



# Yield Protocol

## Security Assessment

August 21, 2020

Prepared For:  
Allan Niemerg | *Yield*  
[allan@yield.is](mailto:allan@yield.is)

Prepared By:  
Gustavo Grieco | *Trail of Bits*  
[gustavo.grieco@trailofbits.com](mailto:gustavo.grieco@trailofbits.com)

Michael Colburn | *Trail of Bits*  
[michael.colburn@trailofbits.com](mailto:michael.colburn@trailofbits.com)

[Executive Summary](#)

[Project Dashboard](#)

[Code Maturity Evaluation](#)

[Engagement Goals](#)

[Coverage](#)

[Automated Testing and Verification](#)

[System properties](#)

[General properties](#)

[ABDK arithmetic properties](#)

[YieldMath properties](#)

[Recommendations Summary](#)

[Short term](#)

[Long term](#)

[Findings Summary](#)

- [1. Flash minting can be used to redeem fyDAI](#)
- [2. Permission-granting is too simplistic and not flexible enough](#)
- [3. pot.chi\(\) value is never updated](#)
- [4. Lack of validation when setting the maturity value](#)
- [5. Delegates can be added or removed repeatedly to bloat logs](#)
- [6. Withdrawing from the Controller allows accounts to contain dust](#)
- [7. Solidity compiler optimizations can be dangerous](#)
- [8. Lack of chainID validation allows signatures to be re-used across forks](#)
- [9. Permit opens the door for griefing contracts that interact with the Yield Protocol](#)
- [10. Pool initialization is unprotected](#)
- [11. Computation of DAI/fyDAI to buy/sell is imprecise](#)

[A. Vulnerability Classifications](#)

[B. Code Maturity Classifications](#)

[C. Code Quality Recommendations](#)

[General](#)

[Controller](#)

[Liquidations](#)

[D. Fix Log](#)

[Detailed fix log](#)

## Executive Summary

From August 3 through August 21, 2020, Yield engaged Trail of Bits to review the security of the Yield Protocol. Trail of Bits conducted this assessment over the course of six person-weeks with two engineers working from commit hash [4422fda](#) from the [yieldprotocol/fyDai](#) repository.

**Week one:** We familiarized ourselves with the codebase and whitepapers. We also began checking for common Solidity flaws and identifying areas that would benefit from tool-assisted analysis.

**Week two:** We continued manual review of the various Yield Protocol contracts, focusing on interactions between the different contracts as well as with the external MakerDAO system. We also began to develop properties for Echidna.

**Final week:** As we concluded our manual review, we focused on the custom arithmetic libraries, the Pool market maker, and Unwind contracts, and finalized the set of properties that were tested.

Our review resulted in 11 findings ranging from high to informational severity. Interestingly, the issues we found do not have any particularity in common: They affect a variety of different areas, but most of them allow us to break some internal invariants, e.g., the redemption of more fyDAI tokens than expected ([TOB-YP-001](#)), the use of invalid maturity values ([TOB-YP-004](#)), or only dust amounts of assets remaining in the controller accounts ([TOB-YP-006](#)). We also make several code quality recommendations in [Appendix C](#).

During the assessment, Yield provided fixes for issues when possible. Trail of Bits verified the fixes for [TOB-YP-002](#), [TOB-YP-005](#), and [TOB-YP-006](#), as well as a partial fix for [TOB-YP-001](#).

Overall, the code follows a high-quality software development standard and best practices. It has suitable architecture and is properly documented. The interactions between components are well-defined. The functions are small, with a clear purpose.

Trail of Bits recommends addressing the findings presented and integrating the property-based testing into the codebase. Yield must be careful with the deployment of the contracts and the interactions of its early users and their advantages. Finally, we recommend performing an economic assessment to make sure the monetary incentives are properly designed.

*Update: On September 14, 2020, Trail of Bits reviewed fixes proposed by Yield for the issues presented in this report. See a detailed review of the current status of each issue in [Appendix D](#).*

*The name of the yDAI token was changed to fyDAI subsequent to our assessment but prior to the finalization of this report. The report has been modified such that all references to the “yDAI” token were replaced with “fyDAI”. However, all references to source code artifacts (e.g., smart contract names such as YDai) remain as they were in the assessed version of the codebase.*

# Project Dashboard

## Application Summary

Name	Yield Protocol
Version	<a href="#">4422fda</a>
Type	Solidity
Platforms	Ethereum

## Engagement Summary

Dates	August 3–August 21, 2020
Method	Whitebox
Consultants Engaged	2
Level of Effort	6 person-weeks

## Vulnerability Summary

Total High-Severity Issues	1	■
Total Medium-Severity Issues	1	■
Total Low-Severity Issues	5	■■■■■
Total Informational-Severity Issues	2	■■
Total Undetermined-Severity Issues	2	■■
Total	11	

## Category Breakdown

Undefined Behavior	2	■■
Access Controls	3	■■■
Data Validation	4	■■■■
Auditing and Logging	1	■
Timing	1	■
Total	11	

## Code Maturity Evaluation

In the table below, we review the maturity of the codebase and the likelihood of future issues. In each area of control, we rate the maturity from strong to weak, or missing, and give a brief explanation of our reasoning.

Category Name	Description
Access Controls	<b>Satisfactory.</b> Appropriate access controls were in place for performing privileged operations.
Arithmetic	<b>Satisfactory.</b> The contracts included use of safe arithmetic and casting functions. No potential overflows were possible in areas where these functions were not used.
Assembly Use	<b>Strong.</b> The contracts only used assembly to fetch the <code>chainID</code> for ERC2612 permit functionality.
Centralization	<b>Satisfactory.</b> While the protocol relied on an owner to correctly deploy the initial contracts, ownership could be renounced later and users would verify this using on-chain events.
Contract Upgradeability	<b>Not Applicable.</b> The contracts contained no upgradeability mechanisms.
Function Composition	<b>Strong.</b> Functions and contracts were organized and scoped appropriately.
Front-Running	<b>Satisfactory.</b> Although some functionality could have been affected by front-running attacks, the impact was low.
Monitoring	<b>Satisfactory.</b> The events produced by the smart contract code were sufficient to monitor on-chain activity.
Specification	<b>Satisfactory.</b> White papers describing the functionality of the protocol and accompanying pool were available. The contract source code included NatSpec comments for all contracts and functions.
Testing & Verification	<b>Moderate.</b> While the contracts included a large number of unit tests, the testing did not include any use of automatic tools such as fuzzers.

