.

14. Contracts used as dependencies do not track upstream changes

Severity: **Low**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-YP2-014
Target: `yieldspace-v2/contracts/Math64x64.sol`	

Description

`Math64x64` has been copied and pasted into the `yieldspace-v2` repository. The code documentation does not specify the exact revision that was made or whether the code was modified. As such, the contracts will not reliably reflect updates or security fixes implemented in this dependency, as those changes must be manually integrated into the contracts.

Exploit Scenario

`Math64x64` receives an update with a critical fix for a vulnerability. An attacker detects the use of a vulnerable contract and can then exploit the vulnerability against any of the Yield Protocol contracts that use `Math64x64`.

Recommendations

Short term, review the codebase and document the source and version of the dependency. Include third-party sources as submodules in your Git repositories to maintain internal path consistency and ensure that any dependencies are updated periodically.

Long term, use an Ethereum development environment and NPM to manage packages in the project.

15. Cauldron's give and tweak functions lack vault existence checks

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-YP2-015
Target: `Cauldron.sol`	

Description

The `Cauldron` depends on the caller (the `Ladle`) to perform a check that is critical to the internal consistency of the `Cauldron`. The `Cauldron` should provide an API that makes the creation of malformed vaults impossible.

The `Cauldron` contract's `give(vaultId, receiver)` function does not check whether the `vaultId` passed to it is associated with an existent vault. If the ID is not that of an existent vault, the protocol will create a new vault, with the owner set to `receiver` and all other fields set to zero. The existence of such a malformed vault could have negative consequences for the protocol.

For example, the `build` function checks the existence of a vault by verifying that `vault.seriesId` is not zero. The `build` function could be abused to set the `seriesId` and `ilkId` of a malformed vault.

The `Cauldron`'s `tweak` function also fails to check the existence of the vault it operates on and can be used to create a vault without an owner.

```
    function _give(bytes12 vaultId, address receiver)
        internal
        returns(DataTypes.Vault memory vault)
    {
        require (vaultId != bytes12(0), "Vault id is zero");
        vault = vaults[vaultId];
        vault.owner = receiver;
        vaults[vaultId] = vault;
        emit VaultGiven(vaultId, receiver);
    }

    /// @dev Transfer a vault to another user.
    function give(bytes12 vaultId, address receiver)
        external
        auth
        returns(DataTypes.Vault memory vault)
    {
        vault = _give(vaultId, receiver);
    }
```

Figure 15.1: The `give` and `_give` functions in the `Cauldron` contract
(`vault-v2/contracts/Cauldron.sol#L228-L246`)

Exploit Scenario

Bob, a Yield Protocol developer, adds a new public function that calls `Cauldron.give` and does not perform a vault existence check. Any user can call the function to create a malformed vault, with unclear consequences for the protocol.

Recommendations

Short term, ensure that the protocol performs zero-value checks for all values that should not be set to zero.

Long term, follow the guidance on data validation laid out in TOB-YP2-016.

16. Problematic approach to data validation and access controls	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-YP2-016
Target: Throughout the code	

Description

Many parts of the codebase lack data validation. The Yield team indicated that these omissions were largely intentional, as it is the responsibility of the front end of other contracts to ensure that data is valid.

The codebase lacks zero-value checks for the following parameters (among others):

- The parameters of `LadleStorage.constructor`
- The `receiver` parameter of `Cauldron._give`
- The parameters of `Ladle.give`
- The `to` parameter of `Pool.buyBase` and `Pool.sellBase`
- The `oracle` parameter of `Cauldron.setLendingOracle` and `Cauldron.setSpotOracle`
- The `owner` parameter of `Cauldron.build`
- The `value` parameter of `FYToken.point`
- The parameters of `Witch.constructor`

It also lacks zero-value checks for the `module` parameter of `Ladle.moduleCall` and `Ladle.addModule`, and the `moduleCall` and `addModule` functions do not perform contract existence checks.

Moreover, many functions do not contain exhaustive data validation and instead rely on their caller or callee to partially handle data validation.

As a result, the protocol's data validation is spread across multiple functions and, in certain cases, across multiple contracts. For example, the `Cauldron` contract's `give` and `tweak` functions (likely among others) require the caller, which is usually the `Ladle`, to check the existence of the vault being modified. The `Ladle` is therefore responsible for ensuring the integrity of the `Cauldron`'s internal data.

This diffuse system of data validation requires developers and auditors to increase their focus on the context of a call, making their work more difficult. More importantly, though, it makes the code less robust. Developers cannot modify a function in isolation; instead, they have to look at all call sites to ensure that required validation is performed. This process is error-prone and increases the likelihood that high-severity issues (like that described in TOB-YP2-006) will be introduced into the system.