## Engagement Goals

The engagement was scoped to provide a security assessment of Yield Protocol smart contracts in the [yieldprotocol/fyDAI](https://github.com/yieldprotocol/fyDAI) repository.

Specifically, we sought to answer the following questions:

- Are appropriate access controls set for the user and the smart contract interactions?
- Does arithmetic regarding token minting, burning, and pool operations hold?
- Is there any arithmetic overflow or underflow affecting the code?
- Can participants manipulate or block tokens or pool operations?
- Is it possible to manipulate the pools by front-running transactions?
- Is it possible for participants to steal or lose tokens?
- Can participants perform denial-of-service or phishing attacks against any of the components?

## Coverage

**Controller.** The Controller contract contains the main business logic and acts as the entry point for users within the Yield Protocol. It allows users to manage collateral and debt levels. We manually reviewed the contract's interactions with the MakerDAO system to ensure proper behavior. We also used property-based testing tools to make sure its invariants held and users were able to perform operations with the contract without unexpected reverts.

**YDai.** The YDai contract implements an ERC20 token that allows a user to mint tokens by locking up their Dai until a fixed maturity date. These tokens can then be bought or sold to other users and later redeemed for Dai. This contract also implements a standard ERC20 token. We verified that all of the expected ERC20 properties hold. Additionally, we conducted a manual review to ensure the flash-minting feature cannot be abused to manipulate the protocol's expected behavior.

**Treasury.** The Treasury contract manages asset transfers between all contracts in the Yield Protocol and other external contracts such as Chai and MakerDAO. Since users do not use the Treasury contract directly, we manually reviewed all of its interactions with other smart contracts of the protocol as well as its access control system to make sure external users cannot interfere with it.

**Liquidations.** The Liquidations contract allows liquidation of undercollateralized vaults using a reverse Dutch auction mechanism. We manually reviewed exactly how and when

each user could be liquidated by any other user, and how the Liquidations contract interacts with the rest of the system.

**Unwind.** The Unwind contract allows users to recover their assets from the Yield Protocol in the event of a MakerDAO shutdown. We manually reviewed this contract to ensure that it can only be used after the shutdown and that users will receive their corresponding collateral.

**Pool.** The Pool contract implements an automatic market maker that exchanges DAI for fyDAI at a price defined by a specific formula that also incorporates time to maturity. We manually reviewed this contract for common flaws affecting exchanges, including incorrect price computation, market manipulation, and front-running.

**Access controls.** Many parts of the system expose privileged functionality, such as setting protocol parameters or minting/burning tokens. We reviewed these functions to ensure they can only be triggered by the intended actors and that they do not contain unnecessary privileges that may be abused.

**Arithmetic.** We reviewed calculations for logical consistency, as well as rounding issues and scenarios where reverts due to overflow may negatively impact use of the protocol.

During the course of the assessment the Yield Protocol team made several pull requests that we also reviewed in addition to the version listed in the Project Dashboard: [246](#), [247](#), [251](#), [252](#), [253](#), [254](#), [268](#), [271](#), and [279](#).

Contracts located in the external, mocks, and peripheral directories were out of scope for this review.



## Automated Testing and Verification

To enhance coverage of certain areas of the contracts, Trail of Bits used automated testing techniques, including:

- [Slither](#), a Solidity static analysis framework. Slither can statically verify algebraic relationships between Solidity variables. We used Slither to detect common flaws across the entire codebase.
- [Echidna](#), a smart contract fuzzer. Echidna can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to test the expected system properties of the Controller contract and its dependencies.
- [Manticore](#), a symbolic execution framework. Manticore can exhaustively test security properties via symbolic execution.

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations:

- Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode.
- Echidna may not randomly generate an edge case that violates a property.
- Manticore may fail to complete its analysis.

To mitigate these risks, we generate 50,000 test cases per property with Echidna, run Manticore for a minimum of one hour, and then manually review all results.

### System properties

System properties can be broadly divided into two categories: general properties of the contracts that state what users can and cannot do, and arithmetic properties for the ABDK and the YieldMath libraries.

Additionally, properties can have three outcomes: Either the verification fails (and we list the corresponding issue), it passes after 50,000 Echidna tests, or it's formally verified using Manticore.

#### General properties

#	Property	Result
1	Calling <code>erase</code> in the Controller never reverts.	PASSED
2	Calling <code>locked</code> in the Controller never reverts.	PASSED

3	Calling <code>powerOf</code> in the Controller never reverts.	PASSED
4	Calling <code>totalDebtDai</code> in the Controller never reverts.	PASSED
5	Posting, borrowing, repaying, and withdrawing using CHAI as collateral properly updates the state variables.	PASSED
6	Posting, borrowing, repaying, and withdrawing using WETH as collateral properly updates the state variables.	PASSED
7	All the WETH balances are above dust or zero in the Controller.	<b>FAILED</b> ( <a href="#">TOB-YP-006</a> )
8	All the WETH balances are above dust or zero in the Liquidations.	PASSED
9	Calling <code>price</code> never reverts on Liquidations	PASSED
10	Transferring tokens to the null address ( <code>0x0</code> ) causes a revert.	PASSED
11	The null address ( <code>0x0</code> ) owns no tokens.	PASSED
12	Transferring a valid amount of tokens to a non-null address reduces the current balance.	PASSED
13	Transferring an invalid amount of tokens to a non-null address reverts or returns false.	PASSED
14	Self-transferring a valid amount of tokens keeps the current balance constant.	PASSED
15	Approving overwrites the previous allowance value.	PASSED
16	The balances are consistent with the <code>totalSupply</code> .	PASSED
17	Burning all the balance of a user resets it zero.	PASSED
18	Burning more than the balance of a user reverts.	PASSED

### ABDK arithmetic properties

#	Property	Result
1	Addition is associative.	VERIFIED
2	Zero is the identity element in addition.	VERIFIED
3	Zero is the identity element in subtraction.	VERIFIED

4	Subtracting a number from itself is zero.	VERIFIED
5	Negation operation is the same as subtracting from zero.	VERIFIED
6	Negation operation is inverse to itself.	VERIFIED
7	One is the identity element in multiplication.	PASSED
8	Zero is the absorbing element in multiplication.	PASSED
9	Square root is the inverse of multiplying a number by itself.	PASSED
10	Multiplication and addition give consistent results.	PASSED

### YieldMath properties

#	Property	Result
1	yDaiOutForDaiIn and daiInForYDaiOut are inverse functions.	<b>FAILED</b> ( <a href="#">TOB-YP-011</a> )
2	daiOutForYDaiIn and yDaiInForDaiOut are inverse functions.	<b>FAILED</b> ( <a href="#">TOB-YP-011</a> )

## 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

- ❑ **Disallow calls to redeem in the YDai and Unwind contracts during flash minting.** This will prevent users from abusing the flash minting feature. ([TOB-YP-001](#))
- ❑ **Rewrite the authorization system to allow only certain addresses to access certain functions.** This will increase users' confidence in the deployment of the contracts. ([TOB-YP-002](#))
- ❑ **Add a call to `pot.drip` every time the `pot.chi` is used.** This will ensure that users receive the correct amount of interest after maturation. ([TOB-YP-003](#))
- ❑ **Add checks to the YDai contract constructor to ensure maturity timestamps fall within an acceptable range.** This will prevent maturity dates from being mistakenly set in the past or too far in the future. ([TOB-YP-004](#))
- ❑ **Add a require statement to check that the delegate address is not already enabled or disabled for the user.** This will ensure log messages are only emitted when a delegate is activated or deactivated. ([TOB-YP-005](#))
- ❑ **Enforce the `aboveDustOrZero` function in the `from` address instead of the `to` address, after modifying its balance during the `withdraw` call.** This will ensure the correct address has an appropriate balance after calls to `withdraw`. ([TOB-YP-006](#))
- ❑ **Measure the gas savings from optimizations,** and carefully weigh them against the possibility of an optimization-related bug. ([TOB-YP-007](#))
- ❑ **Include the `chainID` opcode in the `permit` schema.** This will make replay attacks impossible in the event of a post-deployment hard fork. ([TOB-YP-008](#))
- ❑ **Properly document the possibility of griefing `permit` calls to warn users interacting with `fyDAI` tokens.** This will allow users to anticipate this possibility and develop alternate workflows in case they are targeted by it. ([TOB-YP-009](#))

❑ **Consider restricting calls to `init` to the contract owner and enforce that it can only be called once.** This will ensure initialization is carried out as Yield intends. ([TOB-YP-010](#))

❑ **Review the specification of the `YieldMath` functions and make sure it matches the implementation.** Use Echidna to validate the implementation. ([TOB-YP-011](#))

## Long term

❑ **Do not include operations that allow any user to manipulate an arbitrary amount of funds, even if it is in a single transaction.** This will prevent attackers from gaining leverage to manipulate the market and break internal invariants. ([TOB-YP-001](#))

❑ **Review the rest of the components to make sure they are suitable for their purpose and can be used only for their intended purpose.** ([TOB-YP-002](#)), ([TOB-YP-010](#))

❑ **Review every interaction with the MakerDAO contracts to make sure your code will work as expected.** ([TOB-YP-003](#))

❑ **Always perform validation of parameters passed to contract constructors.** This will help detect and prevent errors during deployment. ([TOB-YP-004](#))

❑ **Review all operations and avoid emitting events in repeated calls to idempotent operations.** This will help prevent bloated logs. ([TOB-YP-005](#))

❑ **Use Echidna or Manticore to properly test the contract invariants.** Automated testing can cover a wide array of inputs that unit testing may miss. ([TOB-YP-006](#))

❑ **Monitor the development and adoption of Solidity compiler optimizations.** This will allow you to assess their maturity and whether they are appropriate to enable. ([TOB-YP-007](#))

❑ **Document and carefully review any signature schemas, including their robustness to replay on different wallets, contracts, and blockchains.** Make sure users are aware of signing best practices and the danger of signing messages from untrusted sources. ([TOB-YP-008](#))

❑ **Carefully monitor the blockchain to detect front-running attempts.** ([TOB-YP-009](#))

❑ **Develop robust unit and automated test suites for the custom math functions.** This will help ensure the correct functionality of this complex arithmetic. ([TOB-YP-011](#))

## Findings Summary

#	Title	Type	Severity
1	<a href="#">Flash minting can be used to redeem fyDAI</a>	Undefined Behavior	Medium
2	<a href="#">Permission-granting is too simplistic and not flexible enough</a>	Access Controls	Low
3	<a href="#">pot.chi() value is never updated</a>	Data Validation	Low
4	<a href="#">Lack of validation when setting the maturity value</a>	Data Validation	Low
5	<a href="#">Delegates can be added or removed repeatedly to bloat logs</a>	Auditing and Logging	Informational
6	<a href="#">Withdrawing from the controller allows accounts to contain dust</a>	Data Validation	Low
7	<a href="#">Solidity compiler optimizations can be dangerous</a>	Undefined Behavior	Undetermined
8	<a href="#">Lack of chainID validation allows signatures to be re-used across forks</a>	Access Controls	High
9	<a href="#">Permit opens the door for griefing contracts that interact with the Yield Protocol</a>	Timing	Informational
10	<a href="#">Pool initialization is unprotected</a>	Access Controls	Low
11	<a href="#">Computation of DAI/fyDAI to buy/sell is imprecise</a>	Data Validation	Undetermined

## 1. Flash minting can be used to redeem fyDAI

Severity: Medium

Type: Undefined Behavior

Target: YDai.sol, Unwind.sol

Difficulty: Medium

Finding ID: TOB-YP-001

### Description

The flash-minting feature from the fyDAI token can be used to redeem an arbitrary amount of funds from a mature token.

The fyDAI token has a special function that allows users to mint and burn an arbitrary amount of tokens in a single transaction:

```
/// @dev Flash-mint yDai. Calls back on `IFlashMinter.executeOnFlashMint()`
/// @param to Wallet to mint the yDai in.
/// @param yDaiAmount Amount of yDai to mint.
/// @param data User-defined data to pass on to `executeOnFlashMint()`
function flashMint(address to, uint256 yDaiAmount, bytes calldata data) external override
{
    _mint(to, yDaiAmount);
    IFlashMinter(msg.sender).executeOnFlashMint(to, yDaiAmount, data);
    _burn(to, yDaiAmount);
}
```

Figure 1.1: flashMint function in YDai.sol.