The deduplication of these checks (such that data validation occurs only once per call stack) is not a secure coding practice; nor is the omission of data validation. We strongly believe that code intended to securely handle millions of dollars in assets should be developed using the most secure coding practices possible.

The protocol's micro-optimizations do not appear to have any benefits beyond a reduction in gas costs. However, these savings are minor.

For example, performing a zero check of a value already on the stack would cost 3 units of gas (see the `ISZERO` opcode). Even with a fairly high gas price of 200 gwei and an ether price of $3,000, this operation would cost $0.0018. Performing 10 additional zero-value checks per transaction would cost only around 2 cents.

Similarly, a read of a value in "cold" storage would have a gas cost of 2,100 (see the `SLOAD` opcode); with the values above, that would be about $1.20. "Warm" access (that is, an additional read operation from the same storage slot within the same transaction) would cost only 100 units of gas, or about 6 cents.

We believe the low cost of these checks to be a reasonable price for the increased robustness that would accompany additional data validation. We do not agree that it is best to omit these checks, as the relatively small gas savings come at the expense of defense in depth, which is more important.

Exploit Scenario

A Yield Protocol developer adds a new function that calls a pre-existing function. This pre-existing function makes implicit assumptions about the data validation that will occur before it is called. However, the developer is not fully aware of these implicit assumptions and accidentally leaves out important data validation, creating an attack vector that can be used to steal funds from the protocol.

Recommendations

Long term, ensure that the protocol's functions perform exhaustive validation of their inputs and of the system's state and that they do not assume that validation has been performed further up in the call stack (or will be performed further down). Such assumptions make the code brittle and increase the likelihood that vulnerabilities will be introduced when the code is modified. Any implicit assumptions regarding data validation or access controls should be explicitly documented; otherwise, modifications to the code could break those important assumptions.

In general, a contract should not assume that its functions will always be called with valid data or that those calls will be made only when the state of the system allows it. This applies even to functions that can be called only by the protocol’s contracts, as a protocol contract could be replaced by a malicious version. An additional layer of defense could mitigate the fallout of such a scenario.

17. isContract may behave unexpectedly

Severity: **Informational**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-YP2-017
Target: `Router.sol`, `Timelock.sol`, `Relay.sol`, `PoolFactory.sol`	

Description

The Yield Protocol system relies on the `isContract()` function in a few of the Solidity files to check whether there is a contract at the target address. However, in Solidity, there is no general way to definitively determine that, as there are several edge cases in which the underlying function `extcodesize()` can return unexpected results. In addition, there is no way to guarantee that an address that *is* that of a contract (or one that is not) will remain that way in the future.

```
library IsContract {
  /// @dev Returns true if `account` is a contract.
  function isContract(address account) internal view returns (bool) {
      // This method relies on extcodesize, which returns 0 for contracts in
      // construction, since the code is only stored at the end of the
      // constructor execution.
      return account.code.length > 0;
  }
}
```

Figure 17.1: `yield-utils-v2/contracts/utils/IsContract.sol#L6-L14`

Exploit Scenario

A function, *f*, within the Yield Protocol codebase calls `isContract()` internally to guarantee that a certain method is not callable by another contract. An attacker creates a contract that calls *f* from within its constructor, and the call to `isContract()` within *f* returns `false`, violating the "guarantee."

Recommendations

Short term, clearly document for developers that `isContract()` is not guaranteed to return an accurate value, and emphasize that it should never be used to provide an assurance of security.

Long term, be mindful of the fact that the Ethereum core developers consider it poor practice to attempt to differentiate between end users and contracts. Try to avoid this practice entirely if possible.

18. Use of multiple repositories

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-YP2-018
Target: Throughout the Codebase	

Description

The Yield Protocol code is spread across four repositories. These repositories are tightly coupled, and the code is broken up somewhat arbitrarily. This makes it more difficult to navigate the codebase and to obtain a complete picture of the code that existed at any one time. It also makes it impossible to associate one version of the protocol with one commit hash. Instead, each version requires four commit hashes.

Exploit Scenario

The master branch of one repository of the protocol is not compatible with the master branch of another. The contracts' incompatibility leads to problems when a new version of the protocol is deployed.

Recommendations

To maintain one canonical version of the protocol, avoid using multiple repositories for the contracts.