This function allows an arbitrary contract to be called with the executeOnFlashMint interface. This arbitrary contract can then call any function. In particular, it can call redeem from the same contract if the token is mature:

```
function redeem(address from, address to, uint256 yDaiAmount)
    public onlyHolderOrDelegate(from, "YDai: Only Holder Or Delegate") {
    require(
        isMature == true,
        "YDai: yDai is not mature"
    );
    _burn(from, yDaiAmount); // Burn yDai from `from`
    uint256 daiAmount = muld(yDaiAmount, chiGrowth()); // User gets interest for
holding after maturity
    _treasury.pullDai(to, daiAmount); // Give dai to `to`, from
Treasury
    emit Redeemed(from, to, yDaiAmount, daiAmount);
}
```

```
}
```

Figure 1.2: redeem function in YDai.sol.

The same transaction can also pull an arbitrary number of funds from the treasure (if available), which can be deposited to mint fyDAI tokens again.

Additionally, this attack could also target the redeem function in the Unwind contract in the event of a MakerDAO shutdown:

```
/// @dev Redeems YDai for weth for any user. YDai.redeem won't work if MakerDAO is in
shutdown.
/// @param maturity Maturity of an added series
/// @param user Wallet containing the yDai to burn.
function redeem(uint256 maturity, address user) public {
    require(settled && cashedOut, "Unwind: Not ready");
    IYDai yDai = _controller.series(maturity);
    uint256 yDaiAmount = yDai.balanceOf(user);
    yDai.burn(user, yDaiAmount);
    require(
        _weth.transfer(
            user,
            daiToFixWeth(muld(yDaiAmount, yDai.chiGrowth()), _fix)
        )
    );
}
```

Figure 1.3: redeem function in Unwind.sol.

### Exploit Scenario

Eve calls `flashMint` on a YDai contract that has already matured and mints a large quantity of tokens to a contract she controls. This contract's `executeOnFlashMint` hook in turn calls `redeem` in the matured YDai contract, and Eve's contract receives a large quantity of Dai. Eve's contract may now negatively impact markets to her advantage.

### Recommendation

Short term, disallow calls to `redeem` in the YDai and Unwind contracts during flash minting.

Long term, do not include operations that allow any user to manipulate an arbitrary amount of funds, even if it is in a single transaction. This will prevent attackers from gaining leverage to manipulate the market and break internal invariants.



## 2. Permission-granting is too simplistic and not flexible enough

Severity: Low

Type: Access Controls

Target: `Orchestrated.sol`

Difficulty: High

Finding ID: TOB-YP-002

### Description

The Yield Protocol contracts implement an oversimplified permission system that can be abused by the administrator.

The Yield Protocol implements several contracts that need to call privileged functions from each other. For instance, only the borrow function in Controller can call the mint function in YDai:

```
function borrow(bytes32 collateral, uint256 maturity, address from, address to, uint256
yDaiAmount)
    public override
    validCollateral(collateral)
    validSeries(maturity)
    onlyHolderOrDelegate(from, "Controller: Only Holder Or Delegate")
    onlyLive
{
    IYDai yDai = series[maturity];
    debtYDai[collateral][maturity][from] =
debtYDai[collateral][maturity][from].add(yDaiAmount);

    require(
        isCollateralized(collateral, from),
        "Controller: Too much debt"
    );

    yDai.mint(to, yDaiAmount);
    emit Borrowed(collateral, maturity, from, toInt256(yDaiAmount));
}
```

Figure 2.1: borrow function in Controller.sol.

```
/// @dev Mint yDai. Only callable by Controller contracts.
/// This function can only be called by other Yield contracts, not users directly.
/// @param to Wallet to mint the yDai in.
/// @param yDaiAmount Amount of yDai to mint.
function mint(address to, uint256 yDaiAmount) public override onlyOrchestrated("YDai: Not
```

```

Authorized") {
    _mint(to, yDaiAmount);
}

```

Figure 2.2: mint function in YDai.sol.

For implementing permissions, there is a special function called orchestrate which allows certain addresses to be added into the list of authorized users:

```

contract Orchestrated is Ownable {
    event GrantedAccess(address access);
    mapping(address => bool) public authorized;
    constructor () public Ownable() {}

    /// @dev Restrict usage to authorized users
    modifier onlyOrchestrated(string memory err) {
        require(authorized[msg.sender], err);
        _;
    }

    /// @dev Add user to the authorized users list
    function orchestrate(address user) public onlyOwner {
        authorized[user] = true;
        emit GrantedAccess(user);
    }
}

```

Figure 2.2: Orchestrated contract.

However, there is no way to specify which operation can be called for every privileged user. All the authorized addresses can call any restricted function, and the owner can add any number of them. Also, the privileged addresses are supposed to be smart contracts; however, there is no check for that. Moreover, once an address is added, it cannot be deleted.

### Exploit Scenario

Eve gains access to the owner's private key and uses it to call the orchestrate function with an additional address to backdoor one of the contracts. As a result, any user interacting with the contracts is advised to review the authorized mapping to make sure the contracts don't allow additional addresses to call restricted functions.

### Recommendation

Short term, rewrite the authorization system to allow only certain addresses to access certain functions (e.g., the minter address can only call mint in YDai).

Long term, review the rest of the components to make sure they are suitable for their purpose and can be used only for their intended purpose.

### 3. pot.chi() value is never updated

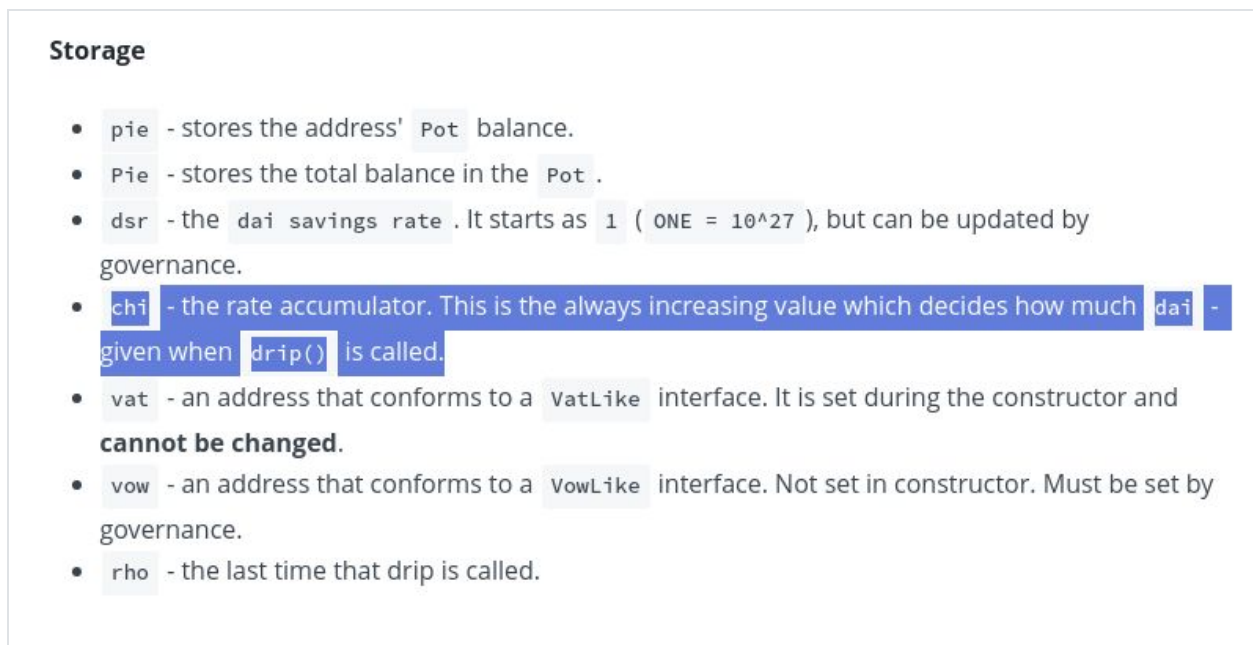
Severity: Low  
Type: Data Validation  
Target: YDai.sol

Difficulty: High  
Finding ID: TOB-YP-003

#### Description

The Yield contracts interact with the Dai Savings Rate (DSR) contracts from MakerDAO to obtain the rate accumulator value without properly calling a function to update its value.

DSR works using the pot contracts from MakerDAO. Once these contracts are deployed, they require the drip function to be called in order to update the accumulated interest rate:



**Storage**

- `pie` - stores the address' `Pot` balance.
- `Pie` - stores the total balance in the `Pot`.
- `dsr` - the `dai savings rate`. It starts as `1` (`ONE = 1027`), but can be updated by governance.
- `chi` - the rate accumulator. This is the always increasing value which decides how much `dai` given when `drip()` is called.
- `vat` - an address that conforms to a `VatLike` interface. It is set during the constructor and **cannot be changed**.
- `vow` - an address that conforms to a `VowLike` interface. Not set in constructor. Must be set by governance.
- `rho` - the last time that drip is called.

Figure 3.1: pot documentation at MakerDAO.

The Yield Protocol uses DSR. In particular, YDai uses the pot contracts directly to provide interest to its users:

```
/// @dev Mature yDai and capture chi and rate
function mature() public override {
    require(
        // solium-disable-next-line security/no-block-members
        now > maturity,
        "YDai: Too early to mature"
    );
    require(
```

```

        isMature != true,
        "YDai: Already matured"
    );
    (, rate0,,) = _vat.ilks(WETH); // Retrieve the MakerDAO Vat
    rate0 = Math.max(rate0, UNIT); // Floor it at 1.0
    chi0 = _pot.chi();
    isMature = true;
    emit Matured(rate0, chi0);
}

```

*Figure 3.1: mature function in YDai.*

However, the drip function is never called on any contract. It could be called manually by the users or the Yield off-chain components; however, this was not documented.

### **Exploit Scenario**

Alice locks DAI in a fyDAI token expecting to obtain a certain interest rate. However, the call to drip is never performed, so Alice obtains less interest than expected after the fyDAI token matures.

### **Recommendation**

Short term, add a call to `pot.drip` every time the `pot.chi` is used. This will ensure that users receive the correct amount of interest after maturation.

Long term, review every interaction with the MakerDAO contracts to make sure your code works as expected.

## 4. Lack of validation when setting the maturity value

Severity: Low  
Type: Data Validation  
Target: YDai.sol

Difficulty: Low  
Finding ID: TOB-YP-004

### Description

When a fyDAI contract is deployed, one of the deployment parameters is a maturity date, passed as a Unix timestamp. This is the date at which point fyDAI tokens can be redeemed for the underlying Dai. Currently, the contract constructor performs no validation on this timestamp to ensure it is within an acceptable range. As a result, it is possible to mistakenly deploy a YDai contract that has a maturity date in the past or many years in the future, which may not be immediately noticed.

```
/// @dev The constructor:
/// Sets the name and symbol for the yDai token.
/// Connects to Vat, Jug, Pot and Treasury.
/// Sets the maturity date for the yDai, in unix time.
/// Initializes chi and rate at maturity time as 1.0 with 27 decimals.
constructor(
    address vat_,
    address pot_,
    address treasury_,
    uint256 maturity_,
    string memory name,
    string memory symbol
) public ERC20(name, symbol) {
    _vat = IVat(vat_);
    _pot = IPot(pot_);
    _treasury = ITreasury(treasury_);
    maturity = maturity_;
    chi0 = UNIT;
    rate0 = UNIT;
}
```

Figure 4.1: The constructor of the YDai contract.

### Exploit Scenario

The Yield Protocol team deploys a new suite of YDai contracts with a variety of target maturity dates. One of the maturity timestamps contains a typo, and the maturity date is set for 10 years from now instead of the intended 6 months. Before this is noticed by either the team or the community, users begin locking up fyDAI in this longer-term contract.

### Recommendation

Short term, add checks to the YDai contract constructor to ensure maturity timestamps fall within an acceptable range. This will prevent maturity dates from being mistakenly set in the past or too far in the future.

Long term, always perform validation of parameters passed to contract constructors. This will help detect and prevent errors during deployment.

## 5. Delegates can be added or removed repeatedly to bloat logs

Severity: Informational  
Type: Auditing and Logging  
Target: helpers/Delegable.sol

Difficulty: Low  
Finding ID: TOB-YP-005

### Description

Several contracts in the Yield Protocol system inherit the `Delegable` contract. This contract allows users to delegate the ability to perform certain operations on their behalf to other addresses. When a user adds or removes a delegate, a corresponding event is emitted to log this operation. However, there is no check to prevent a user from repeatedly adding or removing a delegation that is already enabled or revoked, which could allow redundant events to be emitted repeatedly.

```
/// @dev Enable a delegate to act on the behalf of caller
function addDelegate(address delegate) public {
    delegated[msg.sender][delegate] = true;
    emit Delegate(msg.sender, delegate, true);
}

/// @dev Stop a delegate from acting on the behalf of caller
function revokeDelegate(address delegate) public {
    delegated[msg.sender][delegate] = false;
    emit Delegate(msg.sender, delegate, false);
}
```

Figure 5.1: The `addDelegate` and `revokeDelegate` function definitions.