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 of users 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	Breach of the confidentiality or integrity of data
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	System failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions, locking, 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 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 client has indicated is important.
Medium	Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.
High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

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

Categories	Description
Access Controls	The authentication and authorization of components
Arithmetic	The proper use of mathematical operations and semantics
Assembly Use	The use of inline assembly
Centralization	The existence of a single point of failure
Upgradeability	Contract upgradeability
Function Composition	The separation of the logic into functions with clear purposes
Front-Running	Resistance to front-running
Key Management	The existence of proper procedures for key generation, distribution, and access
Monitoring	The use of events and monitoring procedures
Specification	The comprehensiveness and readability of codebase documentation and specification
Testing and Verification	The use of testing techniques (e.g., unit tests and fuzzing)

Rating Criteria	
Rating	Description
Strong	The control was robust, documented, automated, and comprehensive.
Satisfactory	With a few minor exceptions, the control was applied consistently.
Moderate	The control was applied inconsistently in certain areas.
Weak	The control was applied inconsistently or not at all.
Missing	The control was missing.
Not Applicable	The control is not applicable.
Not Considered	The control was not reviewed.
Further Investigation Required	The control requires further investigation.

C. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. An up-to-date version of the checklist can be found in crytic/building-secure-contracts.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow this checklist, use the below output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of
Echidna and Manticore
```

General Security Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on blockchain-security-contacts.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.

- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.
- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.
- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

D. Whitepaper Variable Representations

The YieldSpace whitepaper defines a series of mathematical variables used throughout the derivations. Trail of Bits recommends revising the whitepaper to include a section that explicitly defines all variables, which would help readers understand the formulas more easily.

To assist in this process, Trail of Bits compiled the following list of variables and their representations in the whitepaper.

x	Reserves of the base token (e.g., DAI)
y	Reserves of the fyToken (e.g., fyDAI)
t	The time to maturity, normalized as $0 \leq t \leq 1$
a	x^{1-t}
b	y^{1-t}
k, a system invariant	$a + b = x^{1-t} + y^{1-t}$
r, marginal interest	$\frac{y}{x} - 1$

E. Code Quality Recommendations

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

General Recommendations

- **Address the warnings emitted by the test suite to ensure that it is not affected by JavaScript bindings.**

```
Creating Typechain artifacts in directory typechain for target ethers-v5
Successfully generated Typechain artifacts!
Secp256k1 bindings are not compiled. Pure JS implementation will be used.
Keccak bindings are not compiled. Pure JS implementation will be used.
```

Figure E.1: Test suite output

`yield-utils-v2`

- **Remove the redundant `account` parameter in `renounceRole`.**

```
function renounceRole(bytes4 role, address account) external virtual {
    require(account == msg.sender, "Renounce only for self");

    _revokeRole(role, account);
}
```

Figure E.2: `yield-utils-v2/contracts/access/AccessControl.sol#L223-L227`

- **Rename the `getRoleAdmin` function in the `AccessControls` contract to `getAdminRole` to make it clear that the function returns the admin of a role.**
- **Add timestamps to the `ERC20Rewards` contract to ensure that users and off-chain monitoring systems can determine the exact time of any reward token updates.**

`yieldspace-v2`

- **Rename the yx variable in the `YieldMath` contract (line 431) to `future fyTokenAmount` to reflect the fact that `fyAmount` is used in a calculation.**

`vault-v2`

`Cauldron.sol`

- **Rename the `addAsset` function to indicate that it can both add and remove assets.** Also rename the `AssetAdded` event emitted in this code to enhance the code's readability.
- **Replace checks like `require(a == true)` by `require(a)` to enhance the code's readability (line 173).**
- **Move the `_level` function closer to the `level` function to increase the code's readability.**
- **Explicitly initialize all variables to 0 (line 151).**
- **Remove the redundant calculations shown in figure E.3 and calculate the ratio required for the final calculation in the `if` block of the `_level` function.**

```
    /// @dev Return the collateralization level of a vault. It will be negative if
undercollateralized.
    function _level(
        DataTypes.Vault memory vault_,
        DataTypes.Balances memory balances_,
        DataTypes.Series memory series_
    )
        internal
        returns (int256)
    {
        DataTypes.SpotOracle memory spotOracle_ =
spotOracles[series_.baseId][vault_.ilkId];
        uint256 ratio = uint256(spotOracle_.ratio) * 1e12;   // Normalized to 18
decimals
        (uint256 inkValue,) = spotOracle_.oracle.get(vault_.ilkId, series_.baseId,
balances_.ink);    // ink * spot