### Exploit Scenario

Alice calls `addDelegate` on the Pool contract with Bob's address several hundred times. For each call, a new event is emitted. This bloats the event logs for the contract and degrades performance of off-chain systems that ingest these events.

### Recommendation

Short term, add a `require` statement to check that the delegate address is not already enabled or disabled for the user. This will ensure log messages are only emitted when a delegate is activated or deactivated.

Long term, review all operations and avoid emitting events in repeated calls to idempotent operations. This will help prevent bloated logs.

## 6. Withdrawing from the Controller allows accounts to contain dust

Severity: Low  
Type: Data Validation  
Target: Controller.sol

Difficulty: Low  
Finding ID: TOB-YP-006

### Description

The `withdraw` operation can break the assumption that no account can contain dust for certain collaterals.

The `aboveDustOrZero` function enforces an invariant that prevents accounts from holding an amount of collateral smaller than DUST (0.025 ETH):

```
/// @dev Return if the collateral of an user is between zero and the dust level
/// @param collateral Valid collateral type
/// @param user Address of the user vault
function aboveDustOrZero(bytes32 collateral, address user) public view returns (bool) {
    uint256 postedCollateral = posted[collateral][user];
    return postedCollateral == 0 || DUST < postedCollateral;
}
```

Figure 6.1: `aboveDustOrZero` function in `Controller.sol`.

While this function is correctly used in the `post` operation, it fails to enforce this invariant in `withdraw`:

```
function withdraw(bytes32 collateral, address from, address to, uint256 amount)
    public override
    validCollateral(collateral)
    onlyHolderOrDelegate(from, "Controller: Only Holder Or Delegate")
    onlyLive
{
    posted[collateral][from] = posted[collateral][from].sub(amount); // Will revert if
not enough posted

    require(
        isCollateralized(collateral, from),
        "Controller: Too much debt"
    );
}
```



```

    if (collateral == WETH){
        require(
            aboveDustOrZero(collateral, to),
            "Controller: Below dust"
        );
        _treasury.pullWeth(to, amount);
    } else if (collateral == CHAI) {
        _treasury.pullChai(to, amount);
    }

    emit Posted(collateral, from, -toInt256(amount));
}

```

Figure 6.2: withdraw function in Controller.sol.

The invariant is enforced for the to address (which is not modified) instead of the from address.

### Exploit Scenario

Alice calls `withdraw` on the Controller assuming that it cannot leave a positive amount of WETH that is lower than DUST in her account. However, the transaction succeeds, leaving the contract in an invalid state.

### Recommendation

Short term, enforce the `aboveDustOrZero` function in the from address instead of the to address, after modifying its balance during the `withdraw` call. This will ensure the correct address has an appropriate balance after calls to `withdraw`.

Long term, use Echidna or Manticore to properly test the contract invariants. Automated testing can cover a wide array of inputs that unit testing may miss.

## 7. Solidity compiler optimizations can be dangerous

Severity: Undetermined

Difficulty: Low

Type: Undefined Behavior

Finding ID: TOB-YP-007

Target: `truffle-config.js`, `buidler.config.ts`

### Description

Yield Protocol 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, 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](#).

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 has 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 `<>` contracts.

### Recommendation

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. Lack of chainID validation allows signatures to be re-used across forks

Severity: High

Type: Access Controls

Target: helpers/ERC20Permit.sol

Difficulty: High

Finding ID: TOB-YP-008

### Description

YDai implements the draft ERC 2612 via the ERC20Permit contract it inherits from. This allows a third party to transmit a signature from a token holder that modifies the ERC20 allowance for a particular user. These signatures used in calls to permit in ERC20Permit do not account for chainsplits. The chainID is included in the domain separator. However, it is not updatable and not included in the signed data as part of the permit call. As a result, if the chain forks after deployment, the signed message may be considered valid on both forks.

```
bytes32 hashStruct = keccak256(  
    abi.encode(  
        PERMIT_TYPEHASH,  
        owner,  
        spender,  
        amount,  
        nonces[owner]++,  
        deadline  
    )  
);
```

Figure 8.1: The reconstruction of the permit parameters in ERC20Permit as signed by the owner, notably omitting the chainID.

### Exploit Scenario

Bob has a wallet holding fyDAI. An EIP is included in an upcoming hard fork that has split the community. After the hard fork, a significant user base remains on the old chain. On the new chain, Bob approves Alice to spend some tokens via a call to permit. Alice, operating on both chains, replays the permit call on the old chain and is able to steal some of Bob's fyDAI.

### Recommendation

Short term, include the chainID opcode in the permit schema. This will make replay attacks impossible in the event of a post-deployment hard fork.

Long term, document and carefully review any signature schemas, including their robustness to replay on different wallets, contracts, and blockchains. Make sure users are aware of signing best practices and the danger of signing messages from untrusted sources.

## 9. Permit opens the door for griefing contracts that interact with the Yield Protocol

Severity: Informational  
Type: Timing  
Target: ERC20Permit.sol

Difficulty: Low  
Finding ID: TOB-YP-009

### Description

The permit function can be front-run to break the workflow from third-party smart contracts.

The YDai contract implements permit, which allows the ERC20 allowance of a user to be changed based on a signature check using ecrecover:

```
function permit(address owner, address spender, uint256 amount, uint256 deadline, uint8
v, bytes32 r, bytes32 s) public virtual override {
    require(deadline >= block.timestamp, "ERC20Permit: expired deadline");

    bytes32 hashStruct = keccak256(
        abi.encode(
            PERMIT_TYPEHASH,
            owner,
            spender,
            amount,
            nonces[owner]++,
            deadline
        )
    );

    bytes32 hash = keccak256(
        abi.encodePacked(
            '\x19\x01',
            DOMAIN_SEPARATOR,
            hashStruct
        )
    );

    address signer = ecrecover(hash, v, r, s);
    require(
```

```

        signer != address(0) && signer == owner,
        "ERC20Permit: invalid signature"
    );

    _approve(owner, spender, amount);
}
}

```

*Figure 9.1: permit function in ERC20Permit.sol.*

While this function is correctly implemented in terms of functionality, there is a potential security issue users must be aware of when developing contracts to interact with fyDAI tokens:

```

## Security Considerations

```

```

Though the signer of a `Permit` may have a certain party in mind to submit their
transaction, another party can always front run this transaction and call `permit` before
the intended party. The end result is the same for the `Permit` signer, however.

```

*Figure 9.2: Security considerations for ERC2612.*

### **Exploit Scenario**

Alice develops a smart contract that leverages permit to perform a transferFrom of fyDAI without requiring a user to call approve first. Eve monitors the blockchain and notices this call to permit. She observes the signature and replays it to front-run her call, which produces a revert in Alice's contract and halts its expected execution.

### **Recommendation**

Short term, properly document the possibility of griefing permit calls to warn users interacting with fyDAI tokens. This will allow users to anticipate this possibility and develop alternate workflows in case they are targeted by it.

Long term, carefully monitor the blockchain to detect front-running attempts.

## 10. Pool initialization is unprotected

Severity: Low  
Type: Access Controls  
Target: Pool.sol

Difficulty: High  
Finding ID: TOB-YP-010

### Description

The Yield Pool contract implements a simple initialization system that can be abused by any user.

The Pool contract needs to be initialized using an `init` function:

```
    /// @dev Mint initial liquidity tokens.
    /// The liquidity provider needs to have called `dai.approve`
    /// @param daiIn The initial Dai liquidity to provide.
    function init(uint128 daiIn)
        external
        beforeMaturity
    {
        require(
            totalSupply() == 0,
            "Pool: Already initialized"
        );
        // no yDai transferred, because initial yDai deposit is entirely virtual
        dai.transferFrom(msg.sender, address(this), daiIn);
        _mint(msg.sender, daiIn);
        emit Liquidity(maturity, msg.sender, msg.sender, -toInt256(daiIn), 0,
            toInt256(daiIn));
    }
```

Figure 10.1: `init` function in Pool.sol.

However, there are some concerns regarding this code:

- Any user can call `init` and provide some initial liquidity.
- If at some point all the tokens are burned and the total supply is zero, it can be called again.
- If the pool is not initialized before the `fyDAI` maturity date, it cannot be initialized.

### Exploit Scenario

Alice deploys the Pool contract. Eve is monitoring the blockchain transactions and notices that Alice has started the deployment. Before Alice can perform any other transaction, Eve calls `init` with the minimal amount of tokens (1), so Alice is forced to provide liquidity using the `mint` function or re-deploy the contract.

**Recommendation**

Short term, consider restricting calls to `init` to the contract owner and enforce that it can only be called once. This will ensure initialization is carried out as Yield intends.

Long term, review the rest of the components to make sure they are suitable for their purpose and can be used only for their intended purpose.

## 11. Computation of DAI/fyDAI to buy/sell is imprecise

Severity: Undetermined  
Type: Data Validation  
Target: YieldMath.sol

Difficulty: Medium  
Finding ID: TOB-YP-011

### Description

It is unclear if the functions used to determine how many DAI or fyDAI a user must buy or sell (given the current total supply and reserves) works as expected or not.

The YieldMath provides several functions to calculate the amount of DAI or fyDAI, given the state of the pool. For instance, yDaiOutForDaiIn computes the amount of fyDAI a user would get for a given amount of DAI:

```
function yDaiOutForDaiIn (
    uint128 daiReserves, uint128 yDAIReserves, uint128 daiAmount,
    uint128 timeTillMaturity, int128 k, int128 g)
internal pure returns (uint128) {
    // t = k * timeTillMaturity
    int128 t = ABDKMath64x64.mul (k, ABDKMath64x64.fromUInt (timeTillMaturity));

    // a = (1 - gt)
    int128 a = ABDKMath64x64.sub (0x10000000000000000, ABDKMath64x64.mul (g, t));
    require (a > 0, "YieldMath: Too far from maturity");

    // xdx = daiReserves + daiAmount
    uint256 xdx = uint256 (daiReserves) + uint256 (daiAmount);
    require (xdx < 0x100000000000000000000000000000000, "YieldMath: Too much Dai in");

    uint256 sum =
        uint256 (pow (daiReserves, uint128 (a), 0x10000000000000000)) +
        uint256 (pow (yDAIReserves, uint128 (a), 0x10000000000000000)) -
        uint256 (pow (uint128(xdx), uint128 (a), 0x10000000000000000));
    require (sum < 0x100000000000000000000000000000000, "YieldMath: Insufficient yDAI
reserves");

    uint256 result = yDAIReserves - pow (uint128 (sum), 0x10000000000000000, uint128 (a));
    require (result < 0x100000000000000000000000000000000, "YieldMath: Rounding induced
```



```

error");

    return uint128 (result);
}

```

Figure 11.1: yDaiOutForDaiIn function in YieldMath.sol.

YieldMath also provides another function called daiInForYDaiOut to calculate the amount of DAI a user would have to pay for a certain amount of fyDAI. These two functions should behave as inverses:

```

function DaiInOut(uint128 daiReserves, uint128 yDAIReserves, uint128 daiAmount,
uint128 timeTillMaturity) public {
    daiReserves = 1 + daiReserves % 2**112;
    yDAIReserves = 1 + yDAIReserves % 2**112;
    daiAmount = 1 + daiAmount % 2**112;
    timeTillMaturity = 1 + timeTillMaturity % (12*4*2 weeks); // 2 years

    require(daiReserves >= 1024*oneDAI);
    require(yDAIReserves >= daiReserves);

    uint128 daiAmount1 = daiAmount;
    uint128 yDAIAmount = YieldMath.yDaiOutForDaiIn(daiReserves, yDAIReserves, daiAmount1,
timeTillMaturity, k, g);

    require(
        sub(yDAIReserves, yDAIAmount) >= add(daiReserves, daiAmount1),
        "Pool: yDai reserves too low"
    );

    uint128 daiAmount2 = YieldMath.daiInForYDaiOut(daiReserves, yDAIReserves, yDAIAmount,
timeTillMaturity, k, g);

    require(
        sub(yDAIReserves, yDAIAmount) >= add(daiReserves, daiAmount2),
        "Pool: yDai reserves too low"
    );
}

```

```
    assert(equalWithTol(daiAmount1, daiAmount2));  
}
```

*Figure 11.2: Echidna property to test functions in YieldMath.sol.*

However, these two functions do not behave as the inverse of each other, as Echidna was able to show. If this property is called with the following parameters...

- daiReserves: 155591140918329338279663772
- yDAIReserves: 12011620595696883763591137622155
- daiAmount: 4726945
- timeTilMaturity: 974285

...the resulting DAI amounts will differ with more than 10 DAI of difference.