        if (uint32(block.timestamp) >= series_.maturity) {
            uint256 accrual_ = _accrual(vault_.seriesId, series_);
            return inkValue.i256() -
uint256(balances_.art).wmul(accrual_).wmul(ratio).i256();
        }

        return inkValue.i256() - uint256(balances_.art).wmul(ratio).i256();
```

Figure E.3: `vault-v2/contracts/Cauldron.sol#L476-L495`

`Ladle.sol`

- **Rename the `addIntegration` and `addToken` functions to `toggleIntegration` and `toggleToken`, respectively.** The current naming scheme does not reflect their incoming arguments, as the functions actually indicate whether integrations and tokens have been added.
- **Rename `getVault` to `getVaultAndCheckOwner`. The latter name will reflect the access controls on the execution of the function.**

- **Remove redundant ternary operators like `(join != IJoin(address(0))) ? true : false`, which can be simplified to `join != IJoin(address(0))`.** (See line 119.)

`Join.sol`

- **Clearly document the purpose of all unchecked portions of the code (line 60).**

`Witch.sol`

- **Rename the `inkPrice` function to `artPrice` or `inkPerArt` to more accurately reflect the value it represents.**

F. Fix Log

On October 18, 2021, Trail of Bits reviewed the fixes and mitigations implemented by the Yield team for the issues identified in this report. The Yield team fixed three of the issues reported in the original assessment, partially fixed three more, and acknowledged but did not fix the other twelve. We reviewed each of the fixes to ensure that the proposed remediation would be effective. For additional information, please refer to the Detailed Fix Log.

ID	Title	Severity	Fix Status
1	Lack of contract existence check on delegatecall may lead to unexpected behavior	High	Partially fixed (`e46a3ae`)
2	Use of delegatecall in a payable function inside a loop	High	Partially fixed (`e46a3ae`)
3	Lack of two-step process for critical operations	Medium	Not fixed
4	Risks associated with use of ABIEncoderV2	Undetermined	Not fixed
5	Project dependencies contain vulnerabilities	Medium	Not fixed
6	Witch's buy and payAll functions allow users to buy collateral from vaults not undergoing auctions	High	Fixed (`698ff84`)
7	Solidity compiler optimizations can be problematic	Informational	Not fixed
8	Risks associated with EIP-2612	Informational	Not fixed
9	Failure to use the batched transaction flow may enable theft through front-running	High	Not fixed
10	Strategy contract's balance-tracking system could facilitate theft	High	Partially fixed (`1a9db0a`)

11	Insufficient protection of sensitive keys	Medium	Not fixed
12	Lack of limits on the total amount of collateral sold at auction	Medium	Fixed (57c238b)
13	Lack of incentives for calls to Witch.auction	Undetermined	Not fixed
14	Contracts used as dependencies do not track upstream changes	Low	Not fixed
15	Cauldron’s give and tweak functions lack vault existence checks	Informational	Fixed (efebd72)
16	Problematic approach to data validation and access controls	Informational	Not fixed
17	isContract may behave unexpectedly	Informational	Not fixed
18	Use of multiple repositories	Informational	Not fixed

Detailed Fix Log

TOB-YP2-001: Lack of contract existence check on delegatecall may lead to unexpected behavior
Partially fixed. The Yield team documented the fact that addModule should be called only by admins and only with addresses of contracts that cannot self-destruct. Assuming this recommendation is followed, this prevents the moduleCall being used on these contracts, due to checks in moduleCall. However, the commit did not add a contract existence check before the delegatecall in the moduleCall. (e46a3ae)

TOB-YP2-002: Use of delegatecall in a payable function inside a loop
Partially fixed. The Yield documentation now indicates that because of the batched transaction flow, modules cannot use msg.value. (e46a3ae)

TOB-YP2-006: Witch's buy and payAll functions allow users to buy collateral from vaults not undergoing auctions
Fixed. The Yield team added checks to ensure that collateral can be bought only from vaults that are undergoing auctions. (698ff84)

TOB-YP2-010: Strategy contract's balance-tracking system could facilitate theft
Partially fixed. The Yield team added docstrings explaining the caveats regarding the reward token system but did not add automated checks of the reward token. (1a9db0a)

TOB-YP2-012: Lack of limits on the total amount of collateral sold at auction
Fixed. The Yield team introduced type-specific limits on the total amount of collateral that can be auctioned off at once as well as unit tests for the limits. (57c238b)

TOB-YP2-015: Cauldron's give and tweak functions lack vault existence checks
Fixed. The Yield team added vault existence checks to the _give, _tweak, and _destroy functions, as well as tests for these checks. (efebd72)