### **Exploit Scenario**

Alice uses the pool to buy/sell DAI/fyDAI, but the resulting amount is unexpected.

### **Recommendation**

Short term, review the specification of the YieldMath functions and make sure it matches the implementation. Use Echidna to validate the implementation.

Long term, develop robust unit and automated test suites for the custom math functions. This will help to ensure the correct functionality of this complex arithmetic.

## 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
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Testing	Related to test methodology or test coverage
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

<b>Difficulty Levels</b>	
<b>Difficulty</b>	<b>Description</b>
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 Maturity Classifications

Code Maturity Classes	
Category Name	Description
Access Controls	Related to the authentication and authorization of components.
Arithmetic	Related to the proper use of mathematical operations and semantics.
Assembly Use	Related to the use of inline assembly.
Centralization	Related to the existence of a single point of failure.
Upgradeability	Related to contract upgradeability.
Function Composition	Related to separation of the logic into functions with clear purpose.
Front-Running	Related to resilience against front-running.
Key Management	Related to the existence of proper procedures for key generation, distribution, and access.
Monitoring	Related to use of events and monitoring procedures.
Specification	Related to the expected codebase documentation.
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.).

Rating Criteria	
Rating	Description
Strong	The component was reviewed and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

## C. 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

- **Consider allowing the owner to adjust the FEE or DUST constants in the protocol.** These are specified in Ether and may become very expensive if the price of this cryptocurrency continues to grow.

### Controller

- **Consider reverting if a call to powerOf uses an invalid collateral.** Reverting if a user tries to obtain the borrowing power with an invalid collateral will prevent invalid results if the user interacts with the Controller contract in an unexpected way.
- **Consider adding FEE/DUST constants.** Properly naming constants will make the codebase easier to maintain, modify, and audit.

### Liquidations

- **Consider removing the dai state variable if it is unused.** Removing unused code will make the codebase easier to maintain, modify, and audit.
- **Address outstanding TODOs in the codebase** or open issues to ensure they are tracked properly and not overlooked when deploying the system. Develop test cases to cover the scenario in which a user is liquidated multiple times, and ensure the expected behavior is carried out.

### Treasury:

- **Consider adding a flashy warning to users in case they want to transfer collateral directly to the Treasury contract address.** If collateral is transferred directly to the Treasury contract address, it will be locked there until the MakerDAO shutdown, so users should be warned about this.

### Pool:

- **Consider reviewing the code comments on the burn function.** The comment mentions that this function requires the use of approve; however, there is no use of transferFrom, so it should not be needed. Keeping documentation up to date will make the codebase easier to maintain, modify, and audit.

## D. Fix Log

Yield addressed issues TOB-YP-001 to TOB-YP-011 in their codebase as a result of our assessment. Each of the fixes was verified by Trail of Bits, and the reviewed code is available in git revision [642b33b166a6b740f907a0e6d85dbd0d87451c77](https://github.com/yieldprotocol/yield-protocol/commit/642b33b166a6b740f907a0e6d85dbd0d87451c77).

ID	Title	Severity	Status
01	Flash minting can be used to redeem fyDAI	Medium	Fixed
02	Permission-granting is too simplistic and not flexible enough	Low	Mitigated
03	pot.chi() value is never updated	Low	Risk accepted
04	Lack of validation when setting the maturity value	Low	Fixed
05	Delegates can be added or removed repeatedly to bloat logs	Informational	Fixed
06	Withdrawing from the controller allows accounts to contain dust	Low	Fixed
07	Solidity compiler optimizations can be dangerous	Undetermined	Risk accepted
08	Lack of chainID validation allows signatures to be re-used across forks	High	Not fixed
09	Permit opens the door for grieving contracts that interact with the Yield Protocol	Informational	WIP
10	Pool initialization is unprotected	Low	Risk accepted
11	Computation of DAI/fyDAI to buy/sell is imprecise	Undetermined	Fixed



## Detailed fix log

This section includes brief descriptions of fixes implemented by Yield after the end of this assessment that were reviewed by Trail of Bits.

### **Finding 1: Flash minting can be used to redeem fyDAI**

Fixed by disallowing a call to redeem in the fyDAI token contract (PR [246](#)) or a call to redeem in Unwind (PR [294](#)) during flash minting.

### **Finding 2: Permission-granting is too simplistic and not flexible enough**

This is mitigated by providing [an external script](#) that allows any user to audit the per-function permissions.

### **Finding 3: `pot.chi()` value is never updated**

Risk accepted. Yield said:

This is intended behavior, to reduce gas costs (which are very likely to exceed the unrecognized accrued interest) and allow these functions to be view. (A previous implementation did `callpot.drip()`; this was removed). We also believe that the Severity here should be marked as "Low" as there is no risk associated with user funds. We will monitor this issue and, if interest accumulation is not being done frequently enough, can provide an external mechanism for users to call `pot.drip`` before interacting with the fyDAI contracts.

### **Finding 4: Lack of validation when setting the maturity value**

Fixed by verifying the maturity date in the YDai constructor (PR [251](#)).

### **Finding 5: Delegates can be added or removed repeatedly to bloat logs**

Fixed by disallowing re-adding and re-removing delegates (PRs [252](#) and [293](#)).

### **Finding 6: Withdrawing from the controller allows accounts to contain dust**

Fixed by enforcing the `aboveDustOrZero` property in all the accounts in the Controller (PR [268](#)).

### **Finding 7: Solidity compiler optimizations can be dangerous**

Risk accepted. Yield said they will continue using the optimizer with 200 runs.

### **Finding 8: Lack of chainID validation allows signatures to be re-used across forks**

Not fixed.

### **Finding 9: Permit opens the door for griefing contracts that interact with the Yield Protocol**

This fix is still in progress. Yield said they will add a note about it in their documentation to warn the user.

**Finding 10: Pool initialization is unprotected**

Risk accepted. Yield said:

This is a feature, we want anyone to be able to initialize the pool, even though it will probably be us calling it. We do not mind somebody else frontrunning the initialization transaction. Should be marked as informational.

**Finding 11: Computation of DAI/fyDAI to buy/sell is imprecise**

Fixed by [adding a flat fee](#) to compensate for the loss of precision and by [limiting trades to valid uint128 values](#). Also, Yield determined expected parameters for the liquidity amounts in order to define exactly how this issue could affect the pool